Document Title	Specification of Operating System
Document Owner	AUTOSAR GbR
Document Responsibility	AUTOSAR GbR
Document Version	2.1.0
Document Status	Draft
Part of Release	2.1
Revision	0014

Document Change History

Date	Version	Changed by	Change Description
31.01.2007	2.1.0	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
28.04.2006	2.0.0	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
28.06.2005	1.0.0	AUTOSAR Administration	Initial Release

Release Notes

Compatibility considerations with respect to current release

There are potential problems in the compatibility of the OS configuration and the AUTOSAR build configuration and process.

Compatibility with previous version might be compromised due to a modified API for Schedule Tables which includes absolute and relative start to underlying counter value and synchronous start relative to a global time.

Errata and known deficiencies

The following topics are currently under discussion and may cause changes in future versions of the specification.

- DisableInterruptSource() and EnableInterruptSource() are currently only available for configurations using timing protection. The services may either be removed OR be made available in all configurations.
- Fault handling (via ProtectionHook()) may be extended by adding more possible actions (besides the killing of Task/ISR/OS Application and the reboot) to improve flexibility
- The configuration language may change (XML as configuration format instead of, or in addition to, OIL). All other AUTOSAR modules use XML; OS is the only exception.
- Timing protection strategy may change. Currently the specification uses the timeframe/budget method, but it is still open if this is sufficient, or if real deadlines are required.
- The configuration parameter unit for time may change: either usage of real times (“nano seconds”) OR hardware timer ticks
- The configuration of memory entities for protection purposes is currently implementation specific, it may be standardized.

• Figure 10 contains old configuration parameters
• Definition of timeframe is missing in glossary
• Improved definition of execution budget is needed

Known and potential problems resulting from known deficiencies

There are potential problems in:
• integrating the OS into an AUTOSAR configuration and build process due to problems in the configuration language,
• timing and memory protection in the presence of faulty TASKs and ISRs,
• interrupt control (DisableInterruptSource() and EnableInterruptSource()),
• synchronising TASKs to external events.

Changes planned for next release

The above deficiencies will be addressed in the next release.

Disclaimer

Advice to users of AUTOSAR Specification Documents:

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).
Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

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) according to requirement BSW00322 in [3].

This document uses the industry standard OSEK OS [7] (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, this OSEK OS.

2 Acronyms and abbreviations

Abbreviation	*Description*
API	Application Programming Interface
BSW	Basic Software Requirement
COM	Communications
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MCU	Microcontroller Unit
MPU	Memory Protection Unit
NM	Network Management
OIL	OSEK Implementation Language
OS	Operating System
OSEK/VDX	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
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.	
Arrival Rate (for ISRs)	For ISRs the arrival rate is the number of ISR invocations within a specific time frame.	
Counter	Counters can be classified in 2 types:	
	Hardware Counter	A counter that is advanced by hardware (e.g. timer). The counter value is “in hardware”.
	Software Counter	A counter which is incremented by making the `IncrementCounter()` API call. The counter value is “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 Fig. 1.	
Delta	Duration between the last expiry point of a schedule table and the point at which the schedule table can be repeated.	
Execution Time	The time a task spends in the `RUNNING` state without entering the `SUSPENDED` or `WAITING` state. For ISRs it is the time from the first to the last instruction. For Tasks/ISRs it excludes all preemptions due to higher priority Tasks/ISRs executing in preference. The execution time includes the time spent in the error, pretask and posttask hooks and the time spent making OS service calls.	
Execution Time Budget	Maximum permitted execution time of a Task in a specific time frame.	
Expiry Point	The point on a Schedule Table at which one or more Tasks are activated and/or one or more events set. A Task can be activated at most once per expiry point. An event can be set at most once per expiry point.	

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).
Interarrival Time	Time between successive activations of the same Task/ISR see Fig. 1.	
Interrupt Lock Time	The longest time all ISR Category 2 or ISR Category 1 and 2 can be disabled.	
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.	
Kill OS-Application	The operating system frees all system objects, e.g. kills Tasks, disables interrupts, etc., which are associated to the OS-Application. OS-Application and internal variables are potentially left in an undefined state.	
Kill task/ISR	The OS terminates the Task/Category 2 ISR and does a cleanup of its held resources. For details see OS108 and OS109.	
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.	
Memory Protection Unit	A »MPU« (Memory Protection Unit) enables memory partitioning with individual protection attributes, whereas a »MMU« (Memory Management Unit) involves the mapping of physical memory to virtual addresses.	
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.
OS-Application	A block of software including Tasks, interrupts, hooks and trusted functions that form a cohesive functional unit. Only trusted applications can provide trusted functions.	
	Trusted	An OS-Application that is executed in privileged mode and has unrestricted access to the API and hardware resources.
	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.	
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 or software trap (division by zero, illegal instruction etc.).
Resource Lock Time	Longest time an OSEK resource is held by a Task/ISR.	
Response Time	The time between a Task/ISR being made ready to execute and generating a specified response. The time includes all preemptions. See Fig. 1.	
Restart an OS-Application	An OS-Application is restarted after self-termination or being killed because of a protection error. When an OS-Application is restarted, the OS activates the configured RESTARTTASK.	
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 exists 4 defined groups of features which are named scalability classes. For details see chapter 7.8	

Schedule Table	Encapsulation of a statically defined set of expiry points. For non-periodic schedule tables the table ends after the configured length. For period schedule tables the length of the period defines the end of the schedule table.	
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.
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
	stop symbol	tags the end of a memory section
Synchronization of schedule tables with global time	Synchronization with global time is achieved, if the expiry points of the schedule table are processed within an absolute deviation from global time that is smaller than or equal to a precision threshold.	
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)	WCET for basic tasks: WCET is the maximum execution time the task spends in the RUNNING state before entering the SUSPENDED state. It is reset at the start of each SUSPENDED->READY transition. WCET for extended tasks: WCET is the maximum execution time the task spends in the RUNNING state before entering either the WAITING state or the SUSPENDED state. It is reset at the start of both a SUSPENDED->READY transition and a WAITING->READY transition. WCET for ISR: Is the maximum execution time of the interrupt service routine. It is reset after the ISR has finished.	
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.	

Tab. 1: Glossary of Terms

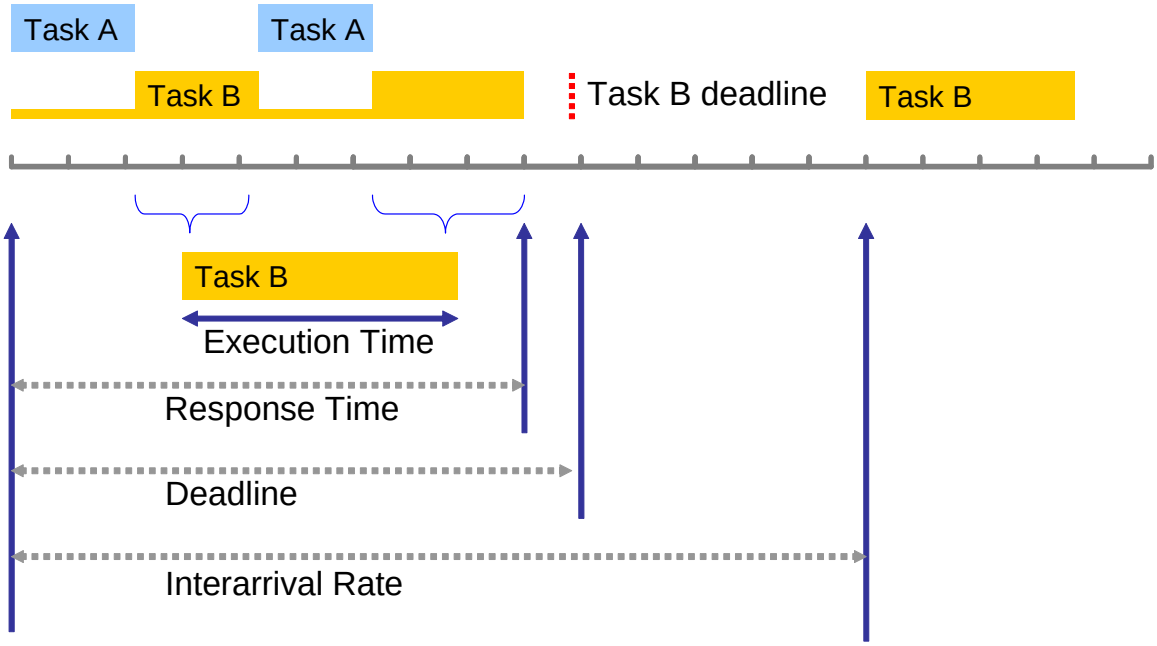

Fig. 1: Definition of Timing Terminology

3 Related documentation

3.1 Input documents

[1] Layered Software Architecture
https://svn.autosar.org/repos/10Releases/
AUTOSAR_LayeredSoftwareArchitecture.pdf

[2] Requirements on Operating System
https://svn.autosar.org/repos/10Releases/
AUTOSAR_SRS_OS.pdf

[3] General Requirements on Basic Software Modules
https://svn.autosar.org/repos/10Releases/
AUTOSAR_SRS_General.pdf

[4] Specification of the Virtual Functional Bus
https://svn.autosar.org/repos/10Releases/
AUTOSAR_VirtualFunctionBus.pdf

[5] Requirements on Software FreeRunningTimer
https://svn.autosar.org/repos/21Releases/
AUTOSAR_SRS_SWFreeRunnningTimer.pdf

[6] Specification of GPT Driver
https://svn.autosar.org/repos/20Releases/
AUTOSAR_SWS_GPT_Driver.pdf

3.2 Related standards and norms

3.2.1 OSEK

[7] OSEK/VDX Operating System, Version 2.2.3, See:
www.osek-vdx.org

[8] OSEK/VDX Time-Triggered Operating System, Version 1.0, July 24, 2001. See:
www.osek-vdx.org

[9] OSEK Implementation Language (OIL) V2.5, See:
www.osek-vdx.org

[10] ORTI (OSEK RunTime Interface), Part A Version 2.1.1, Part B Version 2.1.
See: www.osek-vdx.org

3.2.2 HIS

The HIS (Hersteller Initiative Software) documents are publicly available from www.automotive-his.de

[11] Requirements for Protected Applications under OSEK, Version 1, September 25, 2002.

[12] OSEK OS Extensions for Protected Applications, Version 1.0, July 27, 2003

3.2.3 ISO/IEC

[13] ISO/IEC 9899:1990 Programming Language – C

(Remark: The international ISO standard ISO/IEC 9899:1990, also sometimes simply called »C90«, describes the language C. It was introduced in 1990 and replaced the ANSI C standard that was introduced only one year before, that's why it is also called »C89«. C89 differs from ISO/IEC 9899:1990 essentially only by the copyright note.)

[14] ISO/IEC 9899:1999 Programming Language – C

(Remark: A revised version of the standard was published in 1999. It is officially ISO/IEC 9899:1999, but is more often referred to as »C99«.)

3.3 Company Reports, Academic Work, etc.

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

4 Constraints and assumptions

4.1 Existing Standards

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

- OSEK OS [7] provides a sufficiently flexible scheduling policy to schedule AUTOSAR systems.
- OSEK OS [7] is a mature specification and implementations are used in millions of ECUs worldwide.
- OSEK OS [7] does not provide sufficient support for isolating multi-source software components at runtime.
- OSEK OS [7] does not provide sufficient runtime support for demonstrating the absence of some classes of fault propagation in a safety-case.
- OSEKtime OS [8] and the HIS Protected OSEK [12] are immature specifications that contain concepts necessary for AUTOSAR and satisfy specific application domains. It is the purpose of this document to identify these needs and to recommend the use of parts (or all) of these specifications as appropriate.

4.2 AUTOSAR Configuration Process

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 interfacing between runnables and the basic software modules. The basic software modules consist of 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, in particular the protection features.

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 (OSAL). The interface to the OSAL is exactly that defined for the AUTOSAR OS.

4.3 Limitations

4.3.1 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 hardware dependency. 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.

4.3.2 Programming Language

The API of the operating system is defined as C89 [13] function calls or macros. If other languages are used they must adapt to the C interface. This is because C99 [14] allows for internal dynamic memory allocation during subroutine calls. Most automotive applications are static (non-heap based).

4.3.3 Miscellaneous

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

The operating system is only able to handle a single thread of execution at one time. It is therefore not able to manage software running on a multi-processor system.

4.4 Applicability to car domains

The operating system has the same design constraints regarding size and scalability under which the OSEK OS 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 direct dependencies on other modules, however:

- It is assumed that the operating system may directly manage the timer units.
- If the user needs to drive scheduling 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.

5.1 File structure

5.1.1 Code file structure

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

5.1.2 Header file structure

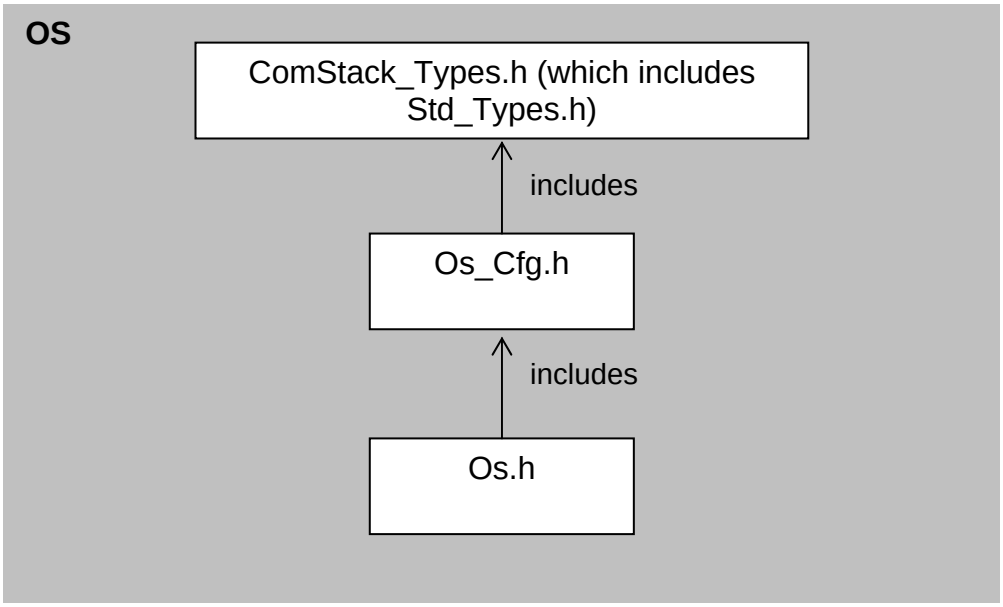

Fig. 2: Header file structure for the OS

The figure above contains the defined AUTOSAR header file hierarchy of the Operating System.

Users of the Operating Systems shall only include the **Os.h** file. If an implementation of the Operating System requires additional header files, it is free to include them.

The **Os.h** file is self containing, that means it will include all other header files which are required by **Os.h** (e.g include files for the GPT driver if required).

6 Requirements Traceability

This chapter contains references to requirements of other AUTOSAR documents.

Functional requirements	***Satisfied by SWS-Requirement***
[BSW159] Tool-based configuration	Not applicable (AUTOSAR OS uses OIL. OIL can be generated using tools.)
[BSW167] Static configuration checking	See chapter 11.2
[BSW171] Configurability of optional functionality	See chapter 10. Requirement focuses on implementation.
[BSW101] Initialization interface	`StartOS()`
[BSW00416] Sequence of Initialization	`StartOS()` is called by the user. Sequence of calls with other modules is not affected. Note that `StartOS()` does not return but starts the configured startup-hook(s) and Tasks.
[BSW00336] Shutdown Interface	`ShutdownOS()`
[BSW00323] API parameter checking	See Section 7.6.3
[BSW00383] List dependencies of configuration files [approved]	Not applicable (SWS has no dependencies for configuration files.)
[BSW00384] List dependencies to other modules [approved]	Not applicable (SWS has no dependencies to other modules.)
[BSW00345] Pre-compile-time configuration	Not applicable (Requirement for implementation)
[BSW00380] Separate C-Files for configuration parameters	Not applicable (Requirement for implementation)
[BSW00419] Separate C-Files for pre-compile time configuration parameters [approved]	Not applicable (Requirement for implementation)
[BSW00381] Separate configuration header file for pre-compile time parameters [approved]	Not applicable (Requirement for implementation)
[BSW00412] Separate H-File for configuration parameters [approved]	Not applicable (Requirement for implementation)
[BSW00406] Check module initialization	Not applicable (Requirement for implementation)
[BSW00407] Function to read out published parameters	Not applicable (Requirement for implementation)
[BSW004] Version check	Not applicable (Requirement for implementation)
[BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interface	Not applicable (AUTOSAR OS does not interact directly with SW-C.)
[BSW00424] BSW main processing function task allocation [BSW00425] Trigger conditions for schedulable objects [BSW00426] Exclusive areas in BSW modules [BSW00427] ISR description for BSW modules [BSW00428] Execution order dependencies of main processing functions [BSW00429] Restricted BSW OS functionality access [BSW00431] The BSW Scheduler module	Requirement for users of AUTOSAR OS especially together with RTE.

implements task bodies [BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path [BSW00433] Calling of main processing functions [BSW00434] The Schedule Module shall provide an API for exclusive areas	
[BSW00337] Classification of errors	Not applicable (AUTOSAR OS does not distinguish between “development” and “production” errors. See Section 7.10.
[BSW00409] Header files for production code error IDs	Not applicable (AUTOSAR OS does not distinguish between “development” and “production” errors. See Section 7.10.
[BSW00385] List possible error notifications	Not applicable (AUTOSAR OS does not distinguish between “development” and “production” errors. See Section 7.10.
[BSW00387] Specify the configuration class of callback function [approved]	See chapter 8.5 for details.
[BSW00388] Introduce containers	See chapter 10.2.
[BSW00389] Containers shall have names	See chapter 10.2.
[BSW00390] Parameter content shall be unique within the module	See chapter 10.2.
[BSW00391] Parameter shall have unique names	See chapter 10.2.
[BSW00392] Parameters shall have a type	See chapter 10.2.
[BSW00393] Parameters shall have a range	See chapter 10.2.
[BSW00394] Specify the scope of the parameters	See chapter 10.2.
[BSW00395] List the required parameters (per parameter)	See chapter 10.2.
[BSW00396] Configuration classes	See chapter 10.2.
[BSW00397] Pre-compile-time parameters	See chapter 10.2.
[BSW00398] Link-time parameters	See chapter 10.2.
[BSW00399] Loadable Post-build time parameters	See chapter 10.2.
[BSW00400] Selectable Post-build time parameters	See chapter 10.2.
[BSW00402] Published information	See chapter 10.2.
[BSW00338] Detection and Reporting of development errors	Not applicable (AUTOSAR OS calls the ErrorHook (defined by the OSEK OS specification [7]) and the ProtectionHook (see Section 7.7) in case of errors. It is possible to call Debug Error Tracer from these hook routines.)
[BSW00369] Do not return development error codes via API	Not applicable (AUTOSAR OS does not distinguish between “development” and “production” errors. In accordance with OSEK OS all possible errors are reported via the `ErrorHook()` and as return values of system services.)
[BSW00339] Reporting of production relevant error status [BSW00421] Reporting of production relevant error events	Not applicable (AUTOSAR OS calls the `ErrorHook()` (defined by the OSEK OS specification [7]) and the `ProtectionHook()` (see Section 7.7) in case of

	errors. It is possible to call the function inhibition or diagnostic event manager (DEM) and handle the debouncing from these hook routines.)
[BSW00422] Debouncing of production relevant error status	Not applicable (AUTOSAR OS calls the `ErrorHook()` (defined by the OSEK OS specification [7]) and the `ProtectionHook()` (see Section 7.7) in case of errors. It is possible to call the function inhibition or diagnostic event manager (DEM) and handle the debouncing from these hook routines.)
[BSW00420] Production relevant error event rate detection	Not applicable (AUTOSAR OS calls the `ErrorHook()` (defined by the OSEK OS specification [7]) and the `ProtectionHook()` (see Section 7.7) in case of errors. It is possible to call the function inhibition or diagnostic event manager (DEM) and handle the debouncing from these hook routines.)
[BSW00386] Configuration for detecting an error	Not applicable (AUTOSAR OS calls the `ErrorHook()` (defined by the OSEK OS specification [7]) and the `ProtectionHook()` (see Section 7.7) in case of errors. It is possible to call the function inhibition or diagnostic event manager (DEM) and handle the debouncing from these hook routines.)

Tab. 2: Traceability between AUTOSAR_SRS_General and those SWS-Requirements here

Functional requirements)	***Contradiction***
[BSW00344] Reference to link time configuration	Not applicable (AUTOSAR OS is a statically configured Operating System. The current version does not support link time configuration.)
[BSW00404] Reference to post build time configuration [BSW00405] Reference to multiple configuration sets	Not applicable (AUTOSAR OS does not support post build time configuration.)
[BSW170] Data for reconfiguration of AUTOSAR SW-Components [BSW00375] Notification of wake-up reason [BSW00417] Reporting of Error Events by Non-Basic Software [BSW168] Diagnostic interface of SW components	Not applicable (e.g. this module does not provide any wake-up reason)

Tab. 3: AUTOSAR_SRS_General requirements not fulfilled

Note: The AUTOSAR_SRS_General also contains non-functional requirements which are not listed in this chapter.

