MICROSAR Classic OS

Technical Reference

Version 3.24.00

Status	Released

Document Information

History

Author	Date	Version	Remarks
Visto	2016-04-27	1.0.0	First release version
Visto	2016-05-18	1.0.1	References to hardware manuals added. Revision work
Visto	2016-06-03	1.0.2	Fix of ESCAN00089598
Visto	2016-06-20	1.1.0	List of OS internal objects added. Additional startup concept chapter added. Chapter “Memory mapping concept” reworked. Description of “generate callout stubs” feature added.
Visto	2016-07-05	1.1.1	Chapter “Memory Mapping Concept” extended. IOC notification callback concept changed. HSI of RH850 family added. HSI of Power PC family added.
Visto	2016-07-19	1.1.2	Chapter “Memory Mapping Concept” changed. Hints for shorter compile times added. Nesting behavior of OS hooks described.
Virbiv	2016-08-11	1.1.3	HSI of ARM family added.
Visto	2016-08-12	1.1.4	Chapter “Memory Mapping Concept” extended. Chapter “Clear Pending Interrupt” extended. Chapter “RH850 Special Characteristics” extended.
Virbiv	2016-08-18	1.1.5	HSI of ARM Zynq UltraScale added.
Visto	2016-08-30	1.1.6	HSI of RH850 extended.
Visto	2016-08-31	1.1.7	ORTI Debugging added. Timing Hook Macros reworked. Chapter “Memory Mapping Concept” changed. Chapter “Category 1 Interrupts” extended.
Virsso Visto	2016-09-15	1.1.8	Chapter “Interrupt Source API” extended. HSI chapter for ARM extended
Visto	2016-09-22	1.2.0	VTT OS and Dual Target Concept added. Chapter ORTI Debugging extended.
Visasl Visdhe	2016-10-14	1.3.0	Ristrictions concerning API usage before StartOS() documented. Clarification concerning forcible termination and schedule tables added. Deviations in IOC added. Notes on mixed criticality systems added. Chapter “RH850 Special Characteristics” extended.

Visto	2016-10-19	1.3.1	Chapter “Configuration of X-Signals” added. Chapter “Power PC Special Characteristics” extended. Correction of startup examples. Chapter “User include files” added. RH850 HSI extended. PPC HSI extended. Hardware Overview extended by RH850.
Visdfe	2016-11-03	1.4.0	PPC HSI extended. Chapter ORTI Debugging extended.
Vismkk	2016-11-25	1.5.0	Updated chapter Timing Hooks
Virsmn	2016-12-08	1.6.0	PPC HSI extended. Updated characteristics of VTT OS.
Visdfe Virjas Virbiv Virsso	2016-12-22	1.7.0	Updated precautions in PreStartTask. Support new Power PC Derivative: PC580003 Support IAR compiler for ARM ARM Cortex-A HSI added
Visdfe Visto	2017-01-23	1.8.0	Chapter “Memory Mapping Concept” changed. Chapter “Resulting sections” extended. Chapter “X-Signals” extended. Chapter “API Description” extended.
Visto Virsso Visdfe	2017-02-06	2.0.0	Chapter “Memory Mapping Concept” corrected. Chapter “MICROSAR Classic OS Deviations from AUTOSAR OS Specification” extended. Chapter “IOC” extended. Feature “Fast Trusted Functions” added. Chapter “Non-Trusted Functions (NTF)” changed. ARM Cortex-M Hardware overview updated. Feature “Barriers” added.
Virsmn Virbse Visdhe Visto Virsso Visasl	2017-03-22	2.1.0	Updated Hardware Overview for Power PC derivative groups (RM revisions). Chapter “MICROSAR Classic OS Deviations from AUTOSAR OS Specification” corrected. Added API OSError_GetScheduleTableStatus_ScheduleStatus Chapter “ARM Special characteristic” extended. Chapter “Cortex-R derivatives” extended. Chapter “Idle Task” extended. TI Compiler added as supported compiler for ARM. Platform POSIX added Added HSI for ARM Cortext-M
Viszfa Virsso	2017-03-31	2.2.0	Added AUTOSAR specification deviations. Changed address parameter type in periperal API functions.

Visdhe Virsmn	2017-04-11	2.3.0	Added HSI for TI AR16xx Added information for Hardware Init Core
Visces Visto Virsmn Visdhe	2017-05-10	2.4.0	Added HSI for R-Car H3. Extended chapter "Memory Mapping Concept". Added chapter "Linking of Spinlocks". Updated HSI for S32K derivatives. Added chapter for exception context manipulation
Viszfa Virsmn	2017-06-19	2.5.0	Removed ORTI tracing from Os_Init and Os_InitMemory Support new Power PC Derivative: SPC574Sxx
Visto	2017-06-06	2.6.0	Added descriptions for category 0 ISRs.
Virbiv Visces	2017-07-05	2.6.1	Chapter "ARM Special characteristic" extended. RH850 HSI extended. Updated Table 1-9 Supported RH850 Compilers. Updated Chapter 4.5.2 RH850
Visto	2017-07-17	2.7.0	Chapter "Software Stack Check" extended. Chapter "VTT OS Specifics" extended. Chapter "Initialization of Interrupt Sources" extended. Chapter "Notes on Category 1 ISRs" extended. Chapter "Notes on Category 0 ISRs" extended. Chapter "Pre-Process Linker Command Files" added. API description of "Os_Init" extended.
Visces Visdhe Virjas	2017-08-15	2.8.0	Documented support for more RH850 derivatives and compiler versions. Updated documentations regarding location of OS identifiers. Support ARM CC (5.x) compiler for ARM Cortex-M Documented support of TC39x derivative with Tasking v6.0r1p2 compiler
Virsmn	2017-08-17	2.9.0	Updated Derivative Support for PPC and RH850
Visces Visto Visrk	2017-10-25	2.10.0	New vector timing hooks OS_VTHACTIVATION_LIMIT and OS_VTH_WAITEVENT_NOWAIT, usage of vector timing hooks now also in safety systems. Chapter "Task Stack Sharing" Extended Added comments on RTE interrupt API
Visdhe Virbse	2017-11-13	2.11.0	Support GCC Linaro compiler for ARM Cortex-A/R and Cortex-M Added HighTec compiler support for PowerPC and TriCore Added MPC56xx derivatives to chapter "Hardware Overview" and "Hardware Software Interfaces" Fixed Timing Hooks API descriptions

Virsso Visto Virbse	2017-12-14	2.12.0	Support for TDA2x family derivatives Support for TriCore Aurix TC38x Added caution to chapter “Aurix Special Characteristics” Fixed descriptions in "Os generated objects"
Visbpz Visces Virsso	2018-01-11	2.13.0	Chapter “VTT OS Specifics” corrected Added RH850 F1KH hardware manual reference Chapter “Floating Point Context Extension” Updated HSI Chapter
Virsmn Virsso	2018-02-02	2.14.0	Adapted HSI – MSR Bits used by OS Added Chapter “User defined processor state.” Support for TMSLS57021x_31x derivatives
Visto	2018-03-09	2.14.1	Adapted a note in chapter “Floating Point Context Extension”
Visces	2018-03-14	2.15.0	Adapted Chapter 4.3.3 “Section Symbols” Added Chapter 2.4.8 “Unhandled Syscalls” Added Chapter 4.2.1.8 “Configuration of Interrupt Sources”
Visrk Virbse Viszfa	2018-04-03	2.16.0	Added chapter 4.10 “Preprocessing of assembler language files” Added supported compiler version for ARM Added deviation regarding spinlock deadlock detection
Virbse	2018-04-17	2.17.0	Added support for TMS570LC43x derivatives Updated chapter "4.2.4.4 ARM Special Characteristics"
Visbpz Virbse	2018-05-14	2.18.0	Added description for Interrupt Mapping support Updated the usage section of the Exception Context Manipulation chapter Added caution for GetTaskID to chapter "2.3.4 Software Stack Check"
Visbpz Virbse	2018-06-28	2.19.0	Support for CYT2Bx derivatives Extended Chapter "4.11.2 ARM Family” Added Chapter "4.12 Stack Summary" Small fix in Chapter "2.17.5 Protection Violation Handling"
Visrk Visbpz	2018-07-16	2.20.0	Improved chapter “5.2.12 Timing Hooks” in order to describe calls to timing hooks in case of forcible termination Added support for IMX8x derivatives

Virsso Virsmn	2018-08-03	2.21.00	Added support for S32x derivatives Improved chapter “1.4.3 Hardware Overview - ARM” Improved chapter “3.15.2 Floating Point Context Extension - Usage” Updated description for Pre- and PostTaskHook behaviour in case of violations.
Virsso	2018-10-17	2.22.00	Added support for TMPV770x derivatives.
Visaev Visrk	2018-11-27	2.24.00	Imroved structure of chapter “3.17 Memory Protection” to avoid duplicates. Improved the structure of chapter “5.2 Hardware Software Interface” and made it more uniform. Add description for “Init Stack”. New function to enable all interrupt sources.
Visdqk Virsso Visaev	2019-03-29	2.25.00	Added a warning to the specific characteristics of the TriCore Aurix Family. Added a limitation to the API service Os_InitialEnableInterruptSources. Corrected the entry KernelErrors in chapter “ErrorHandling”.. Added support for CYT4Bx, Xavier, Cortex-M0. Extended Chapter 2.4.3 “ARM Hardware Overview”. Extended Chapter 5.2.4 “ARM HSI”. Extended Chapter 5.5.4.1 – “X-Signal ISR Interrupt Sources”. Added new return values for Event-API-.
Visaev Virrlu	2019-04-18	2.26.00	STORY-11978, SystemApplication information, adaption of HIS overview. Updated Derivative Support for RH850. Extended Chapter 5.2.4 “ARM HSI”.
Visrk	2019-05-15	2.27.00	STORY-12408: Refinement of XSignal mechanism description in chapters 3.15.3, 4.9 and 5.5. STORY-12722: Improved description of Vector timing hook OS_VTH_SCHEDULE
Virsso Virrlu	2019-06-28	2.28.00	Added support for ExynosAuto9 derivatives. Added support for CYT4Bx (Cortex-M0) derivatives. Added FRT support for iMx8 derivatives. Support Windriver DiabData compiler for RH850.