AUTOSAR_SRS_SWFreeRunningTimer [2]	***Satisfied by SWS-Requirement***
[SWFRT00019] Configure HW Timer Type	OS390
[SWFRT00020] Configuration/initialization of HW Timer	OS371, OS374
[SWFRT00021] Import Used HW Timer's Configuration	N/A since the GPT configuration currently uses XML and the OS uses OIL (for

	compatibility reasons). Tools may offer an automated mechanism, but there is no standardized way defined.
[SWFRT00022] State which HW Timer is used	OS370
[SWFRT00023] Set up Duration of one Tick	OS385
[SWFRT00024] Support different Ranges/Resolutions	OS375
[SWFRT00025] Set up Access Methods	OS383, OS392
[SWFRT00026] Set up Target Count Values	OS386
[SWFRT00028] Ensure Continuous Running Mode	OS373; Note that the OS can not check this directly because of different configuration formats (XML/OIL)
[SWFRT00029] Init Function	The init function of the OS is the StartOS service. If the GPT driver is used by the OS, the init function of the GPT driver has to be called before starting the OS via StartOS.
[SWFRT00030] Start with Zero	OS384
[SWFRT00031] Increment Counter	OS384
[SWFRT00032] Wrap Around	N/A. This requirement targets specific timer features. The OS should minimize the software access to timers (e.g. by using automatic reload features of the hardware).
[SWFRT00033] Read out Ticks	OS377
[SWFRT00034] Calculate Ticks Elapsed since given value	OS382
[SWFRT00041] Shutdown Function	There is no function to shutdown the timers which drive counters
[SWFRT00047] Convert Ticks to Time	OS393
[SWFRT00048] EcuM Modes	The OS is not aware of any EcuM modes and switching modes which influences timers (e.g. stop them or slow down counting) is neither recognized nor handled by the OS. It is the responsibility of the user to take care of this effects.

Tab. 4: Traceability between functional requirements of AUTOSAR_SRS_SWFreeRunningTimer (V0.6) and the SWS-Requirements in this document

Note that the SWFRT requirement document also contains non-functional requirements which are not listed here.

AUTOSAR_SRS_OS [2]	***Satisfied by SWS-Requirement***
[BSW097] Existing OSEK OS	OS001
[BSW11001] Object Grouping	OS114, OS056
[BSW098] Table based schedules	OS002, OS007
[BSW099] Switchable schedules	OS191
[BSW11002] Synchronization with global time	OS206, OS200, OS201, OS013, OS199, OS260, OS227
[BSW11003] Stack Monitoring	OS067, OS068
[BSW11005] Memory Write Access	OS207, OS208, OS195
[BSW11006] Data exchange	OS086, OS196, OS087
[BSW11007] Code Sharing	OS081
[BSW11000] Memory read access	OS026
[BSW11008] Timing Protection	OS028, OS089, OS033, OS037, OS048, OS064
[BSW11009] Protection of the OS	OS051, OS088, OS052, OS069, OS070, OS092, OS093

[BSW11010] Protection of OS-Applications	OS056
[BSW11011] Protecting the OS managed hardware	OS096, OS245
[BSW11012] Scalable Protection	OS241, OS240
[BSW11016] Scalability of the OS	OS241, OS240
[BSW11013] Error Notification	OS068, OS044, OS210, OS033, OS037, OS064, OS051, OS088, OS070, OS093, OS056, OS246
[BSW11014] Protection Error Handling	OS033, OS037, OS106, OS107, OS108, OS109, OS110, OS243, OS244,
[BSW11018] Interrupt services	OS299
[BSW11020] Interface for ticking counters	OS286
[BSW11021] Cascading counters	OS301
[BSW11019] Creation of Interrupt Vector Table	OS336

Tab. 5: Traceability between AUTOSAR_SRS_OS and those SWS-Requirements here

SWS-Requirement on an OS-Service	*Associated API*
OS016	8.4.1
OS097	8.4.3
OS358	8.4.9
OS347	0
OS006	8.4.10
OS191	8.4.11
OS012	8.4.12
OS199	8.4.16, 8.4.17
OS227, OS359	8.4.18
OS099	8.4.4,8.4.5,8.4.2
OS256	8.4.6
OS017	8.4.7
OS258	8.4.19
OS337	8.4.20
OS338	8.4.21
OS383	8.4.13, ,8.4.14
OS201	8.4.15

Tab. 6: Traceability between SWS-Requirements on services and associated API

7 Functional specification

For each section there is a “Background & Rational” subsection which gives general information and places the requirements in context, followed by a list of requirements.

7.1 Core OS

7.1.1 Background & Rationale

The OSEK/VDX Operating System [7] 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 angle 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. Interested readers should consult the OSEK specification [7] for details.

Basing AUTOSAR OS on OSEK OS means that legacy applications will be backward compatible. Applications written for OSEK OS will run on AUTOSAR OS. However, using some of the new features introduced by AUTOSAR OS require restrictions on the use of existing OSEK OS features. For example: it is too inefficient to achieve timing and memory protection for alarm callbacks. They are therefore not allowed in specific scalability classes (OS242). Additionally AUTOSAR OS extends some existing features, e.g. by driving counters directly through alarms (OS301).

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 measurments and the integration of GPT drivers. For more details see also [[5]] (SWFRT).

7.1.2 Requirements

OS001: The Operating System shall provide an API that is backward compatible with the OSEK OS API [7].

OS242: The Operating System must not allow Alarm Callbacks in Scalability classes 2, 3 and 4.

OS301: The Operating System shall provide the ability to increment a software counter as an alternative action on alarm expiry.

OS304: 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 .

OS299: The Operating System shall provide the services `DisableAllInterrupts()`, `EnableAllInterrupts()`, `SuspendAllInterrupts()`, `ResumeAllInterrupts()` prior to calling `StartOS()` and after calling `ShutdownOS()`. (It is assumed that the static variables of theses functions are initialized).

OS373: The Operating System shall assume that the selected GPT channels - which are used to drive counters - are running in continuous mode.

OS374: The Operating System shall handle all the initialization and configuration of timers used directly by the OS and not handled by the GPT driver.

OS383: The Operating System shall have a service to read out the tick value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software ticks when user drives counter).

OS392: The Operating System shall have a service to get the number of ticks between the current tick value and a previously read tick value.

OS384: The Operating System shall adjust the read out values of hardware timers (which drive counters) in a way that the lowest value is 0 and that consecutive reads are returning a larger tick value until the range of the timer exceeds and the timer is starting again at 0.

7.2 Schedule Tables

7.2.1 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 alarms that cannot be modified at runtime.

A schedule table consists of a series of expiry points each associated with one or more actions. For example: activate Task1 at 0ms, 3ms and 11ms; activate Task2 at 2ms, 6ms, 11ms and 29ms. Since the actions are statically configured, the OS does not need to provide services to manipulate the timing behaviour of individual expiry points on the schedule table.

A schedule table has a period that defines the time between successive starts of the schedule table. As schedule tables may be single-shot, allowing systems that require a phased sequence of task activations to be made in response to a system stimulus to be built.

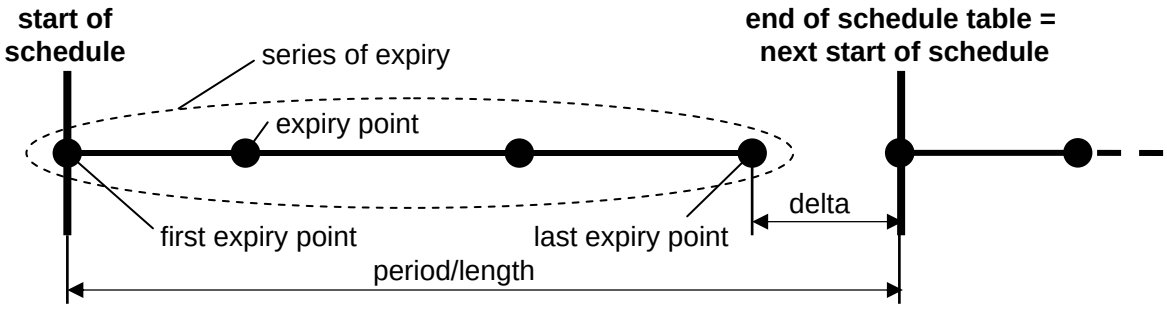

Fig. 3: Illustration of a schedule table

The Operating System shall be able to process at least one schedule table per counter at any given time. Implementations may offer more functionality e.g. driving more than one schedule table from one counter.

Schedule tables can be started either with a relative offset to the current counter value or at a given absolute value. The figure below illustrates the two different methods for a table which belongs to a counter (which counts from 0 to 9). Since the start (-offset) of the schedule table is provided directly via the API, each schedule table must have an expiry point at offset 0.

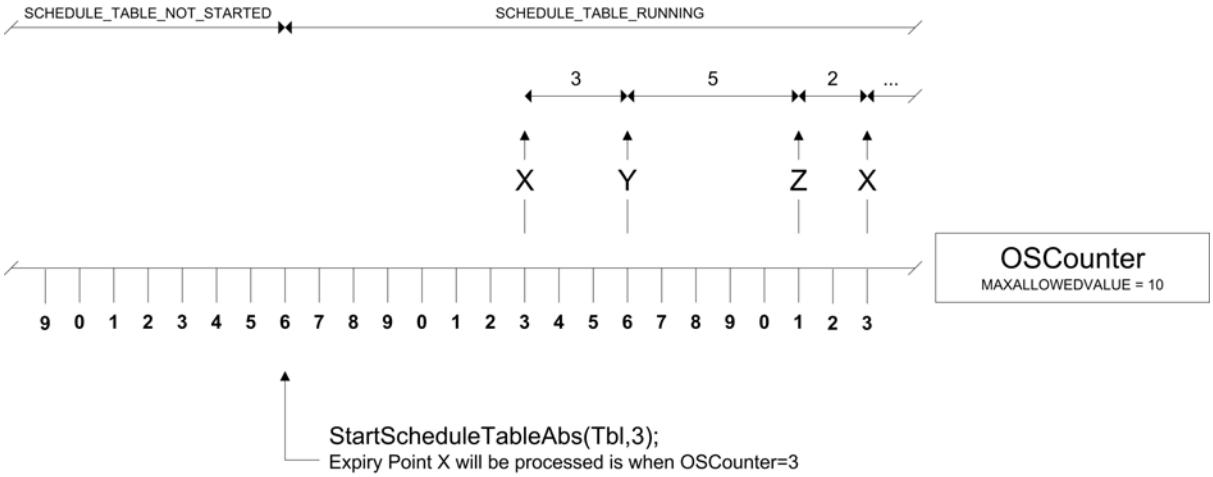

Fig. 4: Start a schedule table at an absolute value

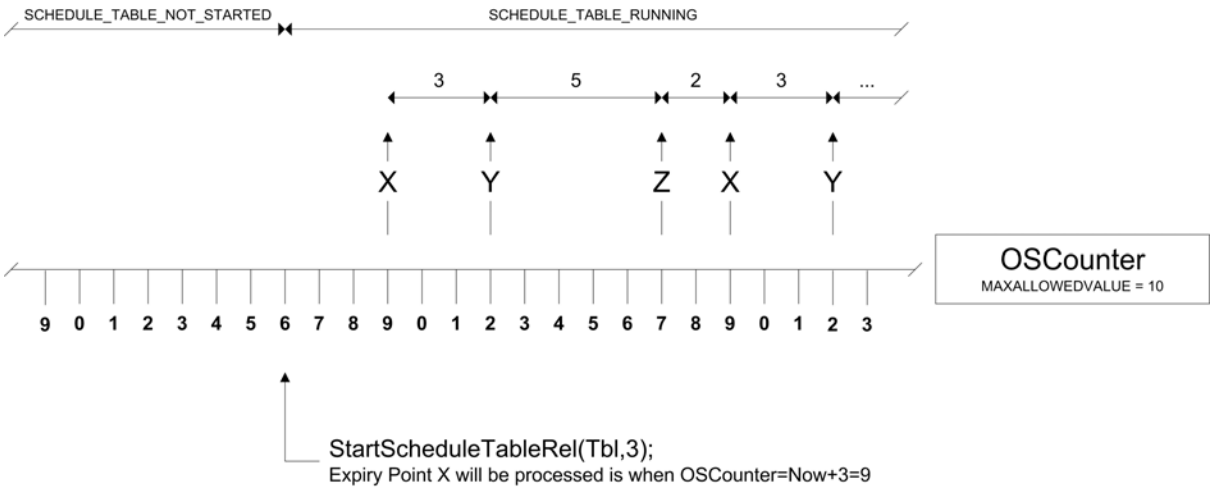

Fig. 5: Start a schedule table relative

7.2.2 Requirements

OS002: The OS shall include a mechanism for processing each expiry point on a schedule table in turn.

OS007: The Operating System shall permit multiple schedule tables to be processed concurrently.

OS360: The Operating System shall support multiple task activations and/or event setting at the same expiry point.

Manage schedule tables

OS358: The Operating System shall provide a service to start the processing of a schedule table at the first expiry point at an absolute time on the underlying counter

(The first expiry point shall be processed when the value of the underlying counter matches the value specified by the service).

OS347: The Operating System shall provide a service to start the processing of a schedule table at the first expiry point at a relative time on the underlying counter (The first expiry point shall be processed when the value of the underlying counter matches the value of the underlying counter plus the specified value at the time the service was called).

OS006: The Operating System shall provide a service to cancel the processing of a schedule table immediately at any point while the schedule table is running.

OS191: The Operating System shall provide a service to switch the processing from one schedule table to another schedule table (the switch is performed at the end of the current schedule table) .

OS359: The Operating System shall provide a service to query the state of a schedule table.

Singular or repeated processing

OS009: If `PERIODIC=FALSE`, the Operating System shall stop the processing of the schedule table after the configured `LENGTH` of the schedule table is reached.

OS194: If `PERIODIC=TRUE`, the Operating System shall repeat the processing of the schedule table from its start after the period has expired.

Driving schedule tables

OS012: The Operating System shall allow the processing of a schedule table to be driven by a software counter.

OS192: The Operating System shall allow the processing of a schedule table to be driven by a hardware counter.

7.3 Synchronization with Global Time

7.3.1 Background & Rationale

In some cases of system design it is important to ensure that computation occurring on different processing units happens synchronously. This is usually achieved by synchronizing the computation to a global (network) time base.

It is not for the OS to be responsible for providing a global (network) time source itself for some reasons:

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

Instead, the source of global time is provided to the OS by the user.

The principle behind synchronizing task activations/event settings to global time is as follows:

- define a schedule table that has a length/period equal to the modulus of the global time source (or divides the global time period in equal parts or is a multiple of the global time period)
- plan the task activations/event settings on the schedule table to occur at the required relative offsets
- ensure that the underlying counter source for the schedule table has the same resolution (granularity) as the global time source.
- keep the schedule table synchronized to the global time. Note that synchronization must be a property of the schedule table itself and not the underlying counter since the counter may be used as a tick/time base for other schedule tables and/or alarms.

The easiest way to ensure that a schedule table is synchronous to global time and starts synchronously with the global time source is to drive the table directly from the global time source. In this case the underlying counter **is** the actual global time. This is a good solution when it is essential that no processing happens if the global time is lost (i.e. the counter stops counting). The schedule table can be started synchronously using `StartScheduleTableSynchron(Table)` (followed by a `SyncScheduleTable(Table,Tick)` call) and because the underlying counter is the global time, the table will always be synchronous. The OS does not need to provide any additional mechanisms in this use case.

An alternative approach to synchronization is to drive the schedule table from a local (non-global) time source and then tell the table the global time and synchronize its processing to that time. This is a good approach when both communication and control functions need to be drive from the same schedule table. However, the local

time is likely to drift with respect to the global time so it becomes necessary to correct such a drift.
In this model there are 3 notions of time involved:

1. the local counter value (with a range from 0 to the maximum counter value, but a resolution matching the global time)
2. the schedule table time (which may be asynchronous to the local time with a range and resolution matching global time)
3. the global (network) time

Synchronization is the act of keeping the drift between schedule table time and global time within some acceptable bounds.

Under this approach the `StartScheduleTable[Abs|Rel]()` or `StartScheduleTableSynchron()` API calls can be used to start the schedule table processing either:

1. Synchronously to global time. In this case the schedule table is started only after the schedule table time is set to global time. In order to start a schedule table synchron, the user calls `StartScheduleTableSynchron()`.

2. Asynchronously to global time. In this case the user can start the schedule table at some arbitrary time according to the current local time using `StartScheduleTable[Abs|Rel]()`. This means that no attempt it made to start within acceptable drift limits - the table is started and drift can be corrected later.

When a table is started asynchronously to global time, or there is drift between the local time and global time, then it must be possible to do re-synchronization. This can be done in two ways:

1. Hard synchronization. If the global time has been provided then schedule time is set to global time at the end of the schedule table. If the schedule table is periodic then the next expiry point is processed at the absolute match before starting next round.

2. Smooth synchronization. If the global time has been provided then adjust delay between expiry points according to a configured adjustment value either backwards/forwards until synchronization is achieved.

The OS provides an API call to tell the schedule table the value of the global time. Since the schedule table length is equal to the modulus of the global time, the OS can determine the difference between local (schedule table) time and global (network) time and decide whether (or not) any action to achieve synchronization is required.

By way of illustration, consider the following schedule table that is periodic with 3 expiry points, X, Y and Z, with a schedule table length of 10:

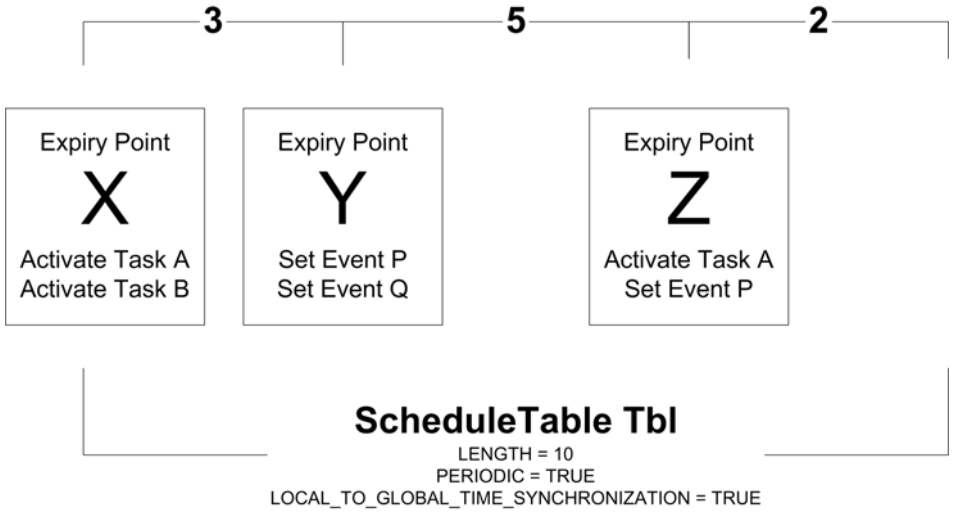

Fig. 6: Expiry points

The combinations of startup (asynchronous or synchronous) and (first) synchronization (hard or smooth) are illustrated below:

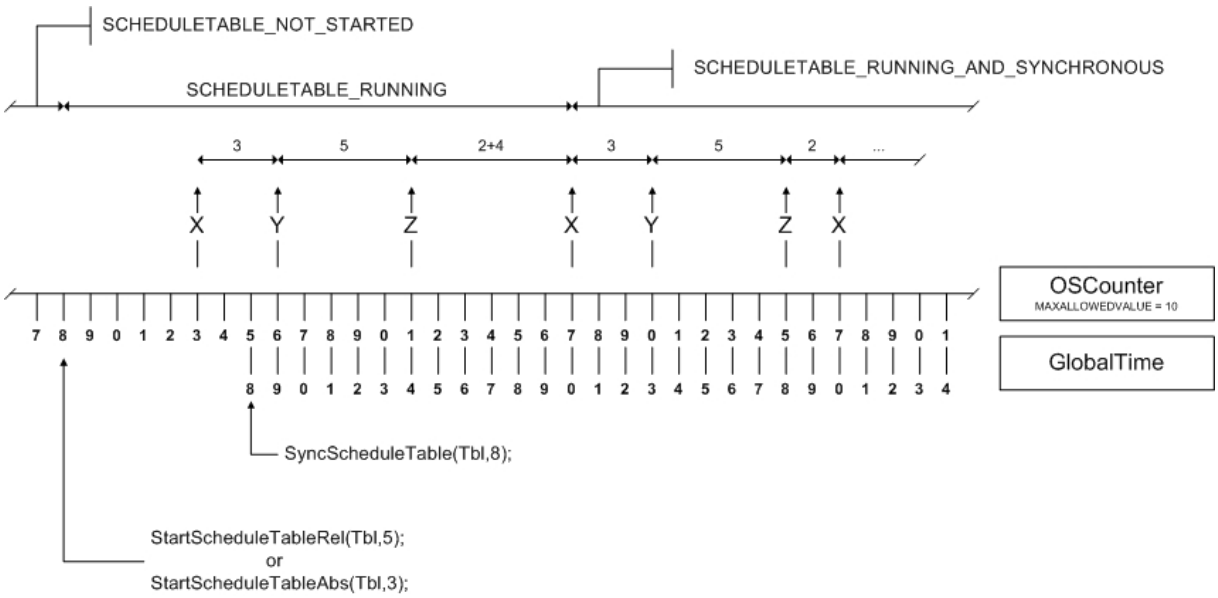

Fig. 7: Asynchronous start of a schedule table with hard synchronization

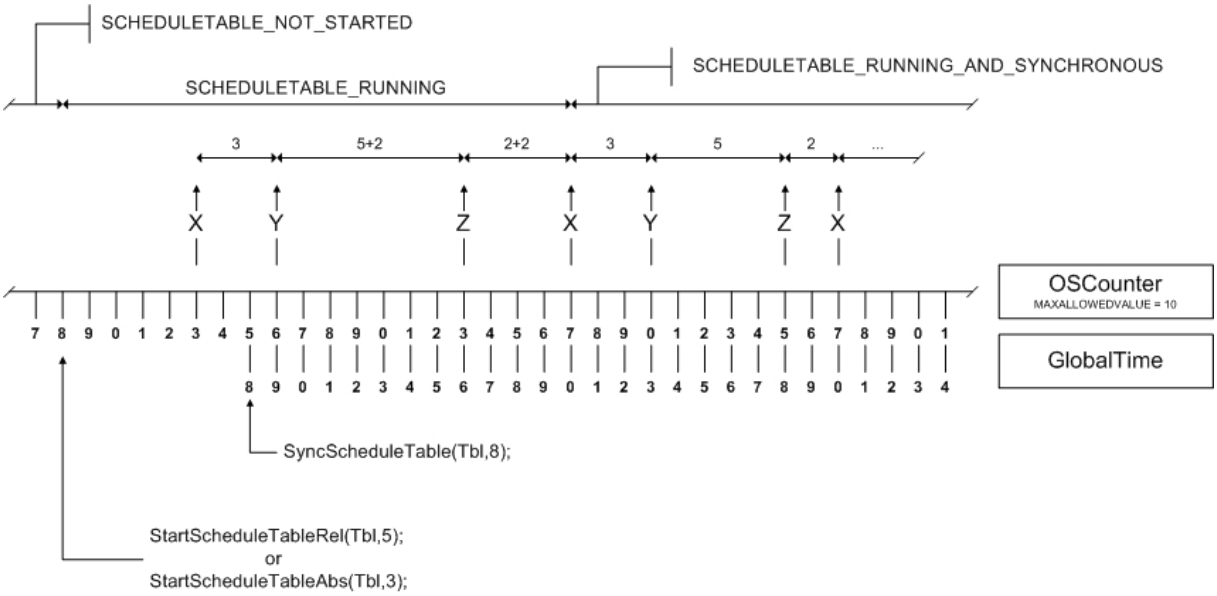

Fig. 8: Asynchronous start of a schedule table with smooth synchronization

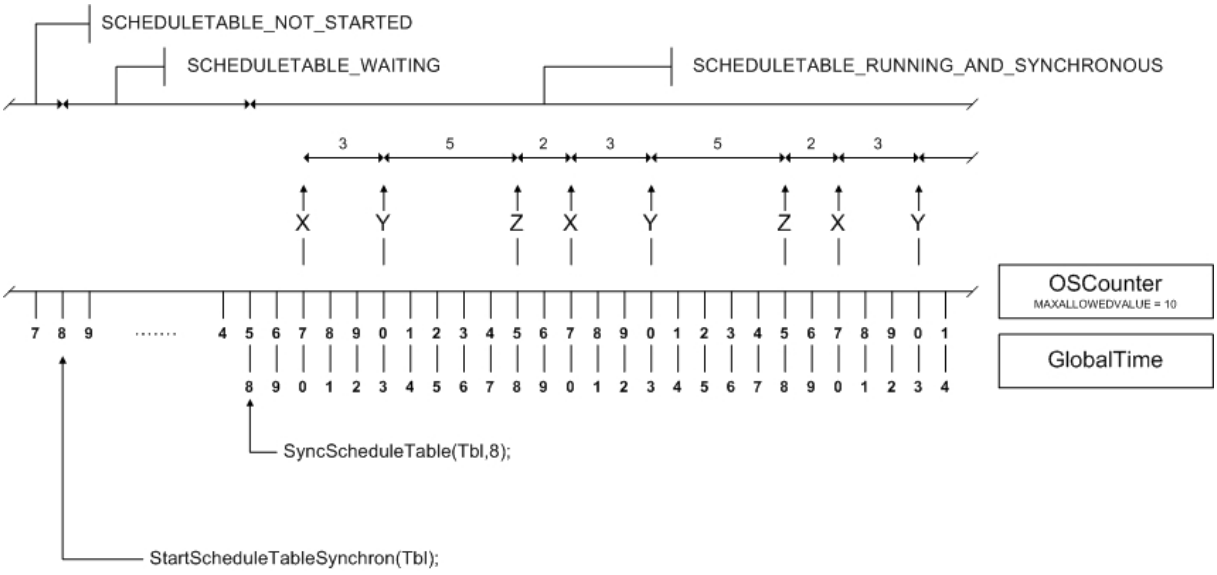

Fig. 9: Synchronous start of a schedule table

Figure 7 shows the asynchronous start of a schedule table, either with `StartScheduleTableRel()` or `StartScheduleTableAbs()`. After starting at the proposed (local) time, the global time is supported via `SyncScheduleTable()` and the schedule table is hardly set to global time at the end of the schedule table (in this case by expanding the time between the first Z and second X by 4).

Figure 8 shows also the asynchronous start of a schedule table, either with `StartScheduleTableRel()` or `StartScheduleTableAbs()`. After the start at the (local) time, the global time is again supported via `SyncScheduleTable()` and the OS smoothly adjust the expiry points of the table to global time. In this case it is

done by expanding the following expiry points by 2 (MAX_CORRECTION_ASYNC=2). The schedule table becomes synchron at the end of the first period.

Figure 9 shows the synchronous start of a schedule table. After the StartScheduleTableSynchron() call the table waits for the global time. After the time is set, the schedule table starts executing expiry points when global time is equal 0.

7.3.2 Requirements

OS013: The Operating System shall provide the ability to synchronize the processing of a schedule table to a global time base.

OS199: The Operating System shall provide services to be called by the user to provide the Operating System with the current global time (besides valid global time values, an input can be that the global time is not available, e.g., due to a temporary loss).

OS206: If SYNC_STRATEGY=SMOOTH AND a new global time is provided, the Operating System shall adjust each span of time between adjacent expiry points by either increasing or decreasing, whichever requires less adjustment steps.

OS200: SYNC_STRATEGY=SMOOTH AND synchronizing a schedule table to global time is done, the Operating System shall limit the adjustment of the span of time between adjacent expiry points to a statically configured range (this is captured by: MAX_CORRECTION_SYNC (OS310), if the schedule table is synchronous or MAX_CORRECTION_ASYNC, if the schedule table is asynchronous).

OS352: If SYNC_STRATEGY=HARD AND a new global time is provided, the Operating System shall adjust the time at the end of the schedule table to the new global time.

OS201:If a service to start a schedule table synchronously is called and the global time has not yet been provided, the Operating System shall start that schedule table synchronously after the global time has been provided.

OS227: The Operating System shall extend the service from OS359 to query the state of a schedule table with respect to synchronization. (criterion: was the deviation of the processing of the schedule table from global time, at the time it was last provided, smaller or equal/greater than the PRECISION threshold in OS310).

7.4 Stack Monitoring Facilities

7.4.1 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 OS 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.

7.4.2 Requirements

OS067: If a task/Category 2 ISR exceeds its specified stack usage, the Operating System shall be capable of detecting it.

OS068: If a stack fault is detected by stack monitoring AND the configured scalability class is 1 or 2, the Operating System shall call the ShutdownOS() service with the status E_OS_STACKFAULT.

OS396: If a stack fault is detected by stack monitoring AND the configured scalability class is 3 or 4, the Operating System shall call the ProtectionHook() with the status E_OS_STACKFAULT.

7.5 OS-Application

7.5.1 Background & Rationale

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

The OS 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, Resources, Counters, Alarms, Message and Schedule tables must belong to an OS-Application. All objects which belong to the same OS-Application have access to each other. Access from other OS-Applications may be granted during configuration. Access means to allow to use these objects within 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 OS API, and need not have their timing behaviour enforced at runtime. They are allowed to run in privileged mode when supported by the processor.

(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 OS 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 OS 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.

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.

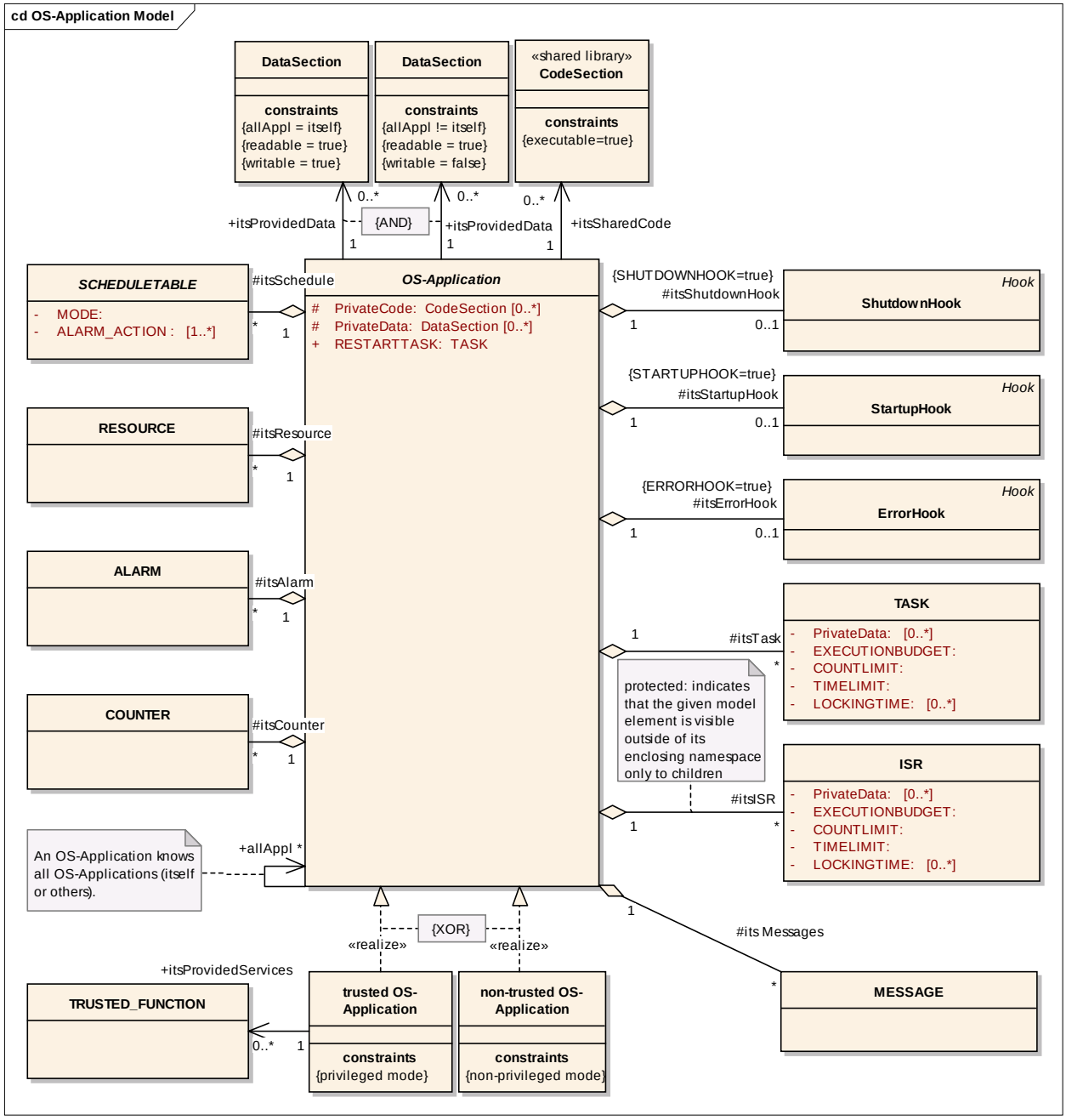

Fig. 10: UML-model of OS-Application

7.5.2 Requirements

OS016: The Operating System shall provide a service to determine the currently running OS-Application (a unique identifier shall be allocated to each application).

OS017: The Operating System shall provide a service to determine to which OS-Application a given Task, ISR, Resource, Counter, Alarm or Schedule Table belongs.

OS256: The Operating System shall provide a service to determine which OS-Applications are allowed to use the IDs of a Task, ISR, Resource, Counter, Alarm or Schedule Table in API calls.

OS258: The Operating System shall provide a service 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 behavior is equal to killing the calling OS-Application)

7.6 Protection Facilities

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

1. 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 shall be avoided. If Category 1 interrupts are used they shall belong to a trusted OS-Application.

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

7.6.1 Memory Protection

7.6.1.1 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 that OS-Application and there is therefore no need to share stack data between OS-Applications. The stacks of Task(s)/Category 2 ISR(s) belonging to different OS-Applications need to be protected.
The smallest unit managed by the OS is the stack of Task(s)/Category 2 ISR(s). Memory protection for the stack for each Task/Category 2 ISR within the same OS-Application is sometimes useful, mainly for two reasons:

(1) Provide a better (more immediate) detection of a stack overflow for the Task/Category 2 ISR compared to stack monitoring.
(2) Provide protection between the constituent parts of an OS-Application (e.g. to satisfy some safety constraints).

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

7.6.1.2 Requirements

Data Sections and Stack

OS198: The Operating System shall prevent write access to its own data sections and its own stack from other non-trusted OS-Applications.

Private data of an OS-Application

OS026: The Operating System may prevent read access to an OS-Application's data section attempted by other non-trusted OS-Applications.

OS086: The Operating System shall permit an OS-Application read and write access to that OS-Application's own private data sections.

OS207: The Operating System shall prevent write access to the OS-Application's private data sections from other non-trusted OS-Applications.

Private Stack of Task/ISR

OS196: The Operating System shall permit a Task/Category 2 ISR read and write access to that Task's/Category 2 ISR's own private stack.

OS208: The Operating System should 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.

OS355: The Operating System 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

OS087: The Operating System shall permit a task/Category 2 ISR read and write access to that task's/Category 2 ISR's own private data sections .

OS195: The Operating System should 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.

OS356: The Operating System 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

OS027: The Operating System may provide an OS-Application the ability to protect its code sections against executing by non-trusted OS-Applications.

OS081: The Operating System shall provide the ability to provide shared library code in sections that are executable by all OS-Applications.

Peripherals

OS209: The Operating System shall permit trusted OS-Applications read and write access to peripherals.

OS083: The Operating System 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

OS044: If a memory access violation is detected, the Operating System shall call the Protection Hook with status code `E_OS_PROTECTION_MEMORY`

7.6.2 Timing Protection

7.6.2.1 Background & Rationale

A timing fault in a real-time system means that a deadline will be missed at runtime. Meeting a deadline is a function of three items:

(1) the task/ISR's execution time performed between activation and before reaching the deadline

(2) the interference it suffers from higher priority tasks/ISRs running in preference

(3) the blocking it suffers from lower priority tasks/ISRs locking shared resources or disabling interrupts

Deadlines will be missed at run-time if any combination of these factors is longer than specified in any offline analysis that is performed to show that all deadlines can be met 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 with the example shown in the figure below. In this example, deadlines are set on Tasks A and B at 8ms and 14ms relative to the start respectively. This is shown according to the timeline at the top of the figure.

The timeline in the lower part of the figure shows that Task A executes incorrectly by running for too long, however it meets its deadline. The result of the incorrect execution of Task A is that Task B misses its deadline, even though it has behaved correctly.

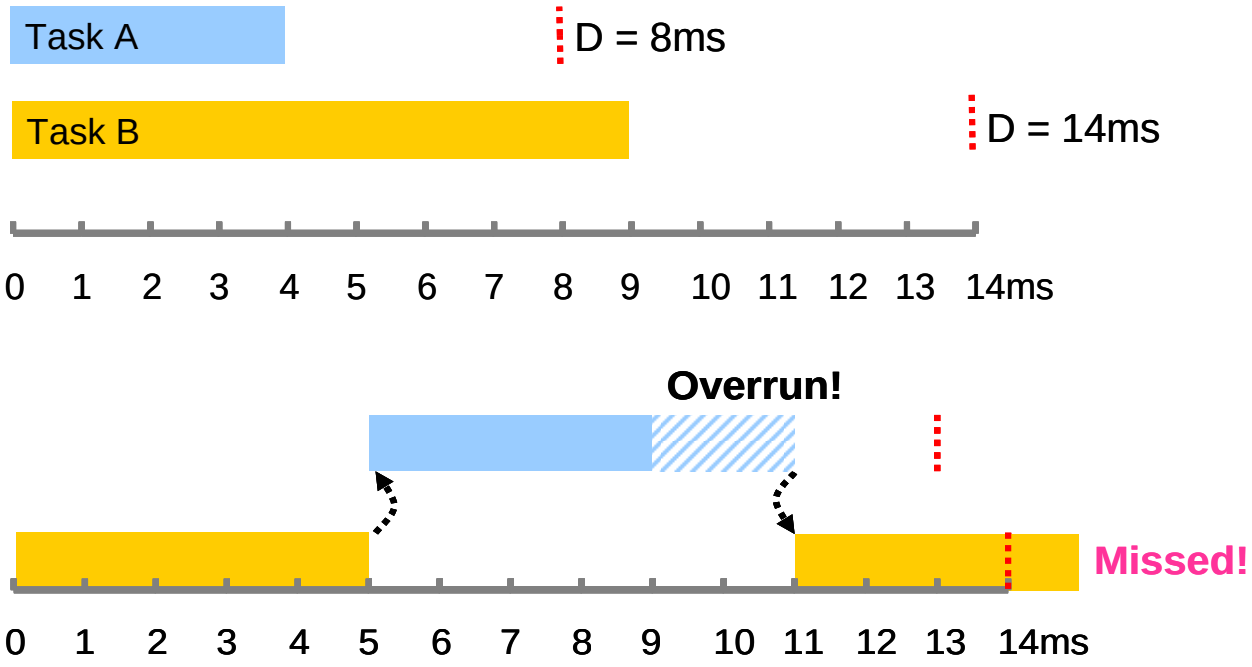

Fig. 11: Insufficiency of Deadline Monitoring

For safe and accurate timing protection it is necessary to enforce limits on the factors that determine whether or not Tasks/ISRs meet their respective deadlines.

To allow the combination of tasks/ISRs with specified deadlines to be executed in the same system with tasks without explicite deadlines it is necessary to define the enforced time limits in the following way:

(1) the execution time of each Category 2 ISR;

(2) the execution time of each Task in a specified timeframe. This is called the execution time budget for the task

(3) the execution time of each Task/Category 2 ISR while holding shared resources / disabling interrupts; and

(4) the arrival rate of each Category 2 ISR.

Execution time enforcement bounds the execution time of ISRs, resource locks and interrupt disabled sections at runtime to a statically configured value. For tasks the execution time in a specified time frame is bound. This prevents timing errors from (1), (2) and (3).

Arrival rate enforcement bounds the number of times that an ISR can execute in a given timeframe to statically configured limits. This prevents timing errors from (4).

Arrival rate enforcement protects 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) and provides the type of protection given by the OSEKtime Interrupt re-enable schedule event [8].

Note 1: The timing protection guarantees each task the time needed to finish before a deadline is reached. If, however, the application splits the processing into several parts (e.g. by state machines controlled by receiving events or by allowing multiple activations) it is only guaranteed that other tasks hold their specified deadlines. Splitting the processing in several parts can mean that parts are executed contiguous, or with a task switch, but inside one timeframe or in different timeframes. For an extended task, for example, a calculation or offline analysis of the execution time is possible but does not affect the Autosar OS timing protection scheme.

Note 2: In the case of a trusted OS-Application 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. Timing protection only applies to Tasks or Catgeory 2 ISRs, never to Catgeory 1 ISRs.

7.6.2.2 Requirements

OS028: In a non-trusted OS-Application, the Operating System shall apply timing protection to every task/Category 2 ISR of this non-trusted OS-Application.

OS089: In a trusted OS-Application, the Operating System shall be able to apply timing protection to task/Category 2 ISRs of this OS-Application.

OS397: If no OS-Application is configured, the Operating System shall be able to apply timing protection to task/Category 2 ISRs.