Virsso Visror Visto Visdqk Virsmn	2019-07-09	2.29.00	Removed RTT Timer references. Fix of ESCAN00103432 Added support of Windriver Diab compiler for TriCore Aurix Added support of GHS compiler for TriCore Aurix Added support of Aurix TC33x, TC35x, TC36x, TC37x Provide Os_BarrierSynchronize as ASIL-D feature. Added new linker labels (*_LIMIT).
Visaev Visdqk	2019-08-21	2.30.00	Added description of the new API function Os_GetCoreStartState
Virrlu, Visbpz Virsmn	2019-09-19	2.31.00	Added description for Interrupt Mapping support (RH850). Extend chapter 5.2.2 “RH850 HSI”. Added support for PPC S32R29x derivatives. Extended chapter 5.2.4 “ARM Family HSI”. Documented support of Arm Family with Tasking v6.2r2p2 Arm compiler. Added additional information for HRT solution and S32K derivative group.
Visbpz Virrlu Virsmn	2019-10-16	2.32.00	Extended chapter 5.2.4 “ARM HSI”. Updated Derivative Support for RH850. Extend chapter 5.2.2 “RH850 HSI”. Extended description for section access rights.
Virsmn Virrlu Visdri	2019-11-04	2.33.00	Update for S32K derivatives HSI description. PPC HSI extended. Updated compiler and derivative support for PPC. Add HSI for TI Jacinto7 TDA4x.
Virsmn Visrk	2019-11-22	2.34.00	Updated description for Timing Hook usage. Fix of ESCAN00105025 by documentation of interrupt state in Shutdown- and ProtectionHook.
Virleh Visrk Vismun Visbpz	2019-12-20	2.35.00	Added documentation of time slice scheduling. Added FRT support for ExynosAuto9 derivatives. Fix of ESCAN00104931.
Virrlu Visto Visror	2020-01-28	2.36.00	Updated Derivative Support for RH850. Updated derivative support for TI Jacinto. Fixed ESCAN00105308: Wrong OS API return value (ReleaseResource).

Vismaa Virleh Vismun Virjas	2020-03-09	2.37.00	Updated Derivative Support ARM Added HSI for S32K2TV/S32K3xx Added additional information to the RTE Interrupt API. Updated support for S32x derivatives. Updated derivative support and HSI for PowerPC HSM.
Virleh Visror	2020-03-30	3.00.00	Added IOC call context documentation. Correction of user API return values. Splitted TechnicalReference in Core and HAL part.
Virsmn	2020-04-21	3.01.00	Added deviation for Pre- and PostTaskHook.
Virsmn	2020-05-26	3.02.00	Added new OS API Os_GetExceptionAddress(). Updated description for Exception Context reading and manipulation.
Virsmn	2020-06-23	3.03.00	Clarification for MPU configuration regarding stacks.
Visror	2020-07-10	3.04.00	Update for feature Fast Trusted Functions.
Virsmn	2020-08-21	3.05.00	Added deviation for ChainTask() API.
Virsmn	2020-12-04	3.06.00	Removed Author identity.
Virsmn	2020-12-30	3.07.00	Update for feature memory protection and trusted function calls.
Virbse, Visror, Virsmn	2021-02-25	3.08.00	Added StackUsage API for NonTrusted functions. Added behavior of StackUsage APIs for VTT. Extended error handling information. Added new OS service Os_InitInterruptOnly.
Visdqk, Virsmn	2021-03-18	3.09.00	Added additional information for Pre-Start tasks.
Virsmn	2021-04-24	3.10.00	Fixed wording related to diagnostic coverage.
Visdqk	2021-07-12	3.11.00	Added call context ISR1 to interrupt API.
Virbse	2022-01-04	3.12.00	Updates regarding OsIocDataTypeRef.
Virbse	2022-03-24	3.13.00	Updated and added chapters about ARTI support.
Visrk	2022-09-13	3.14.00	Name change: “MICROSAR” to “MICROSAR Classic”, improved description of configuration of exceptions.
Sreif, Twurm, Virleh	2022-12-08	3.15.00	Added service Os_EnableInterruptsPreStart. Added new error code E_OS_SYS_TIMEOUT. Describe deadlock possibility. Describe OS_APPMODE_ANY.
Mwohnhaas	2023-02-16	3.16.00	Added feature Counter Algorithm.

Visrk, Sreif	2023-04-13	3.17.00	Sorted glossary alphabetically and added descriptions of hardware operation modes. Described spinlock release on non-forcible thread termination.
Rleinauer, Mwohnhaas	2023-05-15	3.18.00	Extended the description of the behavior when a protection error occurs (chapter 2.13). Added description of the behavior of forcible termination in non-trusted function calls.
Sreif	2023-07-18	3.19.00	Documented standard deviation in GetAlarmBase.
Virsmn	2023-08-10	3.20.00	Added debug support for PanicHook.
Hsimon, Asaleh, Rleinauer, Sreif	2023-09-27	3.21.00	Added API Os_GetInitHookStackUsage. Added APIs ActivateTaskAsyn and SetEventAsyn. Fixed ESCAN00114769: Wrong description of service protection error handling. Fixed ESCAN00115291: 16 bit free running timers can not be used to emulate a periodic interrupt timer.
Sreif, Rleinauer	2023-12-15	3.22.00	Added configuration containers OsTaskPeriod and OsIsrPeriod. Added description of the VTT OS specific definition OS_VTT_TICKS_PER_NOP_INSTRUCTION.
Mwohnhaas	2024-03-15	3.23.00	Extend description of shared data sections.
Visdqk, Sreif	2024-07-25	3.24.00	Added chapter Cybersecurity. Added MMU support. Documented compiler flag needed for VTT.

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of Operating System Document ID 034: AUTOSAR_SWS_OS	4.2.1
[2]	OSEK/VDX	OSEK/VDX Operating System Specification This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)	2.2.3
[3]	OSEK/VDX	OSEK RunTime Interface (ORTI) Part A: Language Specification. This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)	2.2
[4]	OSEK/VDX	OSEK Run Time Interface (ORTI) Part B: OSEK Objects and Attributes This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)	2.2
[5]	Lauterbach	ORTI Representation of SMP Systems (ORTI 2.3)	4
[6]	Vector	vVIRTUALtarget Technical Reference	See delivery information
[7]	Vector	Startup with Vector and vVIRTUALtarget	See delivery information
[8]	Vector	MICROSAR Classic VStdLib Technical Reference TechnicalReference_VstdLib_GenericAsr.pdf	See delivery information
[9]	Vector	MICROSAR Classic OS HAL Technical Reference	See delivery information
[10]	Vector	MICROSAR SafetyManual	See delivery information
[11]	AUTOSAR	Specification of Operating System Document ID 34: AUTOSAR_SWS_OS	R20-11
[12]	AUTOSAR	Specification of Operating System Document ID 34: AUTOSAR_SWS_OS	R19-11

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

1 Introduction ... 29
- 1.1 Architecture Overview ... 29
- 1.2 Abstract ... 30
- 1.3 Characteristics ... 30
- 1.4 VTT OS ... 31
  - 1.4.1 Characteristics of VTT OS ... 31

2 Functional Description ... 32
- 2.1 General ... 32
- 2.2 MICROSAR Classic OS Deviations from AUTOSAR OS Specification ... 32
  - 2.2.1 Generic Deviation for API Functions ... 32
  - 2.2.2 Trusted Function API Deviations ... 32
  - 2.2.3 Service Protection Deviation ... 33
  - 2.2.4 Code Protection ... 33
  - 2.2.5 SyncScheduleTable API Deviation ... 33
  - 2.2.6 CheckTask/ISRMemoryAccess API Deviation ... 33
  - 2.2.7 Interrupt API Deviation ... 34
  - 2.2.8 Cross Core Getter APIs ... 34
  - 2.2.9 Return value upon stack violation ... 35
  - 2.2.10 Handling of OS internal errors ... 35
  - 2.2.11 Forcible Termination of Applications ... 36
  - 2.2.12 OS Configuration ... 36
  - 2.2.13 Spinlocks ... 37
  - 2.2.14 Errors within the PreTaskHook() and PostTaskHook() ... 37
  - 2.2.15 ChainTask API Deviation ... 38
  - 2.2.16 GetAlarmBase API Deviation ... 39
  - 2.2.17 ARTI OS Deviations ... 39
- 2.3 Stack Concept ... 40
  - 2.3.1 Task Stack Sharing ... 42
    - 2.3.1.1 Description ... 42
    - 2.3.1.2 Activation ... 42
    - 2.3.1.3 Usage ... 43
  - 2.3.2 ISR Stack Sharing ... 43
    - 2.3.2.1 Description ... 43
    - 2.3.2.2 Activation ... 43
    - 2.3.2.3 Usage ... 43
  - 2.3.3 Stack Check Strategy ... 43
  - 2.3.4 Software Stack Check ... 44
    - 2.3.4.1 Description ... 44

2.3.4.2 Activation ... 44
2.3.4.3 Usage ... 45
2.3.5 Stack Monitoring by MPU / MMU ... 45
2.3.5.1 Description ... 45
2.3.5.2 Activation ... 46
2.3.5.3 Usage ... 46
2.3.6 Stack Usage Measurement ... 47
2.3.6.1 Description ... 47
2.3.6.2 Activation ... 47
2.3.6.3 Usage ... 47
2.4 Interrupt Concept ... 47
2.4.1 Interrupt Handling API ... 47
2.4.2 Interrupt Levels ... 47
2.4.3 Interrupt Vector Table ... 48
2.4.4 Nesting of Category 2 Interrupts ... 48
2.4.4.1 Description ... 48
2.4.4.2 Activation ... 49
2.4.5 Category 1 Interrupts ... 49
2.4.5.1 Implementation of Category 1 ISRs ... 49
2.4.5.2 Nesting of Category 1 ISRs ... 49
2.4.5.3 Category 1 ISRs before StartOS ... 49
2.4.5.4 Notes on Category 1 ISRs ... 50
2.4.6 Initialization of Interrupt Sources ... 50
2.4.7 Unhandled Interrupts ... 52
2.4.8 Unhandled Syscalls ... 52
2.5 Exception Concept ... 53
2.5.1 Exception Vector Table ... 53
2.5.2 Unhandled Exceptions ... 53
2.5.3 Configuration ... 53
2.5.4 Defining an own exception handler ... 54
2.6 Timer Concept ... 55
2.6.1 Description ... 55
2.6.2 Activation ... 55
2.6.3 Usage ... 55
2.6.4 Dependencies ... 55
2.7 Periodical Interrupt Timer (PIT) ... 56
2.7.1 Description ... 56
2.7.2 Activation ... 56
2.7.3 Driver Configuration ... 56
2.8 High Resolution Timer (HRT) ... 57
2.8.1 Description ... 57