Timing Fault: ISRs

OS048: The Operating System shall limit the number of interrupt occurrences within a configured timeframe (if necessary disable interrupt source and re-enable the interrupts at the start of the next timeframe only if interrupt sources are not disabled by a specific system service or by reset). This requirement is related to (OS337).

OS337: The Operating System shall provide a service to disable the specified interrupt source.

OS338: The Operating System shall provide a service to request to enable the specified interrupt source. This requirement is related to (OS048)

OS210: When a Category 2 ISR reaches its execution time, the Operating System shall call the Protection Hook with `E_OS_PROTECTION_TIME`.

Timing Fault: resource locking and interrupt disabling

OS033: If a task/Category 2 ISR holds an OSEK Resource and exceeds the Resource Lock Time, the Operating System shall call the Protection Hook with `E_OS_PROTECTION_LOCKED`.

OS037: If a task/Category 2 ISR disables interrupts (via `Suspend/Disable|All/OS|Interrupts`) and exceeds the configured Interrupt Lock Time, the Operating System shall call the Protection Hook with `E_OS_PROTECTION_LOCKED`

Timing Fault: Tasks

OS064: If a task's execution time budget limit is reached the Operating System shall call the ProtectionHook with `E_OS_PROTECTION_TIME`.

7.6.2.3 Implementation Notes

Execution time enforcement requires a timing enforcement interrupt. This will typically be provided by the OS and should be the highest priority interrupt in the OS configuration (to prevent the execution time interrupt suffering interference from higher priority interrupts at runtime).

7.6.3 Service Protection

Background & Rationale

As OS-Applications may interact with the OS through services, it is essential that the service calls will not corrupt the OS 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 OS 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.

7.6.3.1 Invalid Object Parameter or Out of Range Value

7.6.3.1.1 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 `mincycle`).

7.6.3.1.2 Requirements

OS051: If an invalid address (address does not belong to the address space of this OS-Application) is passed to an OS service, the Operating System shall return the status code `E_OS_ILLEGAL_ADDRESS`.

7.6.3.2 Service Calls Made from Wrong Context

7.6.3.2.1 Background & Rationale

The current OSEK OS defines the valid calling context for service calls ([7], Fig. 12-1), 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	✓		✓							
TerminateTask	✓		C							
ChainTask	✓		C							
Schedule	✓		C							
GetTaskID	✓		✓	✓	✓	✓				✓
GetTaskState	✓		✓	✓	✓	✓				
DisableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EnableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SuspendAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ResumeAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SuspendOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ResumeOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GetResource	✓		✓							
ReleaseResource	✓		✓							
SetEvent	✓		✓							
ClearEvent	✓		C							
GetEvent	✓		✓	✓	✓	✓				
WaitEvent	✓		C							
GetAlarmBase	✓		✓	✓	✓	✓				
GetAlarm	✓		✓	✓	✓	✓				
SetRelAlarm	✓		✓							
SetAbsAlarm	✓		✓							
CancelAlarm	✓		✓							
GetActiveApplicationMode	✓		✓	✓	✓	✓	✓	✓		
StartOS										
ShutdownOS	✓		✓	✓			✓			
GetApplicationID	✓		✓	✓	✓	✓	✓	✓		✓
GetISRID	✓		✓	✓						✓
CallTrustedFunction	✓		✓							
CheckISRMemoryAccess	✓		✓	✓						✓
CheckTaskMemoryAccess	✓		✓	✓						✓
CheckObjectAccess	✓		✓	✓						✓
CheckObjectOwnership	✓		✓	✓						✓
StartScheduleTableRel	✓		✓							
StartScheduleTableAbs	✓		✓							
StopScheduleTable	✓		✓							
NextScheduleTable	✓		✓							
StartScheduleTableSynchron	✓		✓							

Service	Task	Cat1 ISR	Cat2 ISR	Error Hook	PreTask Hook	PostTask Hook	Startup Hook	Shutdown Hook	Alarm Callback	Protection Hook
SyncScheduleTable	✓		✓							
GetScheduleTableStatus	✓		✓							
SetScheduleTableAsync	✓		✓							
IncrementCounter	✓		✓							
GetCounterValue	✓		✓							
GetElapsedCounterValue	✓		✓							
TerminateApplication	✓		✓	✓[1]						
DisableInterruptSource	✓		✓							
EnableInterruptSource	✓		✓							

Tab. 7: Allowed Calling Context for OS Service Calls

C indicates that validity is only “Checked in Extended status by E_OS_CALLEVEL”.

7.6.3.2.2 Requirements

OS088: If an OS-Application makes a service call from the wrong context AND is currently not inside a Category 1 ISR the Operating System shall not perform the requested action (the service call shall have no affect), and return E_OS_CALLEVEL or the “invalid value” of the service unless it is inside a Category 1 ISR.

7.6.3.3 Services with Undefined Behaviour

7.6.3.3.1 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 OS to be corrupted through its own service calls. The implementation of service protection for the OS 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] Only in application specific ErrorHooks.

7.6.3.3.2 Requirements

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

OS052: If a task returns from its entry function without making a `TerminateTask()` or `ChainTask()` call, the Operating System shall terminate the task (and call the PostTaskHook if configured).

OS069: If a task returns from its entry function without making a `TerminateTask()` or `ChainTask()` call AND the ErrorHook is configured, the Operating System 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.

OS070: If a task returns from the entry function without making a `TerminateTask()` or `ChainTask()` call and still holds OSEK Resources, the Operating System shall release them.

OS239: If a task returns from the entry function without making a `TerminateTask()` or `ChainTask()` call and interrupts are still disabled, the Operating System shall enable them.

Category 2 ISR ends with locked interrupts or allocated resources

OS368: If a Category 2 ISR calls `DisableAllInterupts()` / `SuspendAllInterrupts()` / `SuspendOSInterrupts()` and ends (returns) without calling the corresponding `EnableAllInterrupts()` / `ResumeAllInterrupts()` / `ResumeOSInterrupts()`, the Operating System shall perform the missing service and shall call the ErrorHook (if configured) with the status `E_OS_MISSINGEND`.

OS369: If a Category 2 ISR calls `GetResource()` and ends (returns) without calling the corresponding `ReleaseResource()`, the Operating System shall perform the `ReleaseResource()` call and shall call the ErrorHook (if configured) with the status `E_OS_MISSINGEND`.

PostTaskHook called during ShutdownOS()

OS071: If the PostTaskHook is configured, the Operating System shall not call the hook if `ShutdownOS()` is called.

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

OS092: If `EnableAllInterrupts()` / `ResumeAllInterrupts()` / `ResumeOSInterrupts()` are called and no corresponding `DisableAllInterupts()` / `SuspendAllInterrupts()` / `SuspendOSInterrupts()` was done before, the Operating System shall not perform this OS service.

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

OS093: If interrupts are disabled and any OS services, excluding the interrupt services, are called outside of hook routines, then the Operating System shall return `E_OS_DISABLEDINT`

7.6.3.4 Service Restrictions for Non-Trusted OS-Applications

7.6.3.4.1 Background & Rationale

The OS service calls available are restricted according to the calling context (see Section 7.6.3.2). 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
Fig. 12.

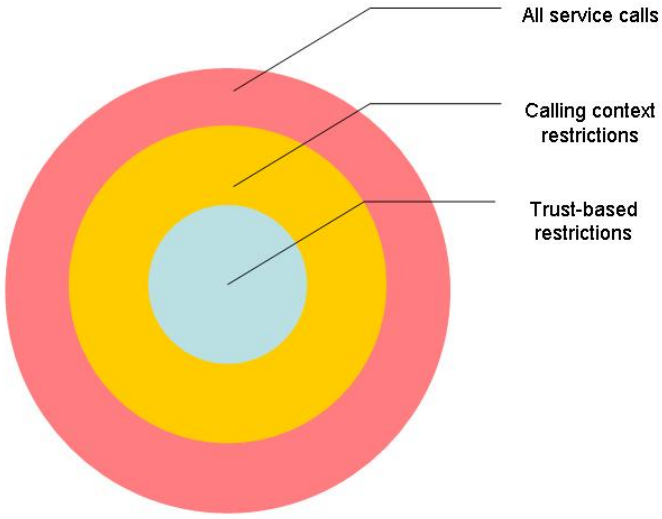

Fig. 12: API Restrictions

There are two defined integrity levels:

1. Trusted
2. Non-Trusted

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

7.6.3.4.2 Requirements

OS054: The Operating System shall ignore calls to `ShutdownOS()` from non-trusted OS-Applications.

7.6.3.5 Service Calls on Objects in Different OS-Applications

7.6.3.5.1 Background

Section 7.6.3.1 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 for two reasons:

(1) the object does not exist (the current OSEK OS meaning)
(2) the caller does not have valid permissions for the object (a new meaning for multi-OS-Application systems)

7.6.3.5.2 Requirements

OS056: If an OS-object identifier is the parameter of a system service, and no sufficient access rights have been assigned at configuration time to the calling Task/Category 2 ISR, the system service shall return `E_OS_ID`.

7.6.4 Protecting the Hardware used by the OS

7.6.4.1 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 OS in privileged mode and Tasks/ISRs in non-privileged mode protects the registers fundamental to OS operation from inadvertent corruption by the objects executing in non-privileged mode. The OS services will need to execute in privileged mode as they will need to modify the registers that are protected outside this mode.

The OS may 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.

7.6.4.2 Requirements

OS058: If supported by hardware, the Operating System shall execute non-trusted OS-Applications in non-privileged mode.

OS096: As far as supported by hardware, the Operating System shall not allow non-trusted OS-Applications to access control registers managed by the Operating

System and the Operating System shall restrict access for trusted OS-Applications to registers exclusively managed by the trusted OS-Applications.

OS245: If an instruction exception occurs (e.g. division by zero) the operating system shall call the protection hook with `E_OS_PROTECTION_EXCEPTION`.

7.6.4.3 Implementation Notes

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

Interrupt handling: Where the MCU is moded (as discussed in this section) ISRs will require the OS 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.

7.6.5 Providing »Trusted Functions«

7.6.5.1 Background & Rationale

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

(1) Each trusted OS-Application may export services which are callable from other OS-Applications.
(2) During configuration these trusted services must be configured to be called from a non-trusted OS-Application.
(3) The call from the non-trusted OS-Application to the trusted service is using a mechanism (e.g. trap/software interrupt) provided by the OS. The service is passed as an identifier that is used to determine, in the trusted environment, if the service can be called.
(4) The OS 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 system does not support »non-trusted services«.

7.6.5.2 Requirements

OS097: The Operating System shall provide a mechanism to call a trusted function from a (trusted or non-trusted) OS-Application.

OS100: If a called trusted function is not configured the Operating System shall call the ErrorHook with `E_OS_SERVICEID`.

OS099: The Operating System shall offer OS-Applications a service 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.

7.7 Protection Errors

7.7.1 Background & Rationale

The OS can detect protection errors based on statically configured information on what the constituent parts of an OS-Application can do at runtime. See Section 7.6.

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 situation where a protection error can occur is described in the glossary. If a protection error occurs before the operating system 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 OS calls a user provided Protection Hook for the notification of protection errors at runtime. The Protection Hook runs in the context of the OS and must therefore be trusted code.

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

- kill the faulty Task/Category 2 ISR
- kill the faulty OS-Application (with or without restart of the OS-Application)
- shutdown the OS.

Requirements OS243 and OS244 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 to be defined.

Note that killing of interrupts is handled differently in “kill the faulty ISR” and “kill the OS-Application”. If a faulty ISR is killed, only the instance of this ISR is killed. If the OS-Application is killed, the interrupt source is additionally disabled.

7.7.2 Requirements

OS211: The Operating System shall call the Protection Hook with the same permissions as the Operating System.

OS106: The Operating System shall perform one of the following reactions depending on the return value of the Protection Hook:

- Kill the faulty Task/Category 2 ISR OR
- Kill the faulty OS-Application OR

- Kill the faulty OS-Application and restart the OS-Application. OR
- Call `ShutdownOS()`.

OS107: If no Protection Hook is configured and a protection error occurs, the Operating System shall call `ShutdownOS()`.

OS243: If the reaction is to kill the Task/Category 2 ISR and no Task or ISR can be associated with the error, the running OS-Application is killed.

OS244: If the reaction is to kill the faulty OS-Application and no OS-Application can be assigned, `ShutdownOS()`is called.

OS108: If the Operating System kills a task, it terminates the task (no PostTaskHook for the task will be called), 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.

OS109: If the Operating System kills 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.

OS110: If the Operating System kills an OS-Application, it:
- kills all task/ISRs of the OS-Application AND
- cancels all alarms of the OS-Application AND
- stops schedule tables of the OS-Application AND
- disables interrupt sources of Category 2 ISRs belonging to the OS-Application

OS315: If the Operating System kills an OS-Application (or Task/Category 2 ISR) the associated local (internal) messages remain in their current state.

OS111: When the Operating System restarts an OS-Application it activates the configured `RESTARTTASK`.

7.8 System Scalability

7.8.1 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 chapter	Scalability Class 1	Scalability Class 2	Scalability Class 3	Scalability Class 4	Hardware requirements
OSEK OS (all conformance classes)	7.1	✓	✓	✓	✓	
Counter Interface	8.4.12	✓	✓	✓	✓	
SWFRT Interface	8.4.13, 8.4.14	✓	✓	✓	✓	Optional feature to support GPT driver
Schedule Tables	7.2	✓	✓	✓	✓	
Stack Monitoring	7.4	✓	✓	✓	✓	
ProtectionHook	7.7		✓	✓	✓	
Timing Protection	7.6.2		✓		✓	Timer(s) with high priority interrupt
Global Time /Synchronization Support	7.3		✓		✓	Global time source
Memory Protection	7.6.1, 7.6.4			✓	✓	MPU
OS-Applications	7.5, 7.9			✓	✓	
Service Protection	7.6.3			✓	✓	
CallTrustedFunction	7.6.5			✓	✓	(Non-)privileged Modes

Tab. 8: 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. 9: Minimum requirements of scalability classes

7.8.2 Requirements

OS240: If an implementation of a lower scalability class supports features of higher classes then the interfaces for the features must comply with this specification.

OS241: The operating system shall support the features according to the configured scalability class. (See Tab. 8)

OS327: The operating system shall always use extended mode in scalability class 3 and 4.

7.9 Hook Functions

7.9.1 Background & Rationale

Hook routines as defined in OSEK OS run at the level of the OS 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.

7.9.2 Requirements

StartupHook

OS060: If an application-specific startup hook is configured for an OS-Application `<App>`, the Operating System shall call `StartupHook_<App>` on startup of the OS.

OS226: The Operating System shall execute an application-specific startup hook with the access rights of the associated OS-Application.

OS236: If both a system-specific and one (or more) application specific startup hook(s) are configured, the Operating System shall call the system-specific startup hook before the application-specific startup hook(s).

ShutdownHook

OS112: If an application-specific shutdown hook is configured for an OS-Application `<App>`, the Operating System shall call `ShutdownHook_<App>` on shutdown of the OS.

OS225: The Operating System shall execute an application-specific shutdown hook with the access rights of the associated OS-Application.

OS237: If both a system-specific and one (or more) application specific shutdown hook(s) are configured, the Operating System shall call the system-specific shutdown hook after the application-specific shutdown hook(s).

Error Hook

OS246: When an error occurs AND an application-specific error hook is configured for the faulty OS-Application `<App>`, the Operating System shall call that application-specific error hook `ErrorHook_<App>` after the system specific error hook is called (if configured).

OS085: The Operating System shall execute an application-specific error hook with the access rights of the associated OS-Application.

OS367: Operating System services which do not return a StatusType shall not raise the error hook(s).

7.10 Error classification

Instead of specifying two versions for production and development errors the AUTOSAR OS provides a finer granularity to adjust the error handling to specific needs, e.g. Scalability Classes, standard and extended status, switching on/off of hook routines.

Type or error	*Relevance*	*Related error code*	*Value*
Service can not be called.	Production	`E_OS_SERVICEID`	`Assigned by implementation`
An invalid address is given as a parameter to a service.	Production	`E_OS_ILLEGAL_ADDRESS`	`Assigned by implementation`
Tasks terminates without a `TerminateTask()` or `ChainTask()` call.	Production	`E_OS_MISSINGEND`	`Assigned by implementation`
A service of the OS is called inside an interrupt disable/enable pair.	Production	`E_OS_DISABLEDINT`	`Assigned by implementation`
A stack fault detected via stack monitoring by the OS	Production	`E_OS_STACKFAULT`	`Assigned by implementation`
A memory access violation occurred	Production	`E_OS_PROTECTION_MEMORY`	`Assigned by implementation`
A Task exceeds its execution time budget	Production	`E_OS_PROTECTION_TIME`	`Assigned by implementation`
A Category 2 ISR exceeds its execution time			
A Task/Category 2 ISR blocks for too long	Production	`E_OS_PROTECTION_LOCKED`	`Assigned by implementation`
A trap occurred	Production	`E_OS_PROTECTION_EXCEPTION`	`Assigned by implementation`

8 API specification

8.1 Constants

8.1.1 Error codes of type StatusType

See Section 7.10 and [7].

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.

8.3 Type definitions

8.3.1 ApplicationType (for OS-Applications)

Type:	Scalar
Description:	This data type identifies the OS-Application.
Constants of this Type:	`INVALID_OSAPPLICATION`

8.3.2 TrustedFunctionIndexType

Type:	Scalar
Description:	This data type identifies a trusted function.

8.3.3 TrustedFunctionParameterRefType

Type:	Pointer
Description:	This data type points to a structure which holds the arguments for a call to a trusted function.

8.3.4 AccessType

Type:	Integral
Description:	This type holds information how a specific memory region can be accessed.

8.3.5 ObjectAccessType

Type:	Scalar
Description:	This data type identifies if an OS-Application has access to an object.
Constants of this Type:	ACCESS NO_ACCESS

8.3.6 ObjectTypeType

Type:	Scalar
Description:	This data type identifies an object.
Constants of this Type:	OBJECT_TASK OBJECT_ISR OBJECT_ALARM OBJECT_RESOURCE OBJECT_COUNTER OBJECT_SCHEDULETABLE OBJECT_MESSAGE

8.3.7 MemoryStartAddressType

Type:	Pointer
Description:	This data type is a pointer which is able to point to any location in the MCU address space.

8.3.8 MemorySizeType

Type:	Scalar
Description:	This data type holds the size of a memory region.

8.3.9 ISRType

Type:	Scalar
Description:	This data type identifies an interrupt service routine (ISR).
Constants of this Type:	INVALID_ISR

8.3.10 ScheduleTableType

Type:	Scalar
Description:	This data type identifies a schedule table.

8.3.11 ScheduleTableStatusType

Type:	Scalar
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_NOT_STARTED) 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 hard synchronization and waits for the global time. (SCHEDULETABLE_WAITING) o The schedule table is started and runs, but is currently not synchronous to a global time source (SCHEDULETABLE_RUNNING) o The schedule table is started, runs and currently synchronous to a global time source (SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS)
Constants of this Type:	SCHEDULETABLE_NOT_STARTED SCHEDULETABLE_NEXT SCHEDULETABLE_WAITING SCHEDULETABLE_RUNNING SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS

8.3.12 ScheduleTableStatusRefType

Type:	Pointer
Description:	This data type points to a variable of the data type ScheduleTableStatusType.

8.3.13 CounterType

Type:	Scalar
Description:	This data type identifies a counter.

8.3.14 GlobalTimeTickType

Type:	Scalar
Description:	This data type identifies a value of a global time source.

8.3.15 ProtectionReturnType

Type:	Scalar
Description:	This data type identifies a value which controls further actions of the OS on return from the protection hook.
Constants of this Type:	PRO_KILLTASKISR PRO_KILLAPPL PRO_KILLAPPL_RESTART PRO_SHUTDOWN

8.3.16 RestartType

Type:	Scalar
Description:	This data type defines the use of a Restart Task after terminating an OS-Application.
Constants of this Type:	`RESTART` `NO_RESTART`

8.3.17 UnitType

Type:	Scalar
Description:	This data type defines the conversation units.

Constants of this Type:	`UNIT_NS`
	`UNIT_US`
	`UNIT_MS`
	`UNIT_SEC`

8.3.18 PhysicalTimeType

Type:	Scalar
Description:	This data type is used for values returned by the conversation macro `OS_TICKS2<Unit>_<Counter>()`.

8.4 Function definitions

The availability of the following services is defined in Tab. 8. The use of these services may be restricted depending on the context they are called from. See Tab. 7 for details.

8.4.1 GetApplicationID

Service name:	GetApplicationID
Syntax:	`ApplicationType GetApplicationID (void)`
Service ID:	`OSServiceId_GetApplicationID`
Sync/Async:	Sync
Reentrancy:	Yes
Parameters (in):	None
Parameters (out):	None
Return value:	<Identifier of running OS-Application> or
	`INVALID_OSAPPLICATION`
Description:	OS261: `GetApplicationID()` shall return the application identifier to which the executing Task/ISR/hook belongs. OS262: If no OS-Application is running, `GetApplicationID()` shall return `INVALID_OSAPPLICATION`.
Caveats:	None
Configuration:	Available in Scalability Classes 3 and 4

8.4.2 GetISRID

Service name:	GetISRID
Syntax:	`ISRType GetISRID (void)`
Service ID:	`OSServiceId_GetISRID`
Sync/Async:	Sync
Reentrancy:	Yes
Parameters (in):	None
Parameters (out):	None
Return value:	<Identifier of running ISR> or
	`INVALID_ISR`
Description:	OS263: If called from interrupt level, `GetISRID()` shall return the identifier of the currently executed ISR. OS264: If its caller is not an active ISR, `GetISRID()` shall return `INVALID_ISR`.
Caveats:	None
Configuration:	Available in all Scalability Classes.

8.4.3 CallTrustedFunction

Service name:	CallTrustedFunction	

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

Service ID:	`OSServiceId_CallTrustedFunction`	
Sync/Async:	Depends on called function. If called function is synchronous then service is synchronous. May cause rescheduling.	
Reentrancy:	Yes	
Parameters (in):	`FunctionIndex`	Index of the function to be called.
	`FunctionParams` or `NULL`	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 (out):	None	
Return value:	`E_OK`	No Error
	`E_OS_SERVICEID`	No function defined for this index

Description:	OS265: If <FunctionIndex> is a defined function index, `CallTrustedFunction()` shall switch the processor into privileged mode AND shall call the function <FunctionIndex> out of a list of implementation specific trusted functions AND shall return `E_OK` after completion. OS312: 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). OS266: When the function <FunctionIndex> is called, it shall get the same permissions (access rights) than the associated trusted OS-Application. OS364: If the trusted function is called from a Category 2 ISR context it 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 7.6.3.2). OS365: If the trusted function is called from a task context it shall continue to run on the same priority and be allowed to call all system services defined for tasks (see table in chapter 7.6.3.2). OS292: If the function index <FunctionIndex> is undefined, `CallTrustedFunction()` shall return `E_OS_SERVICEID`.
Caveats:	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.
Configuration:	Available in Scalability Classes 3 and 4

8.4.4 CheckISRMemoryAccess

Service name:	CheckISRMemoryAccess	
Syntax:	`AccessType CheckISRMemoryAccess ( ISRType ISRID, MemoryStartAddressType Address, MemorySizeType Size )`	
Service ID:	`OSServiceId_CheckISRMemoryAccess`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ISRID`	ISR reference
	`Address`	Start of memory area
	`Size`	Size of memory area
Parameters (out):	None	
Return value:	Value which contains the access rights to the memory area.	
Description:	OS267: If the ISR reference <ISRID> is valid, `CheckISRMemoryAccess()` shall return the access rights of the ISR on the specified memory area. OS313: If an access right is not valid for the whole specified memory area `CheckISRMemoryAccess()` shall yield no access right. OS268: If the ISR reference <ISRID> is not valid, `CheckISRMemoryAccess()` shall yield no access rights.	
Caveats:	None	
Configuration:	Available in Scalability Classes 3 and 4	

8.4.5 CheckTaskMemoryAccess

Service name:	CheckTaskMemoryAccess	
Syntax:	`AccessType CheckTaskMemoryAccess ( TaskType TaskID, MemoryStartAddressType Address, MemorySizeType Size )`	
Service ID:	`OSServiceId_CheckTaskMemoryAccess`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`TaskID`	Task reference
	`Address`	Start of memory area
	`Size`	Size of memory area
Parameters (out):	None	
Return value:	Value which contains the access rights to the memory area.	
Description:	OS269: If the Task reference <TaskID> is valid, `CheckTaskMemoryAccess()` shall return the access rights of the task on the specified memory area. OS314: If an access right is not valid for the whole specified memory area `CheckTaskMemoryAccess()` shall yield no access right. OS270: If the Task reference <TaskID> is not valid, `CheckTaskMemoryAccess()` shall yield no access rights.	
Caveats:	None	
Configuration:	Available in Scalability Classes 3 and 4	

8.4.6 CheckObjectAccess

Service name:	CheckObjectAccess	
Syntax:	`ObjectAccessType CheckObjectAccess ( ApplicationType ApplID, ObjectTypeType ObjectType, … )`	
Service ID:	`OSServiceId_CheckObjectAccess`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ApplID`	OS-Application identifier
	`ObjectType`	Type of the following parameter
	…	The object to be examined
Parameters (out):	None	
Return value:	`ACCESS`	if the `ApplID` has access to the object
	`NO_ACCESS`	otherwise
Description:	OS271: If the OS-Application <ApplID> has access to the queried object, `CheckObjectAccess()` shall return `ACCESS`. OS272: If the OS-Application <ApplID> has no access to the queried object, `CheckObjectAccess()` shall return `NO_ACCESS`. OS318: If the object to be examined is the `RES_SCHEDULER` `CheckObjectAccess()` shall always return `ACCESS`.	

Caveats:	None
Configuration:	Available in Scalability Classes 3 and 4

8.4.7 CheckObjectOwnership

Service name:	CheckObjectOwnership	
Syntax:	`ApplicationType CheckObjectOwnership ( ObjectTypeType ObjectType, … )`	
Service ID:	`OSServiceId_CheckObjectOwnership`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ObjectType`	Type of the following parameter
	…	The object to be examined
Parameters (out):	None	
Return value:	<OS-Application>	The service returns the OS-Application to which the object ObjectType belongs or
	`INVALID_OSAPPLICATION`	If the object does not exists the service returns:
Description:	OS273: If the specified object ObjectType exists, `CheckObjectOwnership()` shall return the identifier of the OS-Application to which the object belongs. OS274: If the specified object ObjectType does not exist, `CheckObjectOwnership()` shall return `INVALID_OSAPPLICATION`. OS319: If the object to be examined is the `RES_SCHEDULER` `CheckObjectOwnership()` shall always return `INVALID_OSAPPLICATION`.	
Caveats:	None	
Configuration:	Available in Scalability Classes 3 and 4	

8.4.8 StartScheduleTableRel

Service name:	StartScheduleTableRel	
Syntax:	`StatusType StartScheduleTableRel ( ScheduleTableType ScheduleTableID, TickType Offset )`	
Service ID:	`OSServiceId_StartScheduleTableRel`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID`	Schedule table to be started
	`Offset`	Relative tick value between now and the first alarm expiry
Parameters (out):	None	
Return value:	`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 `MAXALLOWEDVALUE` or is equal to 0.

	E_OS_STATE	Schedule table was already started.
Description:	OS275: If the schedule table <ScheduleTableID> is not valid, StartScheduleTable() shall return E_OS_ID. OS332: If <Offset> is zero StartScheduleTable() shall return E_OS_VALUE. OS276: If the offset <Offset> is greater than MAXALLOWEDVALUE, StartScheduleTable() shall return E_OS_VALUE. OS277: If the schedule table <ScheduleTableID> was already started or is currently in state SCHEDULETABLE_NEXT, StartScheduleTable() shall return E_OS_STATE. OS278: If its input parameters are valid, StartScheduleTable() shall start the processing of a schedule table <ScheduleTableID> at its first expiry point after offset <Offset> ticks have elapsed.	
Caveats:	None	
Configuration:	Available in all Scalability Classes.	

8.4.9 StartScheduleTableAbs

Service name:	StartScheduleTableAbs	
Syntax:	StatusType StartScheduleTableAbs (ScheduleTableType ScheduleTableID, TickType Tickvalue)	
Service ID:	OSServiceId_StartScheduleTableAbs	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	ScheduleTableID	Schedule table to be started
	Tickvalue	Absolute tick value of the first alarm expiry
Parameters (out):	None	
Return value:	E_OK	No Error
	E_OS_ID (only in EXTENDED status)	ScheduleTableID not valid.
	E_OS_VALUE (only in EXTENDED status)	Tickvalue is greater than MAXALLOWEDVALUE.
	E_OS_STATE	Schedule table was already started.
Description:	OS348: If the schedule table <ScheduleTableID> is not valid, StartScheduleTable() shall return E_OS_ID. OS349: If the <Tickvalue> is greater than MAXALLOWEDVALUE, StartScheduleTable() shall return E_OS_VALUE. OS350: If the schedule table <ScheduleTableID> was already started or is currently in state SCHEDULETABLE_NEXT, StartScheduleTable() shall return E_OS_STATE. OS351: If its input parameters are valid, StartScheduleTable() shall start the processing of schedule table <ScheduleTableID> at its first expiry point after the underlaying counter reaches <Tickvalue>.	

Caveats:	None
Configuration:	Available in all Scalability Classes.

8.4.10 StopScheduleTable

Service name:	StopScheduleTable	
Syntax:	`StatusType StopScheduleTable ( ScheduleTableType ScheduleTableID )`	
Service ID:	`OSServiceId_StopScheduleTable`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID`	Schedule table to be stopped
Parameters (out):	None	
Return value:	`E_OK`	No Error
	`E_OS_ID` (only in `EXTENDED` status)	`ScheduleTableID` not valid.
	`E_OS_NOFUNC`	Schedule table was already stopped
Description:	OS279: If the schedule table identifier <ScheduleTableID> is not valid, `StopScheduleTable()` shall return `E_OS_ID`. OS280: If the schedule table with identifier <ScheduleTableID> was not started, `StopScheduleTable()` shall return `E_OS_NOFUNC`. OS281: If its input parameters are valid, `StopScheduleTable()` shall stop the schedule table <ScheduleTableID> from processing any further expiry points and shall return `E_OK`.	
Caveats:	None	
Configuration:	Available in all Scalability Classes.	

8.4.11 NextScheduleTable

Service name:	NextScheduleTable	
Syntax:	`StatusType NextScheduleTable ( ScheduleTableType ScheduleTableID_current, ScheduleTableType ScheduleTableID_next )`	
Service ID:	`OSServiceId_NextScheduleTable`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID_current`	Schedule table
	`ScheduleTableID_next`	Schedule table that provides its series of expiry points
Parameters (out):	None	
Return value:	`E_OK`	No error
	`E_OS_ID` (only in `EXTENDED` status)	`ScheduleTableID_current` or `ScheduleTableID_next` not valid.
	`E_OS_NOFUNC`	`ScheduleTableID_current` not started.
	`E_OS_STATE` (only in `EXTENDED` status)	`ScheduleTableID_next` is started or next.
Description:	OS282: If the input parameter <ScheduleTableID_current> or <ScheduleTableID_next> is not valid, `NextScheduleTable()` shall return `E_OS_ID`. OS330: If schedule table <ScheduleTableID_next> is driven by different counter than schedule table <ScheduleTableID_current> than `NextScheduleTable()` shall return an error `E_OS_ID`. OS283: If the schedule table <ScheduleTableID_current> is not started OR is itself in state `SCHEDULETABLE_NEXT`, `NextScheduleTable()` shall return `E_OS_NOFUNC`. OS309: If the schedule table <ScheduleTableID_next> is already started or next then `NextScheduleTable()` shall return `E_OS_STATE`. OS284: If its input parameters are valid AND the schedule table <ScheduleTableID_current> is started OR waits for the global time, `NextScheduleTable()` shall start the processing of schedule table <ScheduleTableID_next> after <ScheduleTableID_current> reaches its period/length and shall return `E_OK`. OS324: If the input parameters are valid AND the <ScheduleTableID_current> is running AND `NextScheduleTable()` was previously successful called the new <ScheduleTableID_next> shall replace the previous next value. OS363: The synchronization strategy of the <ScheduleTableID_next> shall come into effect when the Operating System processes the first expiry point of <ScheduleTableID_next>.	
Caveats:	If the `<ScheduleTableID_current>` is stopped before `<ScheduleTableID_next>` is started, `<ScheduleTableID_next>` is not started.	
Configuration:	Available in all Scalability Classes.	

8.4.12 IncrementCounter

Service name:	IncrementCounter	
Syntax:	`StatusType IncrementCounter ( CounterType CounterID )`	
Service ID:	`OSServiceId_IncrementCounter`	
Sync/Async:	Sync, may cause rescheduling.	
Reentrancy:	Yes	
Parameters (in):	`CounterID`	The Counter to be incremented
Parameters (out):	None	
Return value:	`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:	OS285: If the input parameter <CounterID> is not valid OR the counter is a hardware counter, `IncrementCounter()` shall return `E_OS_ID`. OS286: If its input parameter 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`. OS321: If an error happens during the execution of an alarm action, e.g. `E_OS_LIMIT` caused by a task activation, the error hook(s) are called, but the `IncrementCounter()` service itself will return `E_OK`.	
Caveats:	If called from a task, rescheduling may take place.	
Configuration:	Available in all Scalability Classes.	

8.4.13 GetCounterValue

Service name:	GetCounterValue	
Syntax:	`StatusType GetCounterValue ( CounterType CounterID, TickRefType Value )`	
Service ID:	`OSServiceId_GetCounterValue`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`CounterID`	The Counter which tick value should be read
Parameters (out):	`Value`	Contains the current tick value of the counter
Return value:	`E_OK`	No errors
	`E_OS_ID` (only in `EXTENDED` status)	The <CounterID> was not valid.
Description:	OS376: If the input parameter <CounterID> is not valid, the service should return `E_OS_ID`. OS377: If its input parameter is valid, `GetCounterValue()` shall return the current tick value of the counter via `Value` and return `E_OK`.	
Caveats:	Note that for counters of type `HARDWARE` the real timer value is returned, whereas for counters of type `SOFTWARE` the current “software” tick value is returned.	

Configuration:	Available in all Scalability Classes. (Optional feature)

8.4.14 GetElapsedCounterValue

Service name:	GetElapsedCounterValue	
Syntax:	`StatusType GetElapsedCounterValue ( CounterType CounterID, TickType PreviousValue, TickRefType Value )`	
Service ID:	`OSServiceId_GetElapsedCounterValue`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`CounterID`	The Counter to be incremented
	`PreviousValue`	The previously read value
Parameters (out):	`Value`	The difference to the previous read value
Return value:	`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 PreviousValue was not valid
Description:	OS381: If the input parameter <CounterID> is not valid the service will return E_OS_ID. OS391: If the <PreviousValue> is larger than the max allowed value of the counter, the service will return E_OS_VALUE. OS382: If its input parameter are valid, `GetElapsedCounterValue ()` shall return the number of elapsed ticks since the given <PreviousTick> value via <Value> and shall return E_OK..	
Caveats:	If the timer already passed the <PreviousTick> value a second (or multiple) time, the result returned is wrong. The reason is that the service can not detect such an relative overflow.	
Configuration:	Available in all Scalability Classes. (Optional feature)	

8.4.15 StartScheduleTableSynchron

Service name:	StartScheduleTableSynchron	
Syntax:	`StatusType StartScheduleTableSynchron ( ScheduleTableType ScheduleTableID, GlobalTimeTickType GlobalTime )`	
Service ID:	`OSServiceId_StartScheduleTableSynchron`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID`	Schedule table to be started

	`GlobalTime`	Global time value when the schedule table starts
Parameters (out):	None	
Return value:	`E_OK`	No Error
	`E_OS_ID` (only in `EXTENDED` status)	`ScheduleTableID` not valid.
	`E_OS_STATE`	Schedule table was already started.
	`E_OS_VALUE` (only in `EXTENDED` status)	`GlobalTime` is invalid
Description:	OS387: If the schedule table <ScheduleTableID> is not valid OR the schedule table <ScheduleTableID> can not be synchronized (`LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION=FALSE`) `StartScheduleTableSynchron()` shall return `E_OS_ID`. OS388: If the schedule table <ScheduleTableID> is not in the state `SCHEDULETABLE_NOT_STARTED`, the service shall return `E_OS_STATE`. OS395: If the GlobalTime is invalid, the service shall return `E_OS_VALUE`. OS389: If its input parameters are valid, `StartScheduleTableSynchron()` shall start the processing of schedule table <ScheduleTableID> at its first expiry point after the global time of the schedule table is set via `SyncScheduleTable()` and global time equals `GlobalTime`.	
Caveats:	None	
Configuration:	Available in Scalability Classes 2 and 4.	

8.4.16 SyncScheduleTable

Service name:	SyncScheduleTable	
Syntax:	`StatusType SyncScheduleTable ( ScheduleTableType ScheduleTableID, GlobalTimeTickType GlobalTime )`	
Service ID:	`OSServiceId_SyncScheduleTable`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID`	Schedule table
	`GlobalTime`	The current value of the global time.
Parameters (out):	None	
Return value:	`E_OK`	No errors
	`E_OS_ID` (only in `EXTENDED` status)	The `ScheduleTableID` was not valid or schedule table can not be synchronized (`LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION` = FALSE).
	`E_OS_STATE` (only in `EXTENDED` status)	The state of schedule table <`ScheduleTableID`> is equal to `SCHEDULETABLE_NOT_STARTED`.
Description:	This service provides the operating system with the current global time. It is used to synchronize the processing of the schedule table to global time.	
Caveats:	None	
Configuration:	Available in Scalability Classes 2 and 4.	

8.4.17 SetScheduleTableAsync

Service name:	SetScheduleTableAsync	
Syntax:	`StatusType SetScheduleTableAsync ( ScheduleTableType ScheduleTableID )`	
Service ID:	`OSServiceId_SetScheduleTableAsync`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID`	Name of schedule for which status is requested
Parameters (out):	None	
Return value:	`E_OK`	No Error
	`E_OS_ID` (only in `EXTENDED` status)	Invalid `ScheduleTableID`
Description:	OS300: `SetScheduleTableAsync()` shall set the synchronization status of the `ScheduleTableID` to asynchronous. OS362: If SetScheduleTableAsync() is called for a running schedule table the OS shall stop further synchronization until a SyncScheduleTable() call is made. OS323: If `SetScheduleTableAsync()` is called for a running schedule table the OS shall continue to process expiry points on the schedule table.	
Caveats:	None	
Configuration:	Available in Scalability Classes 2 and 4.	

8.4.18 GetScheduleTableStatus

Service name:	GetScheduleTableStatus	
Syntax:	`StatusType GetScheduleTableStatus ( ScheduleTableType ScheduleTableID, ScheduleTableStatusRefType ScheduleStatus )`	
Service ID:	`OSServiceId_GetScheduleTableStatus`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`ScheduleTableID`	Schedule table for which status is requested
Parameters (out):	`ScheduleStatus`	Reference to ScheduleStatusType
Return value:	`E_OK`	No Error
	`E_OS_ID` (only in `EXTENDED` status)	Invalid `ScheduleTableID`
Description:	OS289: If the schedule table <ScheduleTableID> is not yet started, `GetScheduleTableStatus()` shall pass back `SCHEDULETABLE_NOT_STARTED` via the reference parameter <ScheduleStatus> and shall return `E_OK`. OS353: If the schedule table <ScheduleTableID> 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`.	

	OS354: If the schedule table <ScheduleTableID> is configured with hard synchronization strategy and no global time was provided to the Operating System, `GetScheduleTableStatus()` shall return `SCHEDULETABLE_WAITING` via the reference parameter <ScheduleStatus> and shall return `E_OK`. OS290: If the schedule table <ScheduleTableID> is started AND synchronous, `GetScheduleTableStatus()` shall pass back `SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS` via the reference parameter <ScheduleStatus> and shall return `E_OK`. OS291: If the schedule table <ScheduleTableID> is started, but it is not synchronous (deviation is not within the precision interval or the global time is not available), `GetScheduleTableStatus()` shall pass back `SCHEDULETABLE_RUNNING` via the reference parameter ScheduleStatus and shall return `E_OK`. OS293: If the identifier <ScheduleTableID> is not valid, `GetScheduleTableStatus()` shall return `E_OS_ID`.
Caveats:	None
Configuration:	Available in all Scalability Classes.

8.4.19 TerminateApplication

Service name:	TerminateApplication	
Syntax:	`StatusType TerminateApplication(RestartType RestartOption)`	
Service ID:	`OSServiceId_TerminateApplication`	
Sync/Async:	Normally does not return to the caller, except called in the wrong context: sync.	
Reentrancy:	Yes	
Parameters (in):	`RestartOption`	Either `RESTART` for doing a restart of the OS-Application or `NO_RESTART` if OS-Application shall not be restarted.
Parameters (out):	None	
Return value:	`E_OS_CALLEVEL`	Called in the wrong context.
	`E_OS_ID`	RestartOption is neither `RESTART` nor `NO_RESTART`.
Description:	OS287: If called from allowed context – see table [7.6.3.2.1] --, `TerminateApplication()` shall terminate the calling OS-Application (i.e. to kill all tasks and free all other OS resources associated with the application). OS288: If called from wrong context, `TerminateApplication()` shall return `E_OS_CALLEVEL`. OS346: If RestartOption equals `RESTART`, `TerminateApplication()` shall activate the configured `RESTARTTASK` of the terminated OS-Application.	
Caveats:	If no applications are configured the implementation shall make sure that this service is not available.	
Configuration:	Available in Scalability Classes 3 and 4.	