2.8.2 Activation ... 57
2.9 PIT versus HRT ... 57
2.10 Startup Concept ... 58
2.11 Single Core Startup ... 59
2.11.1 Single Core Derivatives ... 59
2.11.2 Multi Core Derivatives ... 59
2.11.2.1 Examples for SC1 / SC2 Systems ... 59
2.11.2.2 Examples for SC3 / SC4 Systems ... 60
2.12 Multi Core Startup ... 62
2.12.1 Example for SC1 / SC2 Systems ... 62
2.12.2 Examples for SC3 / SC4 systems ... 63
2.12.2.1 Only with AUTOSAR Cores ... 63
2.12.2.2 Mixed Core System ... 63
2.13 Error Handling ... 65
2.13.1 Service Protection Error Handling ... 66
2.13.1.1 Access Check ... 66
2.13.1.2 Accessibility Check ... 66
2.13.1.3 Owner Check ... 66
2.13.1.4 Interrupt API Check: ... 66
2.14 Error Reporting ... 68
2.14.1 Extension of Service IDs ... 68
2.14.2 Extension of Error Codes ... 68
2.14.3 Detailed Error Codes ... 69
2.15 Multi Core Concepts ... 70
2.15.1 Scheduling and Dispatching ... 70
2.15.2 Multi Core Data Concepts ... 70
2.15.3 X-Signals ... 70
2.15.4 Master / Slave Core ... 70
2.15.5 Hardware Init Core ... 70
2.15.6 Startup of a Multi Core System ... 70
2.15.7 Spinlocks ... 71
2.15.7.1 Linking of Spinlocks ... 71
2.15.8 Cache ... 71
2.15.9 Shutdown ... 71
2.15.9.1 Shutdown of one Core ... 71
2.15.9.2 Shutdown of all Cores ... 71
2.15.9.3 Shutdown during Protection Violation ... 71
2.16 Debugging Concepts ... 72
2.16.1 Description ... 72
2.16.2 Activation ... 72
2.16.3 ORTI Debugging ... 72

2.16.4 ARTI Debugging 74
2.16.4.1 ARTI Hooks 74
2.17 Memory Protection 75
2.17.1 Usage of the MPUs / MMUs 75
2.17.2 Configuration Aspects 76
2.17.2.1 Static Memory Regions 76
2.17.2.2 Dynamic Memory Regions 77
2.17.2.3 Freedom from Interference 77
2.17.3 Stack Monitoring 77
2.17.4 Protection Violation Handling 78
2.17.5 Optimized / Fast MPU Handling 78
2.17.6 Recommended Configuration 79
2.18 Memory Access Checks 80
2.18.1 Description 80
2.18.2 Activation 80
2.18.3 Usage 80
2.18.4 Dependencies 80
2.19 Timing Protection Concept 80
2.19.1 Description 80
2.19.2 Activation 82
2.19.3 Usage 82
2.20 IOC 83
2.20.1 Description 83
2.20.2 Unqeued (Last Is Best) Communication 83
2.20.2.1 1:1 Communication Variant 83
2.20.2.2 N:1 Communication Variant 83
2.20.2.3 N:M Communication Variant 84
2.20.3 Queued Communication 84
2.20.4 Notification 84
2.20.5 Particularities 84
2.20.5.1 N:1 Queued Communication 84
2.20.5.2 IOC Spinlocks 85
2.20.5.3 Notification 85
2.20.5.4 Complex Data Types 86
2.21 Trusted OS Applications 87
2.21.1 Trusted OS Applications with Memory Protection 87
2.21.1.1 Description 87
2.21.1.2 Activation 87
2.21.1.3 Dependencies 87
2.21.2 Trusted Functions 88
2.22 OS Hooks 89

2.22.1 Runtime Context ... 89
2.22.2 Nesting behavior ... 89
2.22.3 Hints ... 89

3 Vector Specific OS Features ... 91
3.1 Optimized Spinlocks ... 91
3.1.1 Description ... 91
3.1.2 Activation ... 91
3.1.3 Usage ... 91
3.2 Barriers ... 92
3.2.1 Description ... 92
3.2.2 Activation ... 92
3.2.3 Usage ... 92
3.3 Peripheral Access API ... 94
3.3.1 Description ... 94
3.3.2 Activation ... 94
3.3.3 Usage ... 94
3.3.4 Dependencies ... 94
3.3.5 Alternatives ... 94
3.3.6 Common Use Cases ... 94
3.4 Trusted Function Call Stubs ... 95
3.4.1 Description ... 95
3.4.2 Activation ... 95
3.4.3 Usage ... 95
3.4.4 Dependencies ... 95
3.5 Non-Trusted Functions (NTF) ... 96
3.5.1 Description ... 96
3.5.2 Activation ... 96
3.5.3 Usage ... 96
3.5.4 Dependencies ... 97
3.6 Fast Trusted Functions ... 97
3.6.1 Description ... 97
3.6.2 Activation ... 97
3.6.3 Usage ... 97
3.6.4 Dependencies ... 97
3.7 Interrupt Source API ... 98
3.7.1 Description ... 98
3.8 Pre-Start Task ... 99
3.8.1 Description ... 99
3.8.2 Activation ... 99
3.8.3 Usage ... 99

3.8.4 Dependencies ... 100
3.9 X-Signals ... 101
3.9.1 Description ... 101
3.9.1.1 Notes on Synchronous X-Signals ... 103
3.9.1.2 Notes on Mixed Criticality Systems ... 104
3.9.2 Activation ... 104
3.10 Timing Hooks ... 105
3.10.1 Description ... 105
3.10.2 Activation ... 106
3.10.3 Usage ... 106
3.11 Kernel Panic ... 107
3.12 Generate callout stubs ... 108
3.12.1 Description ... 108
3.12.2 Activation ... 108
3.12.3 Usage ... 108
3.13 Exception Context Reading and Manipulation ... 109
3.13.1 Description ... 109
3.13.2 Usage ... 109
3.14 Category 0 Interrupts ... 110
3.14.1 Description ... 110
3.14.2 Usage ... 110
3.14.2.1 Implement Category 0 ISRs ... 110
3.14.2.2 Nesting of Category 0 ISRs ... 111
3.14.2.3 Category 0 ISRs before StartOS ... 111
3.14.2.4 Locations where category 0 ISRs are locked ... 111
3.14.3 Notes on Category 0 ISRs ... 112
3.15 Floating Point Context Extension ... 114
3.15.1 Description ... 114
3.15.2 Usage ... 114
3.16 User defined processor state ... 115
3.16.1 Description ... 115
3.16.2 Usage ... 115
3.17 Interrupt Mapping ... 115
3.17.1 Description ... 115
3.17.2 Usage ... 115
3.18 Time Slice Scheduling ... 115
3.18.1 Description ... 115
3.18.2 Activation ... 116
3.18.3 Usage ... 116
3.18.3.1 Usage with OS resources ... 116
3.19 Interrupt Only Use Case ... 118

3.19.1 Description ... 118
3.19.2 Activation ... 118
3.19.3 Usage ... 118
3.20 OS_APPMODE_ANY ... 119
3.20.1 Description ... 119
3.20.2 Usage ... 119
3.21 Counter Algorithm ... 119
3.21.1 Description ... 119
3.21.2 Activation ... 119
3.22 Asynchronous operation of API functions to activate tasks and set events ... 120
3.22.1 Description ... 120
3.22.2 Activation ... 120
3.22.3 Usage ... 120
3.23 Configuration of Task and ISR Periods ... 120

4 Integration ... 121
4.1 Safety Manual ... 121
4.2 Compiler Optimization Assumptions ... 121
4.2.1 Compile Time ... 121
4.3 Hardware Software Interfaces (HSI) ... 121
4.4 Memory Mapping Concept ... 122
4.4.1 Provided MemMap Section Specifiers ... 122
4.4.1.1 Usage of MemMap Macros ... 125
4.4.1.2 Resulting sections ... 125
4.4.1.3 Access Rights to Variable Sections ... 131
4.4.1.4 Access Rights to Shared Data Sections ... 133
4.4.2 Link Sections ... 134
4.4.2.1 Pre-Process Linker Command Files ... 134
4.4.2.2 Simple Linker Defines ... 135
4.4.2.3 Hierachical Linker Defines ... 135
4.4.2.4 Selecting OS constants ... 135
4.4.2.5 Selecting OS variables ... 136
4.4.2.6 Selecting special OS Variables ... 137
4.4.2.7 Selecting User Constant Sections ... 138
4.4.2.8 Selecting User Variable Sections ... 139
4.4.3 Section Symbols ... 141
4.4.3.1 Aggregation of Data Sections ... 141
4.5 Static Code Analysis ... 142
4.6 Configuration of X-Signals ... 143
4.6.1 VTT OS ... 143
4.7 OS generated objects ... 144

4.7.1 System Application ... 144
4.7.2 Idle Task ... 144
4.7.3 Timer ISR ... 145
4.7.4 System Timer Counter ... 145
4.7.5 Timing Protection Counter ... 145
4.7.6 Timing protection ISR ... 145
4.7.7 Resource Scheduler ... 146
4.7.8 X-Signal ISR ... 146
4.7.9 IOC Spinlocks ... 146
4.8 VTT OS Specifics ... 147
4.8.1 Configuration ... 147
4.8.2 CANoe Interface ... 147
4.8.2.1 Idle Task behavior with VTT OS ... 147
4.8.3 Timing Adjustment ... 147
4.8.4 Compilation ... 148
4.9 User include files ... 148
4.10 Preprocessing of assembler language files ... 149
4.11 Stack Summary ... 150

5 API Description ... 151
5.1 Specified OS services ... 151
5.1.1 StartCore ... 151
5.1.2 StartNonAutosarCore ... 152
5.1.3 GetCoreID ... 153
5.1.4 GetNumberOfActivatedCores ... 154
5.1.5 GetActiveApplicationMode ... 155
5.1.6 StartOS ... 156
5.1.7 ShutdownOS ... 157
5.1.8 ShutdownAllCores ... 158
5.1.9 ControlIdle ... 159
5.1.10 GetSpinlock ... 160
5.1.11 ReleaseSpinlock ... 161
5.1.12 TryToGetSpinlock ... 162
5.1.13 DisableAllInterrupts ... 163
5.1.14 EnableAllInterrupts ... 164
5.1.15 SuspendAllInterrupts ... 165
5.1.16 ResumeAllInterrupts ... 166
5.1.17 SuspendOSInterrupts ... 167
5.1.18 ResumeOSInterrupts ... 168
5.1.19 ActivateTask ... 169
5.1.20 TerminateTask ... 170

5.1.21 ChainTask ... 171
5.1.22 Schedule ... 172
5.1.23 GetTaskID ... 173
5.1.24 GetTaskState ... 174
5.1.25 GetISRID ... 175
5.1.26 SetEvent ... 176
5.1.27 ClearEvent ... 177
5.1.28 GetEvent ... 178
5.1.29 WaitEvent ... 179
5.1.30 IncrementCounter ... 180
5.1.31 GetCounterValue ... 181
5.1.32 GetElapsedValue ... 182
5.1.33 GetAlarmBase ... 183
5.1.34 GetAlarm ... 184
5.1.35 SetRelAlarm ... 185
5.1.36 SetAbsAlarm ... 186
5.1.37 CancelAlarm ... 187
5.1.38 GetResource ... 188
5.1.39 ReleaseResource ... 189
5.1.40 StartScheduleTableRel ... 190
5.1.41 StartScheduleTableAbs ... 191
5.1.42 StopScheduleTable ... 192
5.1.43 NextScheduleTable ... 193
5.1.44 GetScheduleTableStatus ... 194
5.1.45 StartScheduleTableSynchron ... 195
5.1.46 SyncScheduleTable ... 196
5.1.47 SetScheduleTableAsync ... 197
5.1.48 GetApplicationID ... 198
5.1.49 GetCurrentApplicationID ... 199
5.1.50 GetApplicationState ... 200
5.1.51 CheckObjectAccess ... 201
5.1.52 CheckObjectOwnership ... 202
5.1.53 AllowAccess ... 203
5.1.54 TerminateApplication ... 204
5.1.55 CallTrustedFunction ... 205
5.1.56 Check Task Memory Access ... 206
5.1.57 Check ISR Memory Access ... 207
5.1.58 OSErrorGetServiceId ... 208
5.1.59 OSError_Os_DisableInterruptSource_ISRID ... 209
5.1.60 OSError_Os_EnableInterruptSource_ISRID ... 209
5.1.61 OSError_Os_EnableInterruptSource_ClearPending ... 210

5.1.62 OSError_Os_ClearPendingInterrupt_ISRID 210
5.1.63 OSError_Os_IsInterruptSourceEnabled_ISRID 211
5.1.64 OSError_Os_IsInterruptSourceEnabled_IsEnabled 211
5.1.65 OSError_Os_IsInterruptPending_ISRID 212
5.1.66 OSError_Os_IsInterruptPending_IsPending 212
5.1.67 OSError_CallTrustedFunction_FunctionIndex 213
5.1.68 OSError_CallTrustedFunction_FunctionParams 213
5.1.69 OSError_CallFastTrustedFunction_FunctionIndex 214
5.1.70 OSError_CallFastTrustedFunction_FunctionParams 214
5.1.71 OSError_CallNonTrustedFunction_FunctionIndex 215
5.1.72 OSError_CallNonTrustedFunction_FunctionParams 215
5.1.73 OSError_StartScheduleTableRel_ScheduleTableID 216
5.1.74 OSError_StartScheduleTableRel_Offset 216
5.1.75 OSError_StartScheduleTableAbs_ScheduleTableID 217
5.1.76 OSError_StartScheduleTableAbs_Start 217
5.1.77 OSError_StopScheduleTable_ScheduleTableID 218
5.1.78 OSError_NextScheduleTable_ScheduleTableID_From 218
5.1.79 OSError_NextScheduleTable_ScheduleTableID_To 219
5.1.80 OSError_StartScheduleTableSynchron_ScheduleTableID 219
5.1.81 OSError_SyncScheduleTable_ScheduleTableID 220
5.1.82 OSError_SyncScheduleTable_Value 220
5.1.83 OSError_SetScheduleTableAsync_ScheduleTableID 221
5.1.84 OSError_GetScheduleTableStatus_ScheduleTableID 221
5.1.85 OSError_GetScheduleTableStatus_ScheduleStatus 222
5.1.86 OSError_IncrementCounter_CounterID 222
5.1.87 OSError_GetCounterValue_CounterID 223
5.1.88 OSError_GetCounterValue_Value 224
5.1.89 OSError_GetElapsedValue_CounterID 224
5.1.90 OSError_GetElapsedValue_Value 225
5.1.91 OSError_GetElapsedValue_ElapsedValue 225
5.1.92 OSError_TerminateApplication_Application 226
5.1.93 OSError_TerminateApplication_RestartOption 226
5.1.94 OSError_GetApplicationState_Application 227
5.1.95 OSError_GetApplicationState_Value 227
5.1.96 OSError_GetSpinlock_SpinlockId 228
5.1.97 OSError_ReleaseSpinlock_SpinlockId 228
5.1.98 OSError_TryToGetSpinlock_SpinlockId 229
5.1.99 OSError_TryToGetSpinlock_Success 229
5.1.100 OSError_ControlIdle_CoreID 230
5.1.101 OSError_Os_GetExceptionContext_Context 230
5.1.102 OSError_Os_SetExceptionContext_Context 231

5.1.103 OSError_ControlIdle_IdleMode ... 231
5.1.104 OSError_IocSend_IN ... 232
5.1.105 OSError_IocWrite_IN ... 232
5.1.106 OSError_IocSendGroup_IN ... 233
5.1.107 OSError_IocWriteGroup_IN ... 233
5.1.108 OSError_IocReceive_OUT ... 234
5.1.109 OSError_IocRead_OUT ... 234
5.1.110 OSError_IocReceiveGroup_OUT ... 235
5.1.111 OSError_IocReadGroup_OUT ... 235
5.1.112 OSError_StartOS_Mode ... 236
5.1.113 OSError_ActivateTask_TaskID ... 236
5.1.114 OSError_ChainTask_TaskID ... 237
5.1.115 OSError_GetTaskID_TaskID ... 237
5.1.116 OSError_GetTaskState_TaskID ... 238
5.1.117 OSError_GetTaskState_State ... 238
5.1.118 OSError_SetEvent_TaskID ... 239
5.1.119 OSError_SetEvent_Mask ... 239
5.1.120 OSError_ClearEvent_Mask ... 240
5.1.121 OSError_GetEvent_TaskID ... 240
5.1.122 OSError_GetEvent_Mask ... 241
5.1.123 OSError_WaitEvent_Mask ... 241
5.1.124 OSError_GetAlarmBase_AlarmID ... 242
5.1.125 OSError_GetAlarmBase_Info ... 242
5.1.126 OSError_GetAlarm_AlarmID ... 243
5.1.127 OSError_GetAlarm_Tick ... 243
5.1.128 OSError_SetRelAlarm_AlarmID ... 244
5.1.129 OSError_SetRelAlarm_increment ... 244
5.1.130 OSError_SetRelAlarm_cycle ... 245
5.1.131 OSError_SetAbsAlarm_AlarmID ... 245
5.1.132 OSError_SetAbsAlarm_start ... 246
5.1.133 OSError_SetAbsAlarm_cycle ... 246
5.1.134 OSError_CancelAlarm_AlarmID ... 247
5.1.135 OSError_GetResource_ResID ... 247
5.1.136 OSError_ReleaseResource_ResID ... 248
5.1.137 OSError_Os_GetUnhandledIrq_InterruptSource ... 248
5.1.138 OSError_Os_GetUnhandledExc_ExceptionSource ... 249
5.1.139 OSError_BarrierSynchronize_BarrierID ... 249
5.1.140 OSError_ActivateTaskAsyn_TaskID ... 250
5.1.141 OSError_SetEventAsyn_TaskID ... 250
5.1.142 OSError_SetEventAsyn_Mask ... 251
5.2 Additional OS services ... 251

5.2.1 Os_GetVersionInfo ... 251
5.2.2 Peripheral Access API ... 253
5.2.2.1 Read Functions ... 253
5.2.2.2 Write Functions ... 255
5.2.2.3 Bitmask Functions ... 257
5.2.3 Pre-Start Task ... 259
5.2.4 Non-Trusted Functions (NTF) ... 260
5.2.5 Fast Trusted Functions ... 261
5.2.6 Interrupt Source API ... 262
5.2.6.1 Disable Interrupt Source ... 262
5.2.6.2 Enable Interrupt Source ... 263
5.2.6.3 Clear Pending Interrupt ... 264
5.2.6.4 Check Interrupt Source Enabled ... 265
5.2.6.5 Check Interrupt Pending ... 266
5.2.6.6 Initial Enable Interrupt Sources ... 267
5.2.7 Detailed Error API ... 268
5.2.7.1 Get detailed Error ... 268
5.2.7.2 Unhandled Interrupt Requests ... 269
5.2.7.3 Unhandled Exception Requests ... 270
5.2.7.4 Get Exception Address ... 271
5.2.8 Stack Usage API ... 272
5.2.9 RTE Interrupt API ... 273
5.2.10 Time Conversion Macros ... 274
5.2.10.1 Convert from Time into Counter Ticks ... 274
5.2.10.2 Convert from Counter Ticks into Time ... 274
5.2.11 OS Initialization ... 275
5.2.12 Timing Hooks ... 277
5.2.12.1 Timing Hooks for Activation and Termination ... 277
5.2.12.1.1 Task Activation ... 277
5.2.12.1.2 Task Activation Exeeding Limit ... 278
5.2.12.1.3 Set Event ... 278
5.2.12.1.4 Wait Event Not Waiting ... 279
5.2.12.1.5 Timing Hook for Context Switch ... 280
5.2.12.1.6 Forcible Termination ... 281
5.2.12.2 Timing Hooks for Locking Purposes ... 281
5.2.12.2.1 Get Resource ... 281
5.2.12.2.2 Release Resource ... 282
5.2.12.2.3 Request Spinlock ... 283
5.2.12.2.4 Request Internal Spinlock ... 283
5.2.12.2.5 Get Spinlock ... 284
5.2.12.2.6 Get Internal Spinlock ... 284

5.2.12.2.7 Release Spinlock 285
5.2.12.2.8 Release Internal Spinlock 285
5.2.12.2.9 Disable Interrupts.................................... 286
5.2.12.2.10 Enable Interrupts 287
5.2.13 PanicHook 288
5.2.14 Barriers 289
5.2.15 Exception Context Manipulation 290
5.2.15.1 Os_GetExceptionContext.. 290
5.2.15.2 Os_SetExceptionContext.. 291
5.2.16 Os_GetCoreStartState 292
5.2.17 OS_EnableInterruptsPreStart... 293
5.2.18 ActivateTaskAsyn 294
5.2.19 SetEventAsyn 295