8.4.20 DisableInterruptSource

Service name:	DisableInterruptSource	
Syntax:	`StatusType DisableInterruptSource (ISRType DisableISR)`	
Service ID:	`OSServiceId_ DisableInterruptSource`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`DisableISR`	ISR which will be disabled by the service
Parameters (out):	None	
Return value:	`E_OK`	No error
	`E_OS_ID` (only in `EXTENDED` status)	Invalid `DisableISR`
Description:	OS339: `DisableInterruptSource()` shall disable all interrupt sources which can invoke ISR <DisableISR>.	
Caveats:	None	
Configuration:	Available in Scalability Classes 2 and 4.	

8.4.21 EnableInterruptSource

Service name:	EnableInterruptSource	
Syntax:	`StatusType EnableInterruptSource (ISRType EnableISR)`	
Service ID:	`OSServiceId_ EnableInterruptSource`	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`EnableISR`	ISR which might be enabled by the service
Parameters (out):	None	
Return value:	`E_OK`	No error
	`E_OS_ID` (only in `EXTENDED` status)	Invalid `EnableISR`
	`E_OS_NOFUNC`	Arrival rate of ISR is reached, <EnableISR> will be enabled by the OS at the start of the next timeframe.
Description:	OS340: If the arrival rate of ISR <EnableISR> is not reached, `EnableInterruptSource()` shall enable all interrupt sources, which can invoke ISR <EnableISR>, immediately and return `E_OK`. OS341: If the arrival rate of ISR <EnableISR> is reached, `EnableInterruptSource()` shall request to enable all interrupt sources, which can invoke ISR <EnableISR>, at the start of the next timeframe and return `E_OS_NOFUNC`. OS342: If interrupt sources are not disabled by `DisableInterruptSource()` at the start of the next timeframe, the OS shall enable all requested interrupt sources.	
Caveats:	None	
Configuration:	Available in Scalability Classes 2 and 4.	

8.5 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 [9] and/or extensions from 7.9 may be called by the OS.

8.5.1 Protection Hook

Service name:	ProtectionHook	
Syntax:	`ProtectionReturnType ProtectionHook ( StatusType Fatalerror )`	
Service ID:	Not a user service, so no ID.	
Sync/Async:	Sync	
Reentrancy:	Yes	
Parameters (in):	`Fatalerror`	The error which caused the call to the protection hook
Parameters (out):	None	
Return value:	`PRO_KILLTASKISR`	The number defines the action the OS shall take after the protection hook
	`PRO_KILLAPPL`	
	`PRO_KILLAPPL_RESTART`	
	`PRO_SHUTDOWN`	
Description:	The protection hook is always called if a serious error occurs. E.g. exceeding the worst case execution time or violating against the memory protection. Depending on the return value the OS will either kill the Task/Category 2 ISR which causes the problem, kill the OS-Application the Task/Category 2 ISR belong (optional with restart) or shutdown the system.	
Caveats:	OS308: If an invalid value is returned the Operating System shall take the same action as if no protection hook is configured.	
Configuration:	Available in Scalability Classes 2, 3 and 4.	

9 Sequence diagrams

9.1 Sequence chart for calling trusted functions

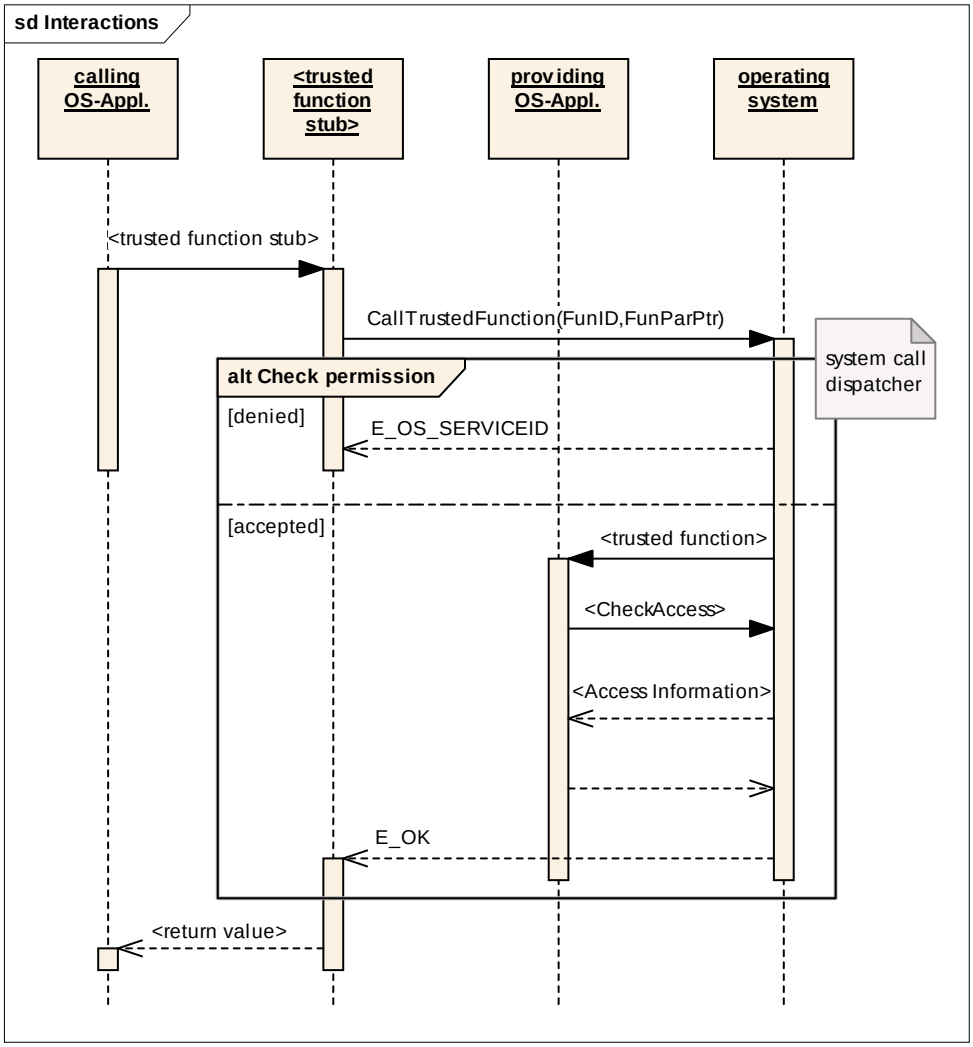

Fig. 13: System Call sequence chart

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

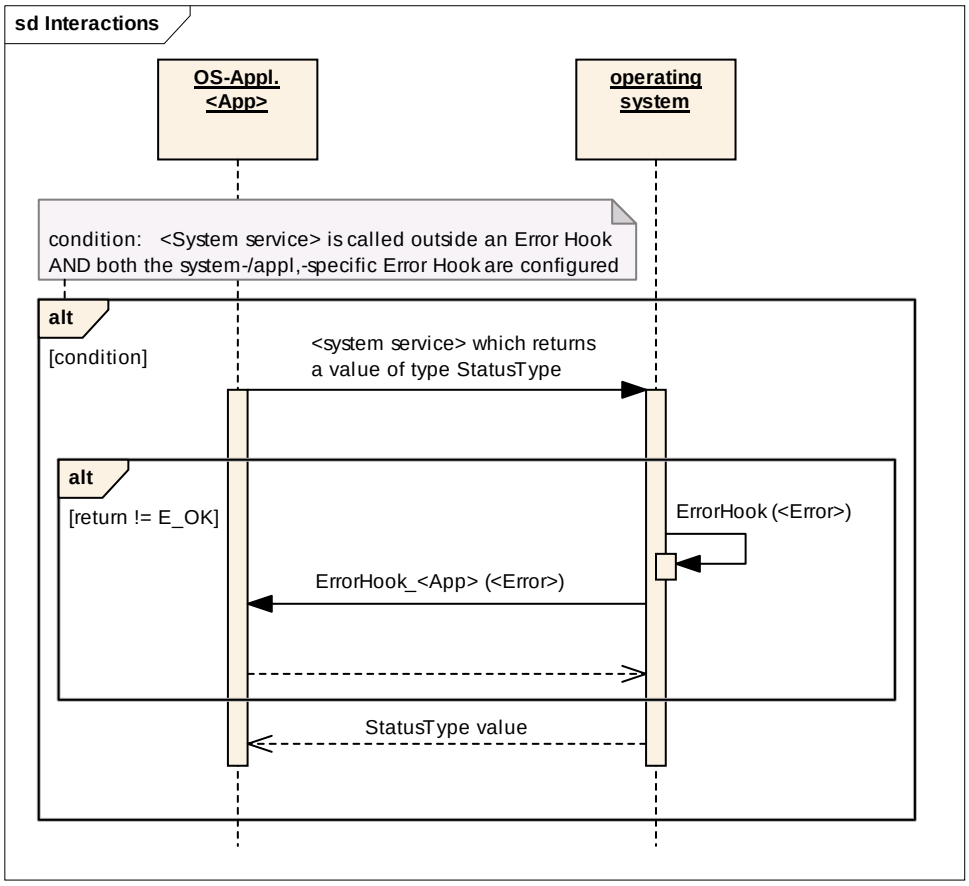

Fig. 14: Error Hook sequence chart

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

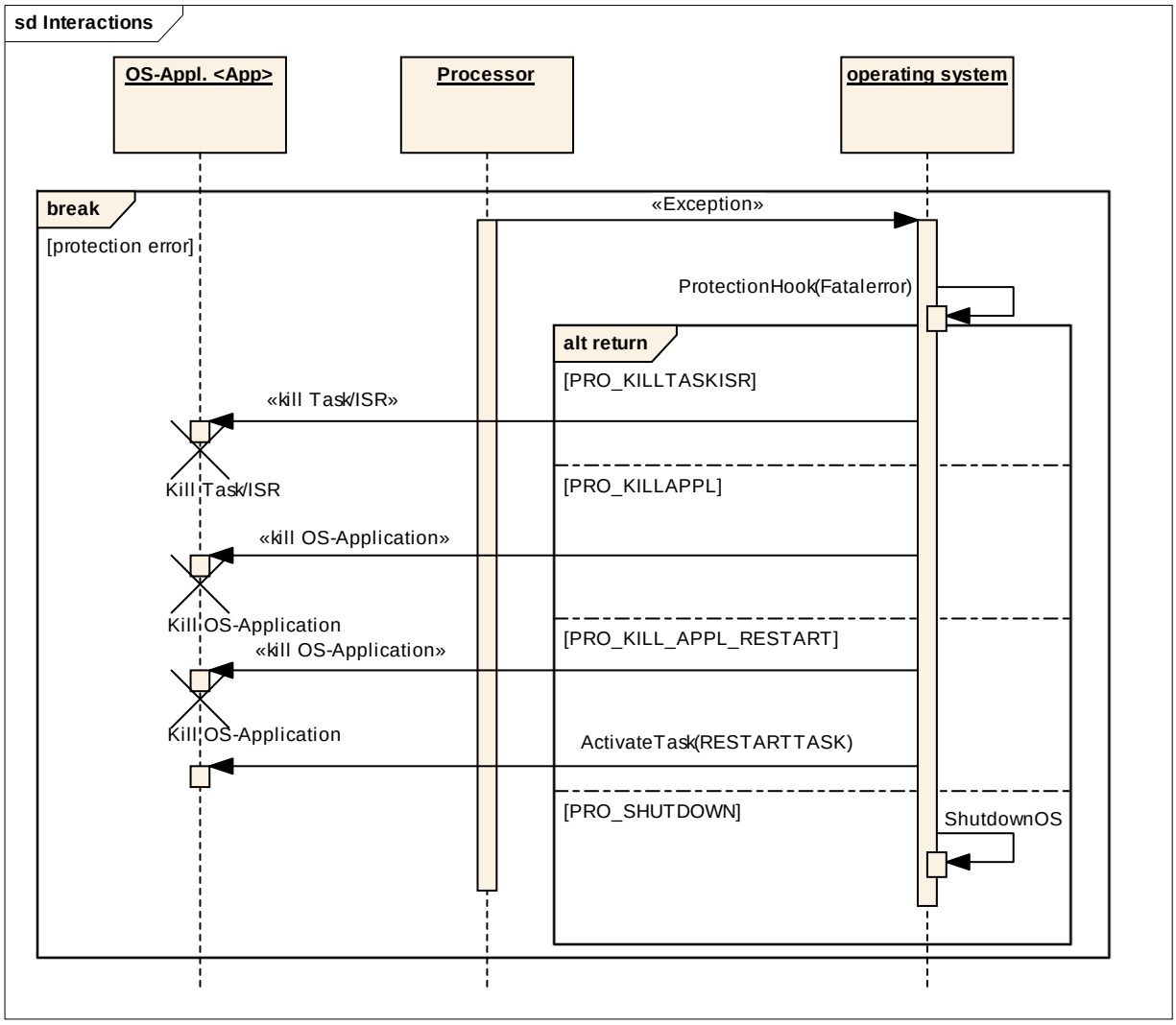

Fig. 15: Protection Hook sequence chart

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 killed or the OS-Application is killed or the system is shut down. If the action is to kill 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

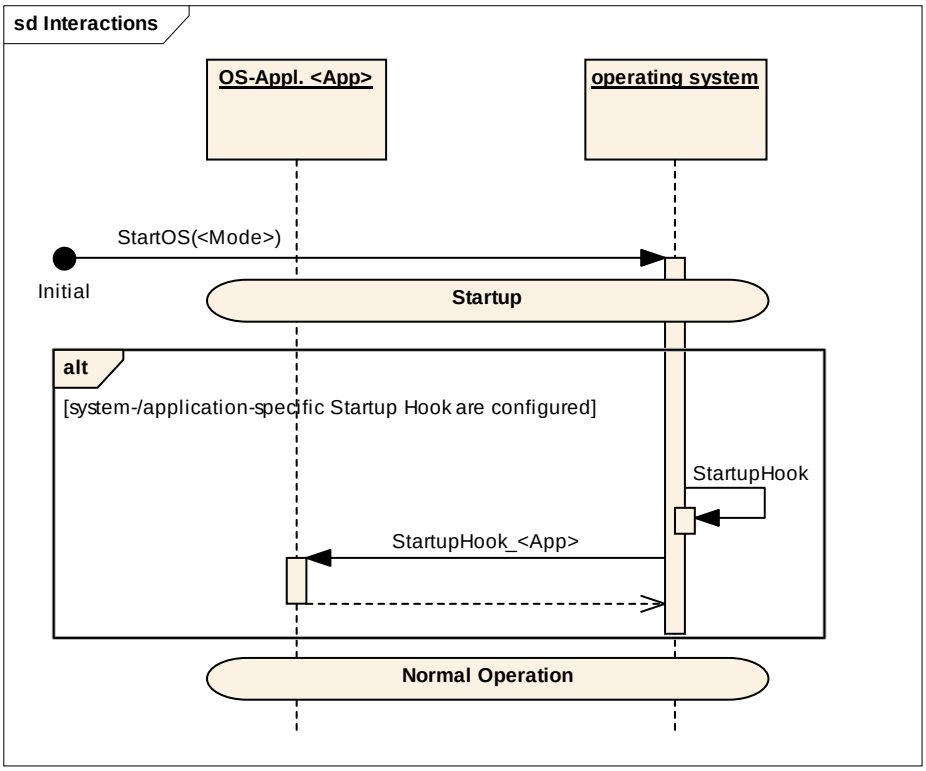

Fig. 16: StartupHook sequence chart

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 is 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.

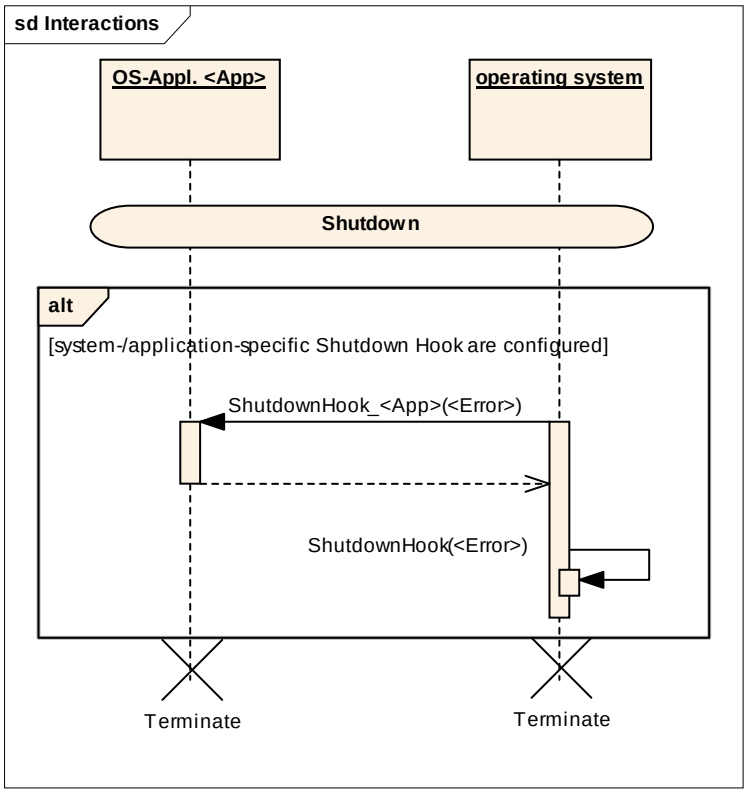

Fig. 17: ShutdownHook sequence chart

10 Configuration specification

10.1 Introduction

The configuration language for the Operating System is an extension to OSEK OIL [9]. AUTOSAR OS uses OIL instead of XML to keep compatibility with OSEK OS. It is possible to convert XML to OIL and vice versa.

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term “configuration class” (of a parameter) shall be used in order to refer to a specific configuration point in time.

Containers structure the set of configuration parameters. This means *all* configuration parameters are kept in containers. The content of the containers is mapped to OIL objects of the same name.

Note that not all attributes may be available in all scalability class.

Memory protection configuration is not standardized and therefore not part of this specification.

10.1.1 General Requirements

OS113: The Configuration language shall permit the selection of a processor.

OS329: The OIL version of the used configuration files shall be “3.0”

10.1.2 System Object »OS«

OS214: The Configuration language shall allow the user to define at most one Protection-Hook (attribute in System Object »OS«: `BOOLEAN PROTECTIONHOOK`).

OS259: The Configuration language shall permit the selection of a scalability class. This attribute is used for cross checking and uses the OIL `AUTO` mechanism (attribute in System Object »OS«: `ENUM WITH_AUTO [SC1,SC2,SC3,SC4] SCALABILITYCLASS = AUTO;`).

OS307: The Configuration language shall permit if stack monitoring is applied within the system or not (attribute in System Object »OS«: `BOOLEAN STACKMONITORING;`). Stack monitoring applies to Tasks and Category 2 ISRs based on vendor specific attributes defining the stack size.

10.1.3 System Object »APPLICATION«

OS114: The Configuration language shall allow the user to define zero or more OS-Applications up to an implementation specific maximum number (System Object: `APPLICATION`).

OS254: The Configuration language shall allow the user to define trusted functions (as part of a trusted OS-Application) available to other OS-Applications (attribute in System Object »APPLICATION«: `BOOLEAN [TRUE {STRING NAME;},FALSE ] TRUSTED_FUNCTION[];`)

10.1.3.1 Configuring Trusted Applications

OS115: The Configuration language shall provide the user the ability of determining each OS-Application to be trusted (attribute in System Object »APPLICATION«: `BOOLEAN TRUSTED = FALSE;`).

10.1.3.2 Application-specific Hooks

OS124: The Configuration language shall allow the user to define at most one Startup-Hook for an OS-Application (attribute in System Object »APPLICATION«: `BOOLEAN STARTUPHOOK`).

OS125: The Configuration language shall allow the user to define at most one Shutdown-Hook for an OS-Application (attribute in System Object »APPLICATION«: `BOOLEAN SHUTDOWNHOOK`).

OS213: The Configuration language shall allow the user to define at most one Error-Hook for an OS-Application (attribute in System Object »APPLICATION«: `BOOLEAN ERRORHOOK`).

10.1.3.3 Re-start Task

OS120: The Configuration language shall allow the user to define optionally one task of an OS-Application as Re-start Task of the same OS-Application (attribute in System Object »APPLICATION«: `BOOLEAN [ TRUE {TASK_TYPE RESTARTTASK;}, FALSE {}] HAS_RESTARTTASK;`).

10.1.3.4 OS-Objects

OS116: The Configuration language shall provide the user the ability of assigning each task to exactly one OS-Application (attribute in System Object »APPLICATION«: `TASK_TYPE TASK[]`).

OS221: The Configuration language shall provide the user the ability of assigning each ISR to exactly one OS-Application (attribute in System Object »APPLICATION«: `ISR_TYPE ISR[]`).

OS231: The Configuration language shall provide the user the ability of assigning each Alarm to exactly one OS-Application (attribute in System Object »APPLICATION«: `ALARM_TYPE ALARM[]`).

OS230: The Configuration language shall provide the user the ability of assigning each schedule table to exactly one OS-Application (attribute in System Object »APPLICATION«: `SCHEDULETABLE_TYPE SCHEDULETABLE[]`).

OS234: The Configuration language shall provide the user the ability of assigning each counter to exactly one OS-Application (attribute in System Object »APPLICATION«: `COUNTER_TYPE COUNTER[]`).