6 Configuration 297

7 Cybersecurity.. 298

8 Glossary 299

9 Contact 301

Illustrations

Figure 1-1 AUTOSAR Architecture Overview 29
Figure 2-1 Stack Safety Gap 46
Figure 2-2 Interrupt Lock Levels 48
Figure 2-3 API functions during startup 58
Figure 2-4 MICROSAR Classic OS Detailed Error Code 69
Figure 2-5 N:1 Multiple Sender Queues 85
Figure 3-1 Barriers 93
Figure 3-2 X-Signal 101
Figure 3-3 Usage of manipulating exception context 109

Tables

Table 1-1 MICROSAR Classic OS Characteristics 30
Table 1-2 VTT OS characteristics 31
Table 2-1 MICROSAR Classic OS Stack Types 42
Table 2-2 Stack Check Patterns 44
Table 2-3 Configuration attributes for exceptions 54
Table 2-4 PIT versus HRT 57
Table 2-5 Types of OS Errors 65
Table 2-6 Extension of Error Codes 69
Table 2-7 Linking of spinlocks 71
Table 2-8 Recommended Configuration MPU Access Rights 79
Table 3-1 Differences of OS and Optimized Spinlocks 91
Table 3-2 Comparison between Synchronous and Asynchronous X-Signal 102
Table 3-3 Priority of X-Signal receiver ISR 103
Table 4-1 Provided MemMap Section Specifiers 124
Table 4-2 MemMap Code Sections Descriptions 125
Table 4-3 MemMap Callout Code Sections Descriptions 125
Table 4-4 MemMap Const Sections Descriptions 126
Table 4-5 MemMap Variable Sections Descriptions 129
Table 4-6 MemMap Variable Stack Sections Descriptions 130
Table 4-7 Recommended Section Access Rights 132
Table 4-8 Recommended Shared Data Section Access Rights 133
Table 4-9 Recommended Shared Data Section Core Access Rights 133
Table 4-10 List of Generated Linker Command Files 134
Table 4-11 OS constants linker define group 135
Table 4-12 OS variables linker define group 136
Table 4-13 OS Barriers and Core status linker define group 137
Table 4-14 User constants linker define group 138
Table 4-15 User variables linker define group 139
Table 5-1 StartCore 151
Table 5-2 StartNonAutosarCore 152
Table 5-3 GetCoreID 153
Table 5-4 GetNumberOfActivatedCores 154
Table 5-5 GetActiveApplicationMode 155
Table 5-6 StartOS 156
Table 5-7 ShutdownOS 157
Table 5-8 ShutdownAllCores 158
Table 5-9 ControlIdle 159
Table 5-10 GetSpinlock 160
Table 5-11 ReleaseSpinlock 161

Table 5-12	TryToGetSpinlock	162
Table 5-13	DisableAllInterrupts	163
Table 5-14	EnableAllInterrupts	164
Table 5-15	SuspendAllInterrupts	165
Table 5-16	ResumeAllInterrupts	166
Table 5-17	SuspendOSInterrupts	167
Table 5-18	ResumeOSInterrupts	168
Table 5-19	ActivateTask	169
Table 5-20	TerminateTask	170
Table 5-21	ChainTask	171
Table 5-22	Schedule	172
Table 5-23	GetTaskID	173
Table 5-24	GetTaskState	174
Table 5-25	GetISRID	175
Table 5-26	SetEvent	176
Table 5-27	ClearEvent	177
Table 5-28	GetEvent	178
Table 5-29	WaitEvent	179
Table 5-30	IncrementCounter	180
Table 5-31	GetCounterValue	181
Table 5-32	GetElapsedValue	182
Table 5-33	GetAlarmBase	183
Table 5-34	GetAlarm	184
Table 5-35	SetRelAlarm	185
Table 5-36	SetAbsAlarm	186
Table 5-37	CancelAlarm	187
Table 5-38	GetResource	188
Table 5-39	ReleaseResource	189
Table 5-40	StartScheduleTableRel	190
Table 5-41	StartScheduleTableAbs	191
Table 5-42	StopScheduleTable	192
Table 5-43	NextScheduleTable	193
Table 5-44	GetScheduleTableStatus	194
Table 5-45	StartScheduleTableSynchron	195
Table 5-46	SyncScheduleTable	196
Table 5-47	SetScheduleTableAsync	197
Table 5-48	GetApplicationID	198
Table 5-49	GetCurrentApplicationID	199
Table 5-50	GetApplicationState	200
Table 5-51	CheckObjectAccess	201
Table 5-52	CheckObjectOwnership	202
Table 5-53	AllowAccess	203
Table 5-54	TerminateApplication	204
Table 5-55	CallTrustedFunction	205
Table 5-56	API Service CheckTaskMemoryAccess	206
Table 5-57	API Service CheckISRMemoryAccess	207
Table 5-58	OSErrorGetServiceId	208
Table 5-59	OSError_Os_DisableInterruptSource_ISRID	209
Table 5-60	OSError_Os_EnableInterruptSource_ISRID	209
Table 5-61	OSError_Os_EnableInterruptSource_ClearPending	210
Table 5-62	OSError_Os_ClearPendingInterrupt_ISRID	210
Table 5-63	OSError_Os_IsInterruptSourceEnabled_ISRID	211
Table 5-64	OSError_Os_IsInterruptSourceEnabled_IsEnabled	211
Table 5-65	OSError_Os_IsInterruptPending_ISRID	212

Table 5-66 OSError_Os_IsInterruptPending_IsPending 212
Table 5-67 OSError_CallTrustedFunction_FunctionIndex 213
Table 5-68 OSError_CallTrustedFunction_FunctionParams 213
Table 5-69 OSError_CallFastTrustedFunction_FunctionIndex 214
Table 5-70 OSError_CallFastTrustedFunction_FunctionParams 214
Table 5-71 OSError_CallNonTrustedFunction_FunctionParams 215
Table 5-72 OSError_StartScheduleTableRel_ScheduleTableID 216
Table 5-73 OSError_StartScheduleTableRel_Offset 216
Table 5-74 OSError_StartScheduleTableAbs_ScheduleTableID 217
Table 5-75 OSError_StartScheduleTableAbs_Start 217
Table 5-76 OSError_StopScheduleTable_ScheduleTableID 218
Table 5-77 OSError_NextScheduleTable_ScheduleTableID_From 218
Table 5-78 OSError_SetScheduleTableAsync_ScheduleTableID 221
Table 5-79 OSError_GetScheduleTableStatus_ScheduleTableID 221
Table 5-80 OSError_GetScheduleTableStatus_ScheduleStatus 222
Table 5-81 OSError_IncrementCounter_CounterID 222
Table 5-82 OSError_GetCounterValue_CounterID 223
Table 5-83 OSError_GetCounterValue_Value 224
Table 5-84 OSError_GetElapsedValue_CounterID 224
Table 5-85 OSError_TerminateApplication_Application 226
Table 5-86 OSError_TerminateApplication_RestartOption 226
Table 5-87 OSError_GetApplicationState_Application 227
Table 5-88 OSError_GetApplicationState_Value 227
Table 5-89 OSError_GetSpinlock_SpinlockId 228
Table 5-90 OSError_ReleaseSpinlock_SpinlockId 228
Table 5-91 OSError_TryToGetSpinlock_SpinlockId 229
Table 5-92 OSError_TryToGetSpinlock_Success 229
Table 5-93 OSError_Os_SetExceptionContext_Context 231
Table 5-94 OSError_ControlIdle_IdleMode 231
Table 5-95 OSError_IocSend_IN 232
Table 5-96 OSError_IocWrite_IN 232
Table 5-97 OSError_IocSendGroup_IN 233
Table 5-98 OSError_IocWriteGroup_IN 233
Table 5-99 OSError_IocReceive_OUT 234
Table 5-100 OSError_IocRead_OUT 234
Table 5-101 OSError_IocReceiveGroup_OUT 235
Table 5-102 OSError_IocReadGroup_OUT 235
Table 5-103 OSError_StartOS_Mode 236
Table 5-104 OSError_ActivateTask_TaskID 236
Table 5-105 OSError_ChainTask_TaskID 237
Table 5-106 OSError_GetTaskID_TaskID 237
Table 5-107 OSError_GetTaskState_TaskID 238
Table 5-108 OSError_GetTaskState_State 238
Table 5-109 OSError_SetEvent_TaskID 239
Table 5-110 OSError_SetEvent_Mask 239
Table 5-111 OSError_ClearEvent_Mask 240
Table 5-112 OSError_GetEvent_TaskID 240
Table 5-113 OSError_GetEvent_Mask 241
Table 5-114 OSError_WaitEvent_Mask 241
Table 5-115 OSError_GetAlarmBase_AlarmID 242
Table 5-116 OSError_GetAlarmBase_Info 242
Table 5-117 OSError_GetAlarm_AlarmID 243
Table 5-118 OSError_GetAlarm_Tick 243
Table 5-119 OSError_SetRelAlarm_AlarmID 244

Table 5-120 OSError_SetRelAlarm_increment ... 244
Table 5-121 OSError_SetRelAlarm_cycle ... 245
Table 5-122 OSError_SetAbsAlarm_AlarmID ... 245
Table 5-123 OSError_SetAbsAlarm_start ... 246
Table 5-124 OSError_SetAbsAlarm_cycle ... 246
Table 5-125 OSError_CancelAlarm_AlarmID ... 247
Table 5-126 OSError_GetResource_ResID ... 247
Table 5-127 OSError_ReleaseResource_ResID ... 248
Table 5-128 OSError_Os_GetUnhandledIrq_InterruptSource ... 248
Table 5-129 OSError_Os_GetUnhandledExc_ExceptionSource ... 249
Table 5-130 OSError_BarrierSynchronize_BarrierID ... 249
Table 5-131 Os_GetVersionInfo ... 251
Table 5-132 Read Peripheral API ... 253
Table 5-133 Write Peripheral APIs ... 255
Table 5-134 Bitmask Peripheral API ... 258
Table 5-135 API Service Os_EnterPreStartTask ... 259
Table 5-136 Call Non-Trusted Function API ... 260
Table 5-137 API Service Os_DisableInterruptSource ... 262
Table 5-138 API Service Os_EnableInterruptSource ... 263
Table 5-139 API Service Os_ClearPendingInterrupt ... 264
Table 5-140 API Service Os_IsInterruptSourceEnabled ... 265
Table 5-141 API Service Os_IsInterruptPending ... 266
Table 5-142 API Service Os_InitialEnableInterruptSources ... 267
Table 5-143 API Service Os_GetDetailedError ... 268
Table 5-144 API Service Os_GetUnhandledIrq ... 269
Table 5-145 API Service Os_GetUnhandledExc ... 270
Table 5-146 Overview: Stack Usage Functions ... 272
Table 5-147 Conversion Macros from Time to Counter Ticks ... 274
Table 5-148 Conversion Macros from Counter Ticks to Time ... 274
Table 5-149 API Service Os_InitInterruptOnly ... 275
Table 5-150 API Service Os_Init ... 276
Table 5-151 API Service Os_InitMemory ... 276
Table 5-152 Barriers ... 289
Table 5-153 Os_GetExceptionContext ... 290
Table 5-154 Os_SetExceptionContext ... 291
Table 5-155 Os_GetCoreStartState ... 292

1 Introduction

This document describes the usage and functions of “MICROSAR Classic OS”, an operating system which implements the AUTOSAR BSW module “OS” as specified in [1]. An overview of the supported hardware as well as platform specific details and restrictions can be found in [9].

This documentation assumes that the reader is familiar with both the OSEK OS[1] specification and the AUTOSAR OS specification.

1.1 Architecture Overview

The following figure shows the location of the OS module within the AUTOSAR architecture.

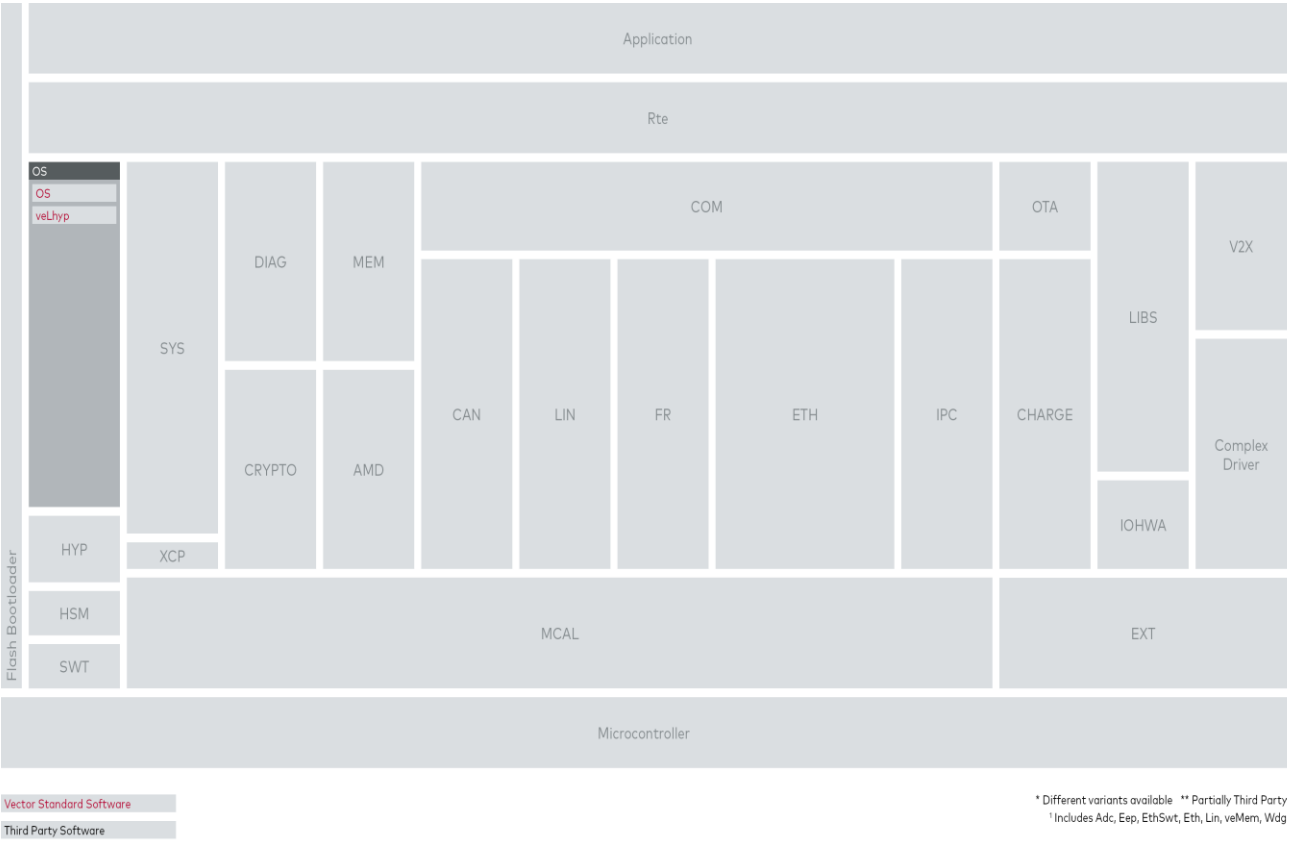

Figure 1-1 AUTOSAR Architecture Overview

[1] OSEK is a registered trademark of Continental Automotive GmbH (until 2007: Siemens AG)

1.2 Abstract

The MICROSAR Classic OS operating system is a real time operating system, which was specified for the usage in electronic control.

As a requirement, there is no dynamic creation of new tasks at runtime; all tasks have to be defined before compilation (pre-compile configuration variant).

The OS has no dynamic memory management and there is no shell for the control of tasks by hand.

Typically the source and configuration files of the operating system and the application source files are compiled and linked together to one executable file, which is loaded into an emulator or is burned into an EPROM or Flash EEPROM.

1.3 Characteristics

MICROSAR Classic OS has the following characteristics:

Supported Scalability Classes	SC1, SC2, SC3, SC4 (as described in [1])
Single Core ECUs	Supported
Multi Core ECUs	Supported
IOC	Supported

Table 1-1 MICROSAR Classic OS Characteristics

MICROSAR Classic OS supports various different processor families of different vendors in conjunction with multiple compilers.

The availability for a particular processor in conjunction with a specific compiler can be queried from Vector Informatik.

1.4 VTT OS

VTT OS stands for “vVIRTUALtarget Operating System”. It runs within Vectors CANoe development tool.

Vectors CANoe is capable of simulating an entire ECU network. Within such a simulated network the OS of each ECU can be simulated.

This is useful in early ECU development phases when no real hardware is available yet. Application development can be started at once.

The VTT OS behaves as regular AUTOSAR OS. All OS objects (e.g. tasks or ISRs) are simulated.

The VTT system is described in [6].

1.4.1 Characteristics of VTT OS

Supported Scalability Classes	SC1, SC2
Single Core ECUs	Supported
Multi Core ECUs	Up to 32 cores are supported
IOC	Supported
Number of Simulated Interrupt Sources	Up to 10000
Simulated Interrupt Levels	VTT OS allows interrupt levels from 1 .. 200 Whereas 1 is the lowest priority and 200 is the highest.
Memory Protection	Not supported[2]
Stack Protection	Not supported
Stack Usage Measurement	Not supported[3]
Stack Sharing	Not supported

Table 1-2 VTT OS characteristics

[2] The memory protection can be configured. However the actual protection mechanism is not executed.

[3] On the VTT platform the configured stacks are not really used. Calling the Stack Usage APIs is possible and does not lead to an error. The behavior of only returning 0 or 1 arises from the stacks being "consumed" via the function Os_Hal_ConsumeStack() while switching/resuming threads.

2 Functional Description

2.1 General

The MICROSAR Classic OS basically implements the OS according to the AUTOSAR OS standard referred in [1].

It is possible that MICROSAR Classic OS deviates from specified AUTOSAR OS behavior. All deviations from the standard are listed in the chapters hereafter.

On the other hand MICROSAR Classic OS extends the AUTOSAR OS standard with numerous functions. These extensions in function are described in detail in chapter 0.

2.2 MICROSAR Classic OS Deviations from AUTOSAR OS Specification

2.2.1 Generic Deviation for API Functions

Specified Behavior	There are some API functions which are only available within specific scalability classes (e.g. TerminateApplication() in SC3 and SC4 only).
Deviation Description	Within the MICROSAR Classic OS all API functions are always available.
Deviation Reason	The static OS code gets more simplified for better maintainability (less pre-processor statements are necessary). Modern toolchains will remove unused function automatically.

2.2.2 Trusted Function API Deviations

Specified Behavior	The Operating System shall not schedule any other Tasks which belong to the same OS-Application as the non-trusted caller of the service. Also interrupts of Category 2 which belong to the same OS-Application shall be disabled during the execution of the service.
Deviation Description	In MICROSAR Classic OS the re-scheduling of tasks in this particular case is not suppressed. The selective disabling of category 2 ISRs is also not done.
Deviation Reason	For a better runtime performance during trusted function calls the specified behavior is not implemented in MICROSAR Classic OS. Data consistency problems can be solved in a more efficient way by using the OS interrupt API and/or OS resource API.

Specified Behavior	All specified OS APIs should be called with interrupts enabled. In case CallTrustedFunction() API is called with disabled interrupts it returns the status code E_OS_DISABLEDINT.
Deviation Description	In MICROSAR Classic OS this limitation does not exist. It is allowed to call CallTrustedFunction() API with disabled interrupts. There is no error check. The return value E_OS_DISABLEDINT is not possible.
Deviation Reason	It offers the possibility to call CallTrustedFunction() API where interrupts may be disabled. This is more convenient and reasonable.

2.2.3 Service Protection Deviation

Specified Behavior	If an invalid address (address is not writable by this OS-Application) is passed as an out-parameter to an Operating System service, the Operating System module shall return the status code E_OS_ILLEGAL_ADDRESS.
Deviation Description	The validity of out-parameters is checked automatically by the MPU. Write accesses to such parameters are always done with the accessing rights of the caller of the OS service. If the address is invalid a MPU exception is raised. The return value E_OS_ILLEGAL_ADDRESS is not possible.
Deviation Reason	Hardware checks by the MPU are much more performant than software memory checks.

2.2.4 Code Protection

Specified Behavior	The Operating System module may provide an OS-Application the ability to protect its code sections against executing by non-trusted OS-Applications.
Deviation Description	The MICROSAR Classic OS does not support code section protection.
Deviation Reason	Design decision.

2.2.5 SyncScheduleTable API Deviation

Specified Behavior	All specified OS APIs should be called with interrupts enabled. In case SyncScheduleTable() is called with disabled interrupts it returns the status code E_OS_DISABLEDINT.
Deviation Description	In MICROSAR Classic OS this limitation does not exist. It is allowed to call SyncScheduleTable() with disabled interrupts. There is no error check. The return value E_OS_DISABLEDINT is not possible.
Deviation Reason	It offers the possibility to call SyncScheduleTable() where interrupts may be disabled. This is more convenient and reasonable.