OS248: The Configuration language shall provide the user the ability of assigning each resource to exactly one OS-Application (attribute in System Object »APPLICATION«: `RESOURCE_TYPE RESOURCE[]`).

OS253: The Configuration language shall provide the user the ability of assigning each message to exactly one OS-Application (attribute in System Object »APPLICATION«: `MESSAGE_TYPE MESSAGE[]`).

10.1.4 System Object »SCHEDULETABLE«

OS141: The Configuration language shall allow the user to define several schedules (System Object »`SCHEDULETABLE`«).

OS145: The Configuration language shall allow the user to define the counter which drives the schedule table (attribute in System Object »SCHEDULETABLE«: `COUNTER_TYPE COUNTER;`)

OS143: The Configuration language shall allow the user to define the actions (»activate a task« or »set an event« with `OFFSET` ticks/nanoseconds relative to the start of the period) of a schedule (attribute in System Object »SCHEDULETABLE«: ENUM [`ACTIVATETASK { UINT64 OFFSET; TASK_TYPE TASK; }, SETEVENT { UINT64 OFFSET; EVENT_TYPE EVENT; TASK_TYPE TASK; } ] ACTION []`;)

OS144: The Configuration language shall allow the user to define a `PERIOD` in ticks/nanoseconds of a schedule table (attribute in System Object »SCHEDULETABLE« of `BOOLEAN PERIODIC, UINT64 LENGTH;`).

OS249: The Configuration language shall allow the user to define which OS-Applications have access to the schedule table (attribute in System Object »SCHEDULETABLE« of `APPLICATION_TYPE ACCESSING_APPLICATION[];`)

OS310: The Configuration language shall allow the user to define the synchronization of a schedule table (attribute in System Object »SCHEDULETABLE«: `BOOLEAN [TRUE { ENUM [HARD, SMOOTH ] SYNC_STRATEGY = HARD; UINT64 MAX_CORRECTION_SYNC; UINT64 MAX_CORRECTION_ASYNC; UINT64 PRECISION;}, FALSE ] LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION ;`). (The `MAX_CORRECTION_SYNC, MAX_CORRECTION_ASYNC` and `PRECISION` values are in nanoseconds.)

OS335: The Configuration language shall allow the user to define if a schedule table should be automatically started with a relative offset to the start of the Operating System. (attribute in System Object »SCHEDULETABLE«: `BOOLEAN [ TRUE { UINT64 OFFSET; APPMODE_TYPE APPMODE[]; }, FALSE ] AUTOSTART;`**)**

10.1.5 System Object »TASK«

OS185: The Configuration language shall allow the user to define the execution time budget (`EXECUTIONBUDGET` nanoseconds per `TIMEFRAME`) for each task (attribute in System Object »TASK«: `UINT64 TIMEFRAME`, `UINT64 EXECUTIONBUDGET`).

OS188: The Configuration language shall allow the user to define an Interrupt Lock Time and/or a Resource Lock Time in nanoseconds of an OSEK resource in nanoseconds (attribute in System Object »TASK«:

```
UINT64 MAXOSINTERRUPTLOCKTIME;
UINT64 MAXALLINTERRUPTLOCKTIME;
ENUM [
   RESOURCELOCK {
      RESOURCE_TYPE RESOURCE;
      UINT64 MAXRESOURCELOCKTIME;
   }
] LOCKINGTIME [];).
```

OS325: The configuration language shall group all parameters related to timing protection of a TASK (OS185, OS188) in one ENUM (attribute in System Object »TASK«: `TIMING_PROTECTION = TRUE` or FALSE (default is FALSE))

OS250: The Configuration language shall allow the user to define which OS-Applications have access to the task (attribute in System Object »TASK« of `APPLICATION_TYPE ACCESSING_APPLICATION[];`)

10.1.6 System Object »ALARM«

OS251: The Configuration language shall allow the user to define which OS-Applications have access to the alarm (attribute in System Object »ALARM« of `APPLICATION_TYPE ACCESSING_APPLICATION[];`)

OS302: The Configuration language shall allow the user to define to increment a counter as action on alarm expiry. (Additional alternative in “ACTION” attribute

INCREMENTCOUNTER with a reference to the counter to be incremented (COUNTER_TYPE COUNTER))

10.1.7 System Object »RESOURCE«

OS252: The Configuration language shall allow the user to define which OS-Applications have access to the resource (attribute in System Object »RESOURCE« of APPLICATION_TYPE ACCESSING_APPLICATION[];)

10.1.8 System Object »COUNTER«

OS317: The Configuration language shall allow the user to define which OS-Applications have access to the counter (attribute in System Object »COUNTER« of APPLICATION_TYPE ACCESSING_APPLICATION[];)

OS255: The Configuration language shall allow the user to define the type of a counter (attribute in System Object »COUNTER«: ENUM [SOFTWARE, HARDWARE] TYPE;).

OS331: The Configuration language shall allow the user to define the unit type of the counter (attribute in System Object »COUNTER«: ENUM [TICKS, NANOSECONDS] UNIT;)

OS371: The Configuration language shall allow the user to define the driver of a counter (for counters of type HARDWARE) (attribute in System Object »COUNTER«, subattribute of HARDWARE: ENUM [OSINTERNAL, GPT] DRIVER;)[2].

OS385: The Configuration language shall allow the user to define the time for one tick of a hardware timer (for counters of type HARDWARE) in nanoseconds (attribute in System Object »COUNTER«, subattribute of HARDWARE: UINT64 NS_PER_HW_TICK;)

OS386: The Configuration language shall allow the user to define constants which can be e.g. used to compare time values with timer tick values. (attribute in System Object »COUNTER«, subattribute of HARDWARE:

```
ENUM [
    TIMECONSTANT {
        UINT64 NS;
        STRING CONSTNAME;
    }
] TIMECONSTANTS [])
```

OS390: The Configuration language shall allow the user to define the name of the timer and the time for one timer tick (in nanoseconds) if the type of the counter is HARDWARE and the GPT is used for this counter. (optional; attribute in System Object »COUNTER«, subattribute of GPT:

[2] The configuration of the GPT driver is an optional feature.

```
UINT64 NS_PER_HW_TICK; STRING GPTCHANNELNAME;)3
```

OS375: The Configuration language shall allow the user to define all required values to setup the range and resolution of a counter of type HARDWARE. [4]

10.1.9 System Object »MESSAGE«

OS316: The Configuration language shall allow the user to define which OS-Applications have access to the message (attribute in System Object »MESSAGE« of APPLICATION_TYPE ACCESSING_APPLICATION[];)

10.1.10 System Object »ISR«

OS222: The Configuration language shall allow the user to define a worst case execution time in nanoseconds for each ISR (attribute in System Object »ISR«: UINT64 EXECUTIONTIME).

OS223: The Configuration language shall allow the user to define the maximum arrival rate (=COUNTLIMIT times in TIMEFRAME nanoseconds) for each ISR (attribute in System Object »ISR«: UINT32 COUNTLIMIT, UINT64 TIMEFRAME).

OS229: The Configuration language shall allow the user to define an Interrupt Lock Time and/or a Resource Lock Time in nanoseconds of an OSEK resource in nanoseconds (attribute in System Object »ISR«:

```
UINT64 MAXOSINTERRUPTLOCKTIME;
UINT64 MAXALLINTERRUPTLOCKTIME;
ENUM [
   RESOURCELOCK {
      RESOURCE_TYPE RESOURCE;
      UINT64 MAXRESOURCELOCKTIME;
   }
] LOCKINGTIME [];) .
```

OS326: The configuration language shall group all parameters related to timing protection of a ISR (OS222, OS223, OS229) in one ENUM (attribute in System Object »ISR«: TIMING_PROTECTION = TRUE or FALSE (default is FALSE))

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters and their containers. The detailed meanings of the parameters describe Chapters 7 and 8.

[3] This CHANNELNAME is required by the OS to call GPT services. The name must match the name of the timer in the GPT configuration.
[4] There are no standard attributes defined, since the required infotmation depends highly on the used MCU.

10.2.1 System object OS

SWS Item	OS214, OS259, OS307
Container Name	OS
Description	Extensions to the system object OS
Configuration Parameters	

Name	PROTECTIONHOOK		
Description	Switch to enable/disable the call to the (user supplied) protection hook.		
Type or Unit	BOOLEAN		
Range	TRUE	Protection hook is called on protection error	
	FALSE	Protection hook is not called	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2,3 and 4		

Name	SCALABILITYCLASS		
Description	Select a scalability class		
Type or Unit	ENUM WITH_AUTO		
Range	SC1, SC2, SC3, SC4		
	AUTO	Default, select scalability class according configured features	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	STACKMONITORING		
Description	Select stack monitoring of Tasks/Category 2 ISRs		
Type or Unit	BOOLEAN		
Range	TRUE	Stacks are monitored	
	FALSE	Stacks are not monitored	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.2 System object APPLICATION

SWS Item	OS114, OS254, OS115, OS124, OS125, OS213, OS120, OS116, OS221, OS231, OS230, OS234, OS248, OS253		
Container Name	APPLICATION		
Description	New system object to describe the parameters of OS-Applications.		
Configuration Parameters			
Name	TRUSTED_FUNCTION		
Description	List of trusted functions with their attributes (like NAME).		
Type or Unit	BOOLEAN [TRUE {STRING NAME;},FALSE]		
Range	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4 and in trusted OS-Applications.		

Name	BOOLEAN TRUSTED = FALSE		
Description	Parameter to specify if an OS-Application is trusted or not.		
Type or Unit	BOOLEAN		
Range	TRUE	OS-Application is trusted	
	FALSE	OS-Application is not trusted (default)	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	STARTUPHOOK		
Description	Select the OS-Application specific startup hook for the OS-Application.		
Type or Unit	BOOLEAN		
Range	TRUE	Hook is called	
	FALSE	Hook is not called	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	SHUTDOWNHOOK		
Description	Select the OS-Application specific shutdown hook for the OS-Application.		
Type or Unit	BOOLEAN		
Range	TRUE	Hook is called	
	FALSE	Hook is not called	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	ERRORHOOK
Description	Select the OS-Application error hook.

Type or Unit	BOOLEAN		
Range	TRUE	Hook is called	
	FALSE	Hook is not called	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	HAS_RESTARTTASK		
Description	Selects if the OS-Application have a restart task or not. If set to TRUE a task must be specified.		
Type or Unit	BOOLEAN [TRUE {TASK_TYPE RESTARTTASK;}, FALSE {}]		
Range	TRUE	Restart Task RESTARTTASK is activated by the Operating System if the protection hook requests it.	
	FALSE	No task is automatically started after a protection error happened.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	TASK[]		
Description	A list of all tasks which belong to the OS-Application.		
Type or Unit	TASK_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	ISR[]		
Description	A list of all ISRs which belong to the OS-Application.		
Type or Unit	ISR_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	ALARM[]		
Description	A list of all alarms which belong to the OS-Application.		
Type or Unit	ALARM_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	

Scope	ECU
Dependency	Required for scalability class 3 and 4.

Name	SCHEDULETABLE[]		
Description	A list of all schedule tables which belong to the OS-Application.		
Type or Unit	SCHEDULETABLE_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	COUNTER[]		
Description	A list of all counters which belong to the OS-Application.		
Type or Unit	COUNTER_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	RESOURCE[]		
Description	A list of all OSEK resources which belong to the OS-Application.		
Type or Unit	RESOURCE_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Name	MESSAGE[]		
Description	A list of all messages which belong to the OS-Application.		
Type or Unit	MESSAGE_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4.		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.3 System Object »SCHEDULETABLE«

SWS Item	OS141, OS143, OS145, OS144, OS249, OS310, OS335
Container Name	SCHEDULETABLE
Description	New system object which contains the parameters for a schedule table.
Configuration Parameters	

Name	COUNTER		
Description	This parameter contains a reference to the counter which drives the schedule table.		
Type or Unit	COUNTER_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	ACTION[]		
Description	List of defined expiry points with the distance from the first expiry point of the schedule table (OFFSET) and their associated action (either ACTIVATETASK or SETEVENT).		
Type or Unit	ENUM [ACTIVATETASK { UINT64 OFFSET; TASK_TYPE TASK; }, SETEVENT { UINT64 OFFSET; EVENT_TYPE EVENT; TASK_TYPE TASK; }]		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	PERIODIC		
Description	Defines if the schedule table is periodic or not.		
Type or Unit	BOOLEAN		
Range	TRUE	Schedule table is periodic	
	FALSE	Schedule table is non-periodic (single-shot)	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	LENGTH		
Description	Defines the length of the schedule table.		
Type or Unit	UINT64 – The exact unit (time or ticks) is derived from the counter type.		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	ACCESSING_APPLICATION[]		
Description	List of OS-Applications which have access to the schedule table (e.g. they can start or stop the schedule table).		
Type or Unit	APPLICATION_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION		
Description	This parameter specifies the synchronization parameters of the schedule table. The base type is BOOLEAN and defines if synchronization is required in general. Sub parameters define the startup behavior and the maximal adjustments. The timing parameters (MAX_CORRECTION_SYNC, MAX_CORRECTION_ASYNC and PRECISION) are given in nanoseconds.		
Type or Unit	BOOLEAN [TRUE { ENUM [HARD, SMOOTH] SYNC_STRATEGY = HARD; UINT64 MAX_CORRECTION_SYNC; UINT64 MAX_CORRECTION_ASYNC; UINT64 PRECISION;}, FALSE]		
Range	TRUE	--	
	FALSE	Default value, no synchronization done.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	System		
Dependency	--		

Name	AUTOSTART		
Description	This parameter specifies if the schedule table is started on startup of the Operating System with a given relative offset.		
Type or Unit	BOOLEAN [TRUE { UINT64 OFFSET; APPMODE_TYPE APPMODE[]; }, FALSE]		
Range	TRUE	Schedule table is started with offset OFFSET if Operating System starts with an application mode of APPMODE[].	
	FALSE	Schedule table does not start at start of the Operating System.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.4 System Object »TASK«

SWS Item	OS185, OS188, OS250, OS325
Container Name	TASK
Description	Extensions to the system object OS
Configuration Parameters	

Name	EXECUTIONBUDGET TIMEFRAME		
Description	These two parameters contain the allowed execution time of the task. The time is specified as a time budget (EXECUTIONBUDGET) in a define time frame (TIMEFRAME)		
Type or Unit	UINT64 (for both parameters, unit is nanoseconds)		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	MAXOSINTERRUPTLOCKTIME MAXALLINTERRUPTLOCKTIME		
Description	These parameters contain the maximum lock times the task locks out interrupts.		
Type or Unit	UINT64 MAXOSINTERRUPTLOCKTIME; UINT64 MAXALLINTERRUPTLOCKTIME;		
Range	MAXOSINTERRUPTLOCKTIME	This parameter contains the longest time in nanoseconds, where the task locks all category 2 interrupts (via SuspendOSInterrupts())	
	MAXALLINTERRUPTLOCKTIME	This parameter contains the longest time in nanoseconds, where the task locks all interrupts (via SuspendAllInterrupts() or DisableAllInterrupts())	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	LOCKINGTIME[]		
Description	This parameter contains a list of times the task holds resources.		

Type or Unit	ENUM [
	RESOURCELOCK {
	RESOURCE_TYPE RESOURCE;
	UINT64 MAXRESOURCELOCKTIME;
	}
	]		

Range	RESOURCE	This parameter contains the reference to the resource used by the task.	
	MAXRESOURCELOCKTIME	This parameter contains the time (in nanoseconds) of the task in which the task has holds the resource ..	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	TIMING_PROTECTION		
Description	This parameter defines the timing protection for the task. If enabled it contains all other parameters regarding timing protection of the task.		
Type or Unit	ENUM [TRUE { /* all timing parameters */ }, FALSE]		
Range	TRUE	This enables the timing protection for the task.	
	FALSE	Default. This disables the timing protection for this task. If the task belongs to a non-trusted OS-Application the parameter is always TRUE.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	ACCESSING_APPLICATION[]		
Description	List of OS-Applications which have access to the task (e.g. they can activate the task).		
Type or Unit	APPLICATION_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.5 System Object »ALARM«

SWS Item	OS251, OS302
Container Name	ALARM
Description	Extensions to the system object ALARM
Configuration Parameters	

Name	INCREMENTCOUNTER		
Description	This parameter specifies an additional alarm action. Besides the standard task activation, event setting and calling a callback function, INCREMENTCOUNTER allows the incrementation of a counter.		
Type or Unit	ENUM [… INCREMENTCOUNTER {COUNTER_TYPE COUNTER;}, …]		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	ACCESSING_APPLICATION[]		
Description	List of OS-Applications which have access to the alarm (e.g. they can start the alarm).		
Type or Unit	APPLICATION_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.6 System Object »RESOURCE«

SWS Item	OS252
Container Name	RESOURCE
Description	Extensions to the system object RESOURCE
Configuration Parameters	

Name	ACCESSING_APPLICATION[]
Description	List of OS-Applications which have access to the alarm (e.g. they can start the alarm).
Type or Unit	APPLICATION_TYPE

Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.7 System Object »COUNTER«

SWS Item	OS317, OS255, OS331
Container Name	COUNTER
Description	Extensions to the system object COUNTER
Configuration Parameters	

Name	ACCESSING_APPLICATION[]		
Description	List of OS-Applications which have access to the alarm (e.g. they can start the alarm).		
Type or Unit	APPLICATION_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4		

Name	TYPE	
Description	This parameter contains the natural type or unit of the counter.	
Type or Unit	ENUM [SOFTWARE, HARDWARE]	
Range	SOFTWARE	The counter is a software counter.
	HARDWARE	The counter is a hardware counter.
Configuration Class	***Pre-compile***	X
	Link time	--
	Post Build	--
Scope	ECU	
Dependency	--	

Name	DRIVER	
Description	This subparameter of HARDWARE contains the information who will drive the counter.	
Type or Unit	ENUM [OSINTERNAL, GPT]	
Range	OSINTERNAL	The timer is managed by the OS internally.
	GPT { UINT64 NS_PER_HW_TICK; STRING	The OS will use the GPT interface to manage the timer. The user have to supply the channel name and the time of one hardware tick in nanoseconds. This

	GPTCHANNELNAME; }	is an optional feature and is only available if the GPT driver is used.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	TIMECONSTANTS[]		
Description	This array contains pairs of times and names. A time value will be converted to a timer tick value during generation and can later on accessed via the CONSTNAME. The conversation is done by rounding time values to the nearest fitting tick value.		

Type or Unit	ENUM [
	TIMECONSTANT {
	UINT64 NS;
	STRING CONSTNAME;
	}
	]		

Range	NS	A UINT64 which contains a time value in nanoseconds.	
	CONSTNAME	The name which is accessed by the application to get the above NS. E.g. COUNTER1_2NS.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Name	UNIT		
Description	This parameter contains the unit type of the counter.		
Type or Unit	ENUM [TICKS, NANOSECONDS]		
Range	TICKS	The timing parameters of the alarms and schedule tables which belong to the counter are given in ticks.	
	NANOSECONDS	The timing parameters of the alarms and schedule tables which belong to the counter are given in nanoseconds.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	--		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.8 System Object »MESSAGE«

SWS Item	OS316
Container Name	MESSAGE
Description	Extensions to the system object MESSAGE
Configuration Parameters	

Name	ACCESSING_APPLICATION[]		
Description	List of OS-Applications which have access to the message (e.g. they can send/receive the message).		
Type or Unit	APPLICATION_TYPE		
Range	--		
	--		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 3 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.2.9 System Object »ISR«

SWS Item	OS222, OS223, OS229, OS326
Container Name	ISR
Description	Extensions to the system object ISR
Configuration Parameters	

Name	TIMING_PROTECTION		
Description	The parameter contains all other parameters which are related to timing protection.		
Type or Unit	ENUM [TRUE { /* all timing parameters */ }, FALSE]		
Range	TRUE	Timing protection is used for this interrupt.	
	FALSE	The interrupt is not supervised regarding timing violations.	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	EXECUTIONTIME		
Description	The parameter contains the worst case execution time of the interrupt.		
Type or Unit	UINT64		
Range	--		
	--		
	Pre-compile	X	

	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	`COUNTLIMIT` `TIMEFRAME`		
Description	These parameters contain the number of interrupts (`COUNTLIMIT`) which are allowed in a specific time frame (`TIMEFRAME`).		
Type or Unit	`UINT32 COUNTLIMIT` `UINT64 TIMEFRAME`		
Range	`--`		
	`--`		
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	`MAXOSINTERRUPTLOCKTIME` `MAXALLINTERRUPTLOCKTIME`		
Description	This parameter contains the times where a interrupt locks out other interrupts		
Type or Unit	`UINT64 MAXOSINTERRUPTLOCKTIME;` `UINT64 MAXALLINTERRUPTLOCKTIME;`		
Range	`MAXOSINTERRUPTLOCKTIME`	This parameter contains the longest time (in nanoseconds) where the interrupts locks all other category 2 interrupts via SuspendOSInterrupt() .	
	`MAXALLINTERRUPTLOCKTIME`	This parameter contains the longest time (in nanoseconds) where the interrupts locks all other interrupts via SuspendAll/DisableAllInterrupt() .	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Name	`LOCKINGTIME[]`		
Description	This parameter contains a list of times the interrupts uses resources. All times in nanoseconds.		