2.2.6 CheckTask/ISRMemoryAccess API Deviation

Specified Behavior	All specified OS APIs should be called with interrupts enabled. In case one of these APIs is called with disabled interrupts it issues the error E_OS_DISABLEDINT.
Deviation Description	In MICROSAR Classic OS this limitation does not exist. It is allowed to call these API functions with disabled interrupts. There is no error check. The return value E_OS_DISABLEDINT is not possible.
Deviation Reason	It offers the possibility to call these functions e.g. from hardware drivers where interrupts may be disabled. This is more convenient and reasonable.

Specified Behavior	The API functions CheckTask/ISRMemoryAccess() are only allowed within specific OS call contexts (Task/Cat2 ISR/ErrorHook/ProtectionHook) In case one of these APIs is called within the wrong OS call context it issues the error E_OS_CALLEVEL.
Deviation Description	In MICROSAR Classic OS this limitation does not exist. It is allowed to call these API functions from all OS contexts. The return value E_OS_CALLEVEL is not possible.
Deviation Reason	Practically it is more reasonable to allow these APIs in all OS runtime contexts.

2.2.7 Interrupt API Deviation

Specified Behavior	The API functions SuspendOSInterrupts() and ResumeOSInterrupts() are allowed within a category 1 ISR
Deviation Description	In MICROSAR Classic OS it is not allowed to use SuspendOSInterrupts() and ResumeOSInterrupts() within a category 1 ISR.
Deviation Reason	If the function SuspendOSInterrupts()/ResumeOsInterrupts() is interrupted by a category 1 ISR which also calls the function SuspendOSInterrupts()/ResumeOsInterrupts(), this call may lead to data inconsistencies. Access to the OS datastructures is not atomic and therefore corruption of the nesting counter is possible if used in a category 1 ISR. For a better runtime performance during interrupt service calls, the specified behavior is not implemented in MICROSAR Classic OS.

2.2.8 Cross Core Getter APIs

Specified Behavior	All getter APIs (e.g. GetTaskID()) may be called cross core within hooks and non nestable category 2 ISRs.
Deviation Description	MICROSAR Classic OS does not allow usage of those functions within OS Hooks and non-nestable category 2 ISRs.
Deviation Reason	Deadlock avoidance due to disabled interrupts in case that there are two simultaneous concurrent usages of those APIs from multiple cores.

2.2.9 Return value upon stack violation

Specified Behavior	If a stack fault is detected by stack monitoring AND no ProtectionHook is configured, the Operating System module shall call the ShutdownOS() service with the status E_OS_STACKFAULT.
Deviation Description	Within a SC3 / SC4 system with MPU stack supervision: If a stack fault is detected by stack monitoring AND no ProtectionHook is configured, the Operating System module shall call the ShutdownOS() service with the status E_OS_PROTECTION_MEMORY.
Deviation Reason	With hardware stack supervision MICROSAR Classic OS is not possible to distinguish between stack violation and other memory violation

Specified Behavior	If a stack fault is detected by stack monitoring AND a ProtectionHook is configured the Operating System module shall call the ProtectionHook() with the status E_OS_STACKFAULT.
Deviation Description	Within a SC3 / SC4 system with MPU stack supervision: If a stack fault is detected by stack monitoring AND a ProtectionHook is configured the Operating System module shall call the ProtectionHook() with the status E_OS_PROTECTION_MEMORY.
Deviation Reason	With hardware stack supervision MICROSAR Classic OS is not possible to distinguish between stack violation and other memory violation

2.2.10 Handling of OS internal errors

Specified Behavior	In cases where the OS detects a fatal internal error all cores shall be shut down.
Deviation Description	In case that the OS detects an internal error the kernel panic mode is entered.
Deviation Reason	In case of OS internal errors normal operations (e.g. calling the protection hook) are possible no more, as the OS is in an inconsistent state.

2.2.11 Forcible Termination of Applications

Specified Behavior	AUTOSAR does not specify the handling of “next” schedule tables in case of forcible termination of applications.
Deviation Description	Use case: An application has a running schedule table which itself has a nexted schedule table of a foreign application. The foreign application is forcibly terminated. The OS removes the “next” request from the running schedule table.
Deviation Reason	Clarification of behavior. Impact on other applications should be minimal, therefore the current schedule table is not stopped. This is different to the behavior of StopScheduleTable().

Specified Behavior	AUTOSAR does not specify the handling of “next” schedule tables in case of forcible termination of applications.
Deviation Description	Use case: An application has a running schedule table which itself has a nexted schedule table of a foreign application. The first application is forcibly terminated. The OS stops the current schedule table of the terminated application. and removes the “next” request. As a result it does not switch to the “next” schedule table of the foreign application.
Deviation Reason	Clarification of behavior. Impact on other applications should be minimal. The described behavior is identical to the behavior of StopScheduleTable().

2.2.12 OS Configuration

Specified Behavior	The generator shall print out information about timers used internally by the OS during generation (e.g. on console, list file).
Deviation Description	In case of MICROSAR Classic OS there is no such output. Instead the timer is visible to the user as any other timer during configuration.
Deviation Reason	In order to increase the transparency, OS internal objects are visible to the user during configuration time.

Specified Behavior	If ShutdownOS() is called and ShutdownHook() returns then the Operating System module shall disable all interrupts and enter an endless loop.
Deviation Description	If ShutdownOS() is called and ShutdownHook() returns then the Operating System module enters the kernel panic mode.
Deviation Reason	In case of unusual situations the MICROSAR Classic OS enters the kernel panic mode. To keep the behaviour of the OS consistent, the kernel panic mode is also applied in case that the ShutdownHook() returns.

2.2.13 Spinlocks

Specified Behavior	The AUTOSAR Operating System shall generate an error if a TASK/ISR2 on a core, where the same or a different TASK/ISR already holds a spinlock, tries to seize another spinlock that has not been configured as a direct or indirect successor of the latest acquired spinlock (by means of the OsSpinlockSuccessor configuration parameter) or if no successor is configured.
Deviation Description	The nesting order check is only valid for a single task or ISR and if all nested spinlocks are members of the same lock order list.
Deviation Reason	By implementing this check, the user of MICROSAR Classic OS would be enforced to ▶ either configure a single lock order list ▶ or the user would be enforced to ensure correct nesting of spinlock between tasks or ISRs of different diagnostic coverage.

2.2.14 Errors within the PreTaskHook() and PostTaskHook()

Specified Behavior	The AUTOSAR Operating System shall call the ErrorHook(), if a system service that is called within the PreTaskHook() or PostTaskHook() does not return with E_OK.
Deviation Description	In MICROSAR Classic OS the ErrorHook() is never called within the PreTaskHook() and PostTaskHook(). The status value returned by the system service has to be evaluated by the user of MICROSAR Classic OS to mke sure that the service has been executed.
Deviation Reason	PreTaskHook() and PostTaskHook() are implemented as callbacks during the context switch. There shall be no additional context switch to the ErrorHook() in case of an errorneous usage of the AUTOSAR APIs.

2.2.15 ChainTask API Deviation

Specified Behavior	The AUTOSAR OS service ChainTask causes the termination of the calling task. After termination of the calling task a succeeding task is activated. Using this service ensures that the succeeding task starts to run at the earliest after the calling task has been terminated.
Deviation Description	In MICROSAR Classic OS the succedding task may start it's execution before the calling task has been terminated. This can only happen if the core where the calling task is executed differs from the core of the succeeding task (Cross core call of ChainTask API).
Deviation Reason	Before the calling task can be terminated, an activation request for the succeeding task is made by a cross core call. If the subsequent termination of the calling task needs more time than the cross core activation of the succeeding task, the succeeding task is active while the calling task is not terminated yet. The termination of the calling task can be delayed due to interrupt handling on the calling core or due to different runtime behaviour on the two cores.

Caution

Due to the deviation of the ChainTask API in cross core usage, the result of the GetTaskState API may be interpreted wrong.

Even if the succeeding task is already in RUNNING state, the calling task does not need to be in the SUSPENDED state.

Note

For VTT systems, the behavior can be either stimulated or suppressed by using the possibility to configure the minimum time the OS is spending in a waiting loop (used for synchroneous X-Signal). Lowering the time reduces the probability that the devation becomes active. For more information see chapter 4.8.3 (Timing Adjustment).

2.2.16 GetAlarmBase API Deviation

Specified Behavior	The AUTOSAR OS service GetAlarmBase returns E_OS_ACCESS if the owner application is not in the state APPLICATION_ACCESSIBLE.
Deviation Description	In MICROSAR Classic OS the GetAlarmBase API does not check the state of the Os-Application and executes succefully even if the application is not accessible, unless another specified error condition is met.
Deviation Reason	The API only reads constant data that is independent from the state of the Os-Application. Checking the state would require cross-core communication, which causes an unreasonable overhead to read constant data.

2.2.17 ARTI OS Deviations

MICROSAR Classic OS supports the ARTI debugging functionality according to [11], as it was not yet specified in [1]. The following deviations refer accordingly to the ARTI requirements from [11].

Specified Behavior	The `<eventParameter>` is an `uint32` representation of either one of the function parameters or the return value. It depends on the service call and is listed in the following table.
Deviation Description	In MICROSAR Classic OS the handled `<eventParameter>` in service call hooks is always zero.
Deviation Reason	The code complexity would increase significantly which would be negative for the runtime and the ROM consumption.

Specified Behavior	To be compatible to the pure OS state diagram, `AR_CP_OS_TASK` refers to this state model, knowing that tools need to postprocess the event flow to get all relevant information.
Deviation Description	MICROSAR Classic OS is able to support the extend `AR_CP_OSARTI_TASK` state model.
Deviation Reason	As specified by AUTOSAR, the `AR_CP_OSARTI_TASK` should be preferred if possible.

Specified Behavior	The class `AR_CP_OSARTI_CAT2ISR` contains events allowing the tracing of Cat2Isrs with an enhanced state model.
Deviation Description	MICROSAR Classic OS only supports the `AR_CP_OS_CAT2ISR` state model.
Deviation Reason	MICROSAR Classic OS is not able to represent the "Activated" state accordingly.