Type or Unit	`ENUM [`
	`  RESOURCELOCK {`
	`  RESOURCE_TYPE RESOURCE;`
	`  UINT64 MAXRESOURCELOCKTIME;`
	`  }`
	`]`		

Range	`RESOURCE`	This parameter contains the reference to the resource which is used by the interrupt.	
	`MAXRESOURCELOCKTIME`	This parameter contains the maximum time the interrupt holds the given resource (in nanoseconds)	
Configuration Class	***Pre-compile***	X	
	Link time	--	
	Post Build	--	
Scope	ECU		
Dependency	Required for scalability class 2 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
--	--	--

10.3 Published Information

Published information contains data defined by the implementer of the SW module that does not change when the module is adapted (i.e. configured) to the actual HW/SW environment. It thus contains version and manufacturer information.

SWS Item	OS357:	
Information elements		
Information element name	***Type / Range***	***Information element description***
`OS_VENDOR_ID`	`#define / uint16`	Vendor ID of the dedicated implementation of this module according to the AUTOSAR vendor list
`OS_MODULE_ID`	`#define / 0x01`	Module ID of this module from Module List
`OS_AR_MAJOR_VERSION`	`#define / uint8`	Major version number of AUTOSAR specification on which the appropriate implementation is based on.
`OS_AR_MINOR_VERSION`	`#define / uint8`	Minor version number of AUTOSAR specification on which the appropriate implementation is based on.
`OS_AR_PATCH_VERSION`	`#define / uint8`	Patch level version number of AUTOSAR specification on which the appropriate implementation is based on.
`OS_SW_MAJOR_VERSION`	`#define / uint8`	Major version number of the vendor specific implementation of the module. The numbering is vendor specific.
`OS_SW_MINOR_VERSION`	`#define / uint8`	Minor version number of the vendor specific implementation of the module. The numbering is vendor specific.
`OS_SW_PATCH_VERSION`	`#define / uint8`	Patch level version number of the vendor specific implementation of the module. The numbering is vendor specific.

11 Generation of the OS

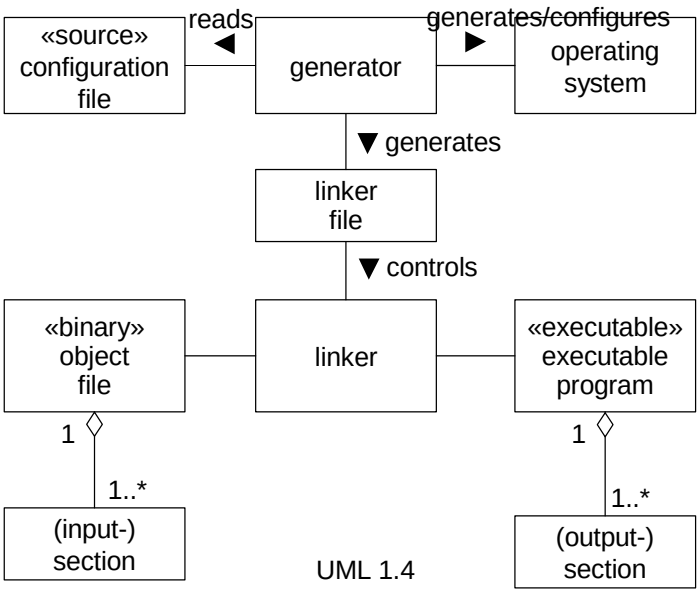

Fig. 18: Generation Activities

11.1 Read in configuration

OS172: The generator shall provide the user the ability of reading the information of a selectable configuration file.

11.2 Consistency check

OS173: The generator shall provide the user the ability of performing a consistency check of the current configuration.

OS050: If service protection is required and `STATUS` is not equal to `EXTENDED` (all the associated error handling is provided), the consistency check shall issue an error.

OS045: If timing protection is configured together with OSEK OS Category 1 interrupts, the consistency check shall issue an error.

OS320: 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 an error.

OS175: If `SCALABILITYCLASS` is SC3 or SC4 AND the same OS-object is assigned to more than one OS-Application, the consistency check shall issue an error.

OS311: If SCALABILITYCLASS is SC3 or SC4 AND a Task OR Category 2 ISR OR Resources OR Counters OR Message OR Alarms OR Schedule tables does not belong to exactly one OS-Application the consistency check shall issue an error.

OS361: If SCALABILITYCLASS 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

OS177: If SCALABILITYCLASS 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.

OS303: If INCREMENTCOUNTER is configured as action on alarm expiry AND the alarm is driven directly or indirectly (a cyclic chain of alarm actions with INCREMENTCOUNTER) by that counter, the consistency check shall issue a warning..

OS328: If STATUS is STANDARD and SCALABILITYCLASS is SC3 or SC4 the consistency check shall issue an error.

OS334: If a schedule table has no expiry point at offset zero, the consistency check shall issue a warning.

OS343: If SCALABILITYCLASS 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.

OS344: If SCALABILITYCLASS 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.

OS345: If SCALABILITYCLASS is SC3 or SC4 AND a task is referenced within a message object AND the OS-Application of the message has no access to the task, the consistency check shall issue an error.

11.3 Generating operating system

OS179: 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.

OS336: The generator shall generate a relocatable memory section containing the interrupt vector table.

OS370: The generator shall print out information about timers used internally by the OS during generation (e.g. on console).

OS393: 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()`)

12 Configuration: OIL Implementation

This chapter shows the OIL Implementation for the AUTOSAR OS. The implementation is based on the subset for internal communication (CCCA and CCCB only) for brevity. This subset is different from the full definition in the following objects:

- MESSAGE object (changes),
- NETWORKMESSAGE object (removed),
- COM object (changes),
- IPDU object (removed).

For AUTOSAR OS new objects or attributes are written in bold letters.

```
OIL_VERSION = “3.0”;

IMPLEMENTATION Standard
{
   OS
   {
      ENUM [STANDARD, EXTENDED] STATUS;
      BOOLEAN STARTUPHOOK;
      BOOLEAN ERRORHOOK;
      BOOLEAN SHUTDOWNHOOK;
      BOOLEAN PRETASKHOOK;
      BOOLEAN POSTTASKHOOK;
      BOOLEAN USEGETSERVICEID;
      BOOLEAN USEPARAMETERACCESS;
      BOOLEAN USERESSCHEDULER = TRUE;

      BOOLEAN PROTECTIONHOOK;
      ENUM WITH_AUTO [SC1,SC2,SC3,SC4] SCALABILITYCLASS = AUTO;
      BOOLEAN STACKMONITORING;
   };

   APPMODE
   {
   };

   APPLICATION
   {
      BOOLEAN [
         TRUE {
            BOOLEAN [
               TRUE {STRING NAME;},
               FALSE
            ] TRUSTED_FUNCTION[];
         },
         FALSE
      ] TRUSTED = FALSE;
      BOOLEAN STARTUPHOOK;
      BOOLEAN SHUTDOWNHOOK;
      BOOLEAN ERRORHOOK;
      BOOLEAN [
```

```
      TRUE {TASK_TYPE RESTARTTASK;},
      FALSE
   ] HAS_RESTARTTASK;
   TASK_TYPE TASK[];
   ISR_TYPE ISR[];
   ALARM_TYPE ALARM[];
   SCHEDULETABLE_TYPE SCHEDULETABLE[];
   COUNTER_TYPE COUNTER[];
   RESOURCE_TYPE RESOURCE[];
   MESSAGE_TYPE MESSAGE[];
};

TASK
{
   BOOLEAN [
      TRUE {
         APPMODE_TYPE APPMODE[];
      },
      FALSE
   ] AUTOSTART;
   UINT32 PRIORITY;
   UINT32 ACTIVATION;
   ENUM [NON, FULL] SCHEDULE;
   EVENT_TYPE EVENT[];
   RESOURCE_TYPE RESOURCE[];
   MESSAGE_TYPE MESSAGE[];

   BOOLEAN [
      TRUE {
         UINT64 EXECUTIONBUDGET;
         UINT64 TIMEFRAME;
         UINT64 MAXOSINTERRUPTLOCKTIME;
         UINT64 MAXALLINTERRUPTLOCKTIME;
         ENUM [
            RESOURCELOCK {
               RESOURCE_TYPE RESOURCE;
               UINT64 MAXRESOURCELOCKTIME;
            }
         ] LOCKINGTIME [];
      },
      FALSE
   ] TIMING_PROTECTION;

   APPLICATION_TYPE ACCESSING_APPLICATION[];

};

ISR
{
   UINT32 [1, 2] CATEGORY;
   RESOURCE_TYPE RESOURCE[];
   MESSAGE_TYPE MESSAGE[];

   BOOLEAN [
      TRUE {
         UINT64 EXECUTIONTIME;
```

```
            UINT32 COUNTLIMIT;
            UINT64 TIMEFRAME;
            UINT64 MAXOSINTERRUPTLOCKTIME;
            UINT64 MAXALLINTERRUPTLOCKTIME;
            ENUM [
               RESOURCELOCK {
                  RESOURCE_TYPE RESOURCE;
                  UINT64 MAXRESOURCELOCKTIME;
               }
            ] LOCKINGTIME [];
         },
         FALSE
      ] TIMING_PROTECTION;

   };

   COUNTER
   {
      UINT32 MINCYCLE;
      UINT32 MAXALLOWEDVALUE;
      UINT32 TICKSPERBASE;

      ENUM [
         SOFTWARE,
         HARDWARE {
            ENUM [
               OSINTERNAL,
               GPT {
                  UINT64 NS_PER_HW_TICK;
                  STRING GPTCHANNELNAME;
               }
            ] DRIVER;
            ENUM [
               TIMECONSTANT {
                  UINT64 NS;
                  STRING CONSTNAME;
               }
            ] TIMECONSTANTS []
         }
      ] TYPE;

      ENUM [
         TICKS,
         NANOSECONDS
      ] UNIT;

      APPLICATION_TYPE ACCESSING_APPLICATION[];
   };

   ALARM
   {
      COUNTER_TYPE COUNTER;
      ENUM [
         ACTIVATETASK {
            TASK_TYPE TASK;
         },
```

```
      SETEVENT {
         TASK_TYPE TASK;
         EVENT_TYPE EVENT;
      },
      ALARMCALLBACK {
         STRING ALARMCALLBACKNAME;
      },
      INCREMENTCOUNTER {
         COUNTER_TYPE COUNTER;
      }
   ] ACTION;
   BOOLEAN [
      TRUE {
         UINT32 ALARMTIME;
         UINT32 CYCLETIME;
         APPMODE_TYPE APPMODE[];
      },
      FALSE
   ] AUTOSTART;

   APPLICATION_TYPE ACCESSING_APPLICATION[];
};

EVENT
{
   UINT64 WITH_AUTO MASK;
};

RESOURCE
{
   ENUM [
      STANDARD,
      LINKED {
         RESOURCE_TYPE LINKEDRESOURCE;
      },
      INTERNAL
   ] RESOURCEPROPERTY;

   APPLICATION_TYPE ACCESSING_APPLICATION[];
};

MESSAGE
{
   ENUM [
      SEND_STATIC_INTERNAL {
         STRING CDATATYPE;
      },
      RECEIVE_UNQUEUED_INTERNAL {
         MESSAGE_TYPE SENDINGMESSAGE;
         UINT64 INITIALVALUE = 0;
      },
      RECEIVE_QUEUED_INTERNAL {
         MESSAGE_TYPE SENDINGMESSAGE;
         UINT32 QUEUESIZE;
      }
   ] MESSAGEPROPERTY;
```

```
      ENUM [
         NONE,
         ACTIVATETASK {
            TASK_TYPE TASK;
         },
         SETEVENT {
            TASK_TYPE TASK;
            EVENT_TYPE EVENT;
         },
         COMCALLBACK {
            STRING CALLBACKROUTINENAME;
            MESSAGE_TYPE MESSAGE[];
         },
         FLAG {
            STRING FLAGNAME;
         }
      ] NOTIFICATION = NONE;

      APPLICATION_TYPE ACCESSING_APPLICATION[];
   };

   COM
   {
      BOOLEAN COMERRORHOOK = FALSE;
      BOOLEAN COMUSEGETSERVICEID = FALSE;
      BOOLEAN COMUSEPARAMETERACCESS = FALSE;
      BOOLEAN COMSTARTCOMEXTENSION = FALSE;
      STRING COMAPPMODE[];
      ENUM [
         COMSTANDARD,
         COMEXTENDED
      ] COMSTATUS = COMSTANDARD;
   };

   NM
   {
   };

   SCHEDULETABLE
   {
       COUNTER_TYPE COUNTER;

        BOOLEAN [
           TRUE {
              UINT64 OFFSET;
              APPMODE_TYPE APPMODE[];
           },
           FALSE
        ] AUTOSTART;

      BOOLEAN [
         TRUE {
            ENUM [HARD, SMOOTH ] SYNC_STRATEGY = HARD;
            UINT64 MAX_CORRECTION_SYNC;
            UINT64 MAX_CORRECTION_ASYNC;
            UINT64 PRECISION;
```

```
         },
         FALSE
      ] LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION = FALSE;

      BOOLEAN PERIODIC;
      UINT64 LENGTH;

      ENUM [
         ACTIVATETASK {
            UINT64 OFFSET;
            TASK_TYPE TASK;
         },
         SETEVENT {
            UINT64 OFFSET;
            EVENT_TYPE EVENT;
            TASK_TYPE TASK;
         }
      ] ACTION [];
      APPLICATION_TYPE ACCESSING_APPLICATION[];
   };
};
```

13 Application Notes

13.1 Memory Allocation

If memory protection is used in the OS, then the user must group the data and code sections into contiguous memory regions across which protection can be applied. Typically this is done using compiler-specific pragmas and/or a linker command file.

To achieve portability, the user shall group all variables belonging to a private data section (task/ISR or OS-Application) in separate files.

13.2 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).

13.3 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 element1, type2 element2, 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)
{
    type1 parameter1;
    type2 parameter2;

    /* 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(OSMEMORY_IS_WRITEABLE(CheckTaskMemoryAccess(
                GetTaskId(),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.

13.4 Migration hints for OSEKtime OS users

All important OSEKtime OS features are supported in AUTOSAR OS and it should be relatively easy to port applications from OSEKtime OS to AUTOSAR OS. However, most OSEKtime OS features are implemented slightly different requiring some porting effort. The following steps show how to proceed.

- Dispatcher tables can be implemented by using schedule tables provided by AUTOSAR OS. Synchronization to a global time base can be done in a similar way to OSEKtime by using the `SyncScheduleTable()` API call. A more elegant synchronization solution is also available by driving the schedule table directly from the global time source. However, the AUTOSAR OS implements

priority based scheduling rather than the stack based scheduling of OSEKtime. Therefore, priorities have to be chosen for the tasks. If a given OSEKtime dispatcher table has to be converted, all tasks can be given the same priority as long as there are no task preemptions. If this cannot be guaranteed, in each case where a task could be pre-empted at a dispatch point, the pre-empting task must be allocated a strictly higher priority than the task it pre-empts. Usually, there are few preemptions in OSEKtime systems, so the priorities are easy to calculate – a simple monotonically increasing priority assignment relative to the tasks position in the schedule table should suffice in most cases. Caveat: In OSEKtime, it is theoretically possible that task A pre-empts task B at one point in the dispatcher table and task B pre-empts task A at another point (however, this is rarely used in practice). Such a behaviour is not directly possible in AUTOSAR OS. It can, however, be emulated if required, either by constructing a simple state machine in the task bodies, or by adding two tasks A' and B' using the same code as tasks A and B respectively.

- Deadline monitoring is not supported by AUTOSAR OS - instead, worst-case execution time enforcement is provided. Schedulability analysis can be used to calculate whether given deadlines are met in a system of periodic tasks with given worst-case execution times.
- Reenabling of interrupts defined offline is not supported by AUTOSAR OS - instead, the AUTOSAR OS will automatically re-enable interrupt sources after a given amount of time (arrival rate monitoring).
- Tasks that have precedence over interrupt service routines are not supported by AUTOSAR OS, however, this behaviour can be easily emulated by activating a low-priority task from an ISR.
- Smooth synchronisation is achieved by adjusting the delay between adjacent expiry points, generalising OSEKtime OS' approach, where the synchronisation of the local time to the global time is done during several dispatcher rounds by extending or shortening the last ground state of the dispatcher round.

The OSEK time specification allows dispatcher rounds to take 3 modes:

1. Synchronous
2. Asynchronous/Hard
3. Asynchronous/Smooth

Users of OSEKtime who are migrating the AUTOSAR OS can define a schedule table that has the same range/tick resolution as their global time source (with an accompanying AUTOSAR OS counter that has the same resolution as the global time) and can synthesise these modes as follows:

1. Synchronous: Define `LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION=TRUE` and `SYNC_STRATEGY=HARD`. Start using `StartScheduleTableAbs()`.

2. Asynchronous/Hard: Define `LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION=TRUE` and `SYNC_STRATEGY=HARD`. Start using `StartScheduleTableRel()`.

3. Asynchronous/Smooth: Define `LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION=TRUE` and `SYNC_STRATEGY=SMOOTH`. Start using `StartScheduleTableRel()`.

13.5 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, because 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 or set of tasks."*

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 or set of tasks."*

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.

13.6 Global Time Synchronization

The OS currently assumes that the global time synchronization is done by the user. 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.

13.7 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
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. The first option requires an interface to the OS to indicate that a hardware counter has expired and "some work" must be done as a result. Such an interface is not standardized in AUTOSAR OS – further direction on such a scheme should be provided by the OS vendor.
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 require then an OS COUNTER object should be configured that has a MAXALLOWEDVALUE equal to the maximum number of Cycles. If Cycle/MacrotickOffset is required then an OS COUNTER object should be configured with a MAXALLOWEDVALUE 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.

Synchronisation between the OS and an external global time source is provided by the telling the OS the global time through the SyncScheduleTable() service call. This call takes a scalar parameter of GlobalTimeTickType 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. GlobalTimeTickType is at least 24-bits wide.

```
#define OSTIME(x) (GlobalTimeTickType)(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.

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, mainly as a basis for discussion with other WPs, probably to be standardized.

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.

15 Changes to Release 1

This chapter contains all major changes to the previous release. Changes made are either based on bugzilla entries or on proposals which were presented during work group meetings. Note that small changes or typos or reformatting are not listed.

15.1 Deleted SWS Items

SWS Item	***Rationale***
Chapter 4.5	Deleted completely
Chapter 7	OS260 (in chapter 7.3), OS305 (in chapter 7.7)
Chapter 8	Deleted `StartScheduleTable()`
Chapter 11	OS174 (in chapter 11.2), OS178 (in chapter 11.3)

15.2 Replaced SWS Items

SWS Item of Release 1	***replaced by SWS Item***	***Rationale***
Rational and background of chapter 7.3	New rational and background	Replaced complete description which also contains now an example how relative/absolute starts of schedule tables influences the synchronization

15.3 Changed SWS Items

SWS Item	***Rationale***
Glossary of Terms	Adapted some terms to the new specification/meaning.
Chapter 6	Updated tables to reflect new API and added missing requirements
Chapter 7	Clarified and extended synchronization of schedule tables (OS206, OS200). Changed handling of memory protection for Task/Category 2 ISR private stack/data within an OS-Application to an optional feature (OS208, OS195). Changed arrival rate handling: For Tasks the model changed from a state-based to a time budget behavior. For Category 2 ISRs two APIs were introduced to allow a save implementation. Exclude Category 1 Interrupts from some requirements (e.g. OS088) Restrict Scalability Class 3 and 4 to EXTENDED mode. Service protection makes no sense in standard mode. The interrupt locking time for Tasks/Category 2 ISRs is split into two timings: One for the time a Task/Category 2 ISR disables all interrupts and one time where only the Category 2 interrupts are disabled. Changed several requirments to fit to new configuration parameters.
Chapter 8	Improved wording of constants for schedule table status type. Added argument for `TerminateApplication()` to allow a restart. Removed 3rd argument of `SyncScheduleTable()` (now obsolete).
Chapter 12	Updated the OIL example to new attributes.

15.4 Added SWS Items

SWS Item	*Rationale*
Glossary of Terms	Added some new terms which are now covered by the SWS, e.g. Interrupt Vector Table.
Chapter 7	Added new figure to 7.2 explaining the start of a schedule table.
Chapter 8	Added new APIs: o `StartScheduleTableRel()` o `StartScheduleTableAbs()` o `DisableInterruptSource()` o `EnableInterruptSource()` Extended `GetScheduleTableStatus()`
Chapter 10	Added containers from new SWS template. These contain now also the OIL attribute and their meaning.
Chapter 11	Added some additional consistency checks (OS343, OS344, OS345).
Chapter 13	Added more explanations to 13.4 covering the migration from OSEKtime to AUTOSAR OS.