Specified Behavior	According to ECUC_Arti_00174 (see [11]), the ArtiOsIsrInstanceCategory supports the categories "CATEGORY_1" and "CATEGORY_2".
Deviation Description	MICROSAR Classic OS only creates ArtiOsIsrInstances for ISRs of "CATEGORY_2".
Deviation Reason	MICROSAR Classic OS stores no ARTI relevant information about "CATEGORY_1" ISRs.

Specified Behavior	According to ECUC_Arti_00127 (see [11]), the OS shall provide information about OsMessages in "ArtiOsMessageContainer" containers.
Deviation Description	MICROSAR Classic OS does not support OsMessages.
Deviation Reason	OsMessages are not specified by AUTOSAR (see [1]).

Specified Behavior	According to ECUC_Arti_00120 (see [11]), the OS shall provide information about Task contexts in "ArtiOsContext" containers.
Deviation Description	MICROSAR Classic OS does not provide any information about task contexts to the ARTI module.
Deviation Reason	The context information depends heavily on the underlying hardware, which would require a specific implementation for each supported hardware.

2.3 Stack Concept

MICROSAR Classic OS implements a specific stack concept.

It defines different stacks which may be used by stack consumers (runtime contexts). Whereas not all stacks may be used by all consumers.

The following table gives an overview.

Stack Type	Multiplicity	Possible Stack Consumers
Init Stack	1 per core	> OS initialization, Os_PanicHook(), Category 0/1 ISRs
Kernel stack	1 per core	> OS memory exception handling > Os_PanicHook() > Category 0 ISRs
Protection stack	0..1 per core	> ProtectionHook() > OS API calls > Os_PanicHook() > Category 0 ISRs
Error stack	0..1 per core	> ErrorHooks (global and OS-application specific) > OS API calls > Category 0/1 ISRs

		> Os_PanicHook()
Shutdown stack	0..1 per core	> ShutdownHooks (global and OS-application specific) > OS API calls > Os_PanicHook() > Category 0 ISRs
Startup stack	0..1 per core	> StartupHooks (global and OS-application specific) > OS API calls > Category 0/1 ISRs > Os_PanicHook()
NTF stacks	0..n	> Non-trusted functions > OS API calls > OS ISR wrapper > Trusted functions > Alarm callback functions > Pre / PostTaskHook() > Category 0/1 ISRs > Os_PanicHook()
No nesting interrupt stack	0..1 per core	> No nesting category 2 ISRs > OS API calls > Trusted functions > Alarm callback functions > Category 0/1 ISRs > Os_PanicHook()
Interrupt level stacks	0..n	> Nesting category 2 ISRs > OS API calls > OS ISR wrapper > Trusted functions > Alarm callback functions > Category 0/1 ISRs > Os_PanicHook()
Task stacks	1..n	> Tasks > OS API calls > OS ISR wrapper > Trusted functions > Alarm callback functions > Pre / PostTaskHook() > Category 0/1 ISRs > Os_PanicHook()

IOC receiver pull callback stack	0..1 per core	> IOC receiver pull callback functions > Category 0 ISRs

Table 2-1 MICROSAR Classic OS Stack Types

Note
The stack sizes of all stacks must be configured within the ECU configuration

2.3.1 Task Stack Sharing

2.3.1.1 Description

In order to save RAM it is possible that different basic tasks share the same task stack. Tasks which fulfill the following requirements share a stack:

> Basic tasks which have the same configured priority.

> Basic tasks which are non-preemptive and are configured to share stacks. Within such basic tasks the call of the OS service Schedule() is not allowed.

> Basic tasks which share an internal resource and are configured to share stacks. Within such basic tasks the call of the OS service Schedule() is not allowed.

2.3.1.2 Activation

The attribute "OsTaskStackSharing" of a basic task has to be set to TRUE. The OS decides then in dependancy of the preemption settings and assigned internal resources whether the stack of basic tasks may be shared or not.

The size of the shared task stack is the maximum of all stack sizes of tasks which share the stack.

Note
The OS activates stack sharing automatically for basic tasks with the same configured priority regardless of the value of OsTaskStackSharing.

Note
By setting "OsTaskStackSharing" to TRUE the OS API service Schedule() may not be called within the corresponding basic task.

The OS throws an error if Schedule() is called within a task with activated stack sharing.

Note
Stack sharing of tasks can only be achieved between tasks which are assigned to the same core!

2.3.1.3 Usage

Tasks which are cooperative to each other are sharing the same stack. No additional actions are necessary.

2.3.2 ISR Stack Sharing

2.3.2.1 Description

In order to save RAM it is possible that different category 2 ISRs share the same ISR stack.

- All category 2 ISRs which are not nestable can share one stack.
- All Category 2 ISRs which have the same priority can share one stack.

2.3.2.2 Activation

The attribute "OsIsrEnableNesting" must be set to FALSE for a category 2 ISR.

The size of the shared ISR stack is the maximum of all configured ISR stack sizes of non-nestable category 2 ISRs.

Note
Stack sharing of ISRs can only be achieved between ISRs which are assigned to the same core!

2.3.2.3 Usage

The feature is used automatically by the OS. All category 2 ISRs on the same core which are not nestable are sharing the same stack.

2.3.3 Stack Check Strategy

All OS stacks must be protected from overflowing.

MICROSAR Classic OS offers different strategies to detect stack overflows or even to prevent stacks from overflowing.

In dependency of the configured scalability class there are the following strategies:

Scalability Class	Stack check strategy
SC1 / SC2	Software stack check (see 2.3.4)
SC3 / SC4	Stack monitoring by memory protection peripherals (MPU / MMU) (see 2.3.5)

2.3.4 Software Stack Check

2.3.4.1 Description

The OS initializes the very last element of each stack to a specific stack check pattern. Whenever a stack switch is performed (e.g. a task switch) the OS checks whether this last element of the valid stack still holds the stack check pattern.

If the OS detects that the stack check pattern has been altered it assumes that the last valid stack did overflow.

	Stack Check Pattern
32-Bit Microcontrollers	0xAAAAAAAA

Table 2-2 Stack Check Patterns

Note
The software stack check is able to detect stack overflows. It is not capable to avoid them!

Caution
The software stack check is not able to detect all stack overflows. There may be scenarios where the memory of the adjacent stack is already overwritten, but the last element of the current stack still holds the stack check pattern.

In such cases the software stack check is not able to detect the overflow.

Caution
The software stack check is not able to detect the amount memory which has been destroyed.

Caution
In case of error reporting due to a stack fault (E_OS_STACK_FAULT), the API GetTaskID() might not return the ID of the causing task.

2.3.4.2 Activation

Within a SC1 or SC2 configuration the attribute “OsStackMonitoring” has to be set to TRUE to activate the software stack check feature.

Expert Knowledge
On platforms which disable the MPU in supervisor mode, the software stack check may be activated also for SC3 and SC4 configurations.

On other platforms the software stack check should be switched off in a SC3 or SC4 configuration.

2.3.4.3 Usage

Once the feature is activated the OS checks the stacks automatically upon each stack switch.

If the OS detects a stack overflow it goes into shutdown. If a ShutdownHook is configured it is invoked to inform the application about OS shutdown.

Note
Debugging hint: The stack check pattern is restored by the OS before the ShutdownHook() is called.

2.3.5 Stack Monitoring by MPU / MMU

2.3.5.1 Description

During the whole runtime of the OS the current active stack is monitored by a memory protection peripheral (MPU / MMU) of the microcontroller.

Stack overflows cannot happen since the memory protection peripheral avoids write accesses beyond the stack boundaries.

Whenever a memory violation is recognized (e.g. due to a stack violation) an exception is raised. Within the exception handling the OS calls the ProtectionHook().

The application decides in the ProtectionHook() how to deal with the memory protection violation. If the application invokes the shutdown of the OS, the ShutdownHook() is called as well (if configured).

Note
The stack supervision recognizes write accesses beyond stack boundaries and suppresses them.

Note
In case an MPU is used, the OS reserves one hardware MPU region which is reprogrammed by the OS on every stack switch.

2.3.5.2 Activation

The system must be configured as a SC3 or SC4 system.

2.3.5.3 Usage

In a SC3 / SC4 system the OS automatically provides a memory region for stack monitoring.

To safely detect stack violations special care must be taken with configuring additional memory regions and also with linking of sections:

> When configuring additional memory regions, the memory region must never grant write access to any OS stack section. Also overlapping of regions need to be considered here (see chapter 2.17.2 for constraints on the MPU / MMU configuration).

> By using an OS generated linker command file it is assured that the OS stacks are linked consecutively into the RAM.

> A stack safety gap is needed which is linked adjacent to the stacks (dependent on the stack growth direction; see Figure 2-1). Otherwise, a stack violation of the stack with the lowest address cannot be detected. No software parts must have write access to the stack safety gap.

> The size of the stack safety gap must be at least the granularity of the MPU. This restriction is usually not sufficient, as the stack may be accessed with an offset larger than the MPU granularity. A possible solution is to link the stack section at the start or end of the RAM (dependent on the stack growth direction).

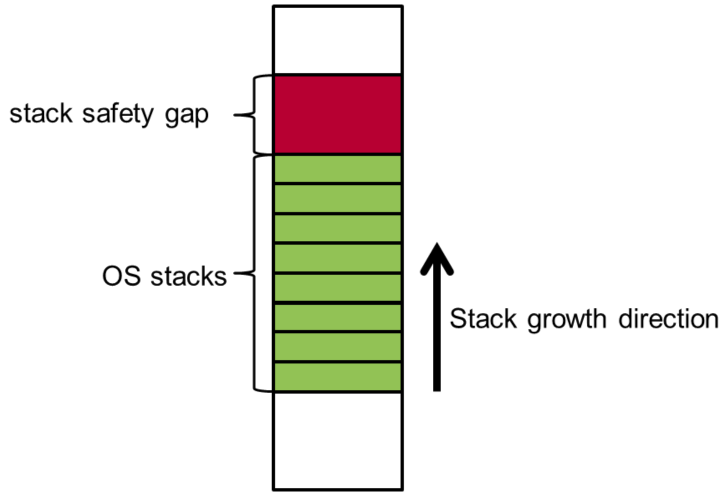

Figure 2-1 Stack Safety Gap

Caution
Don't configure memory regions which grant write access to any OS stack.

Caution
Add a stack safety gap to the linkage scheme. The stack safety gap is a restricted memory region. No software parts must have write access to this region.
The safety gap is not needed if the stacks are linked adjacent to a memory area where write access is restricted by hardware (e.g. ROM).

2.3.6 Stack Usage Measurement

2.3.6.1 Description

During runtime of the OS the maximum stack usage can be obtained by the application. The OS initializes all OS stacks with the stack check pattern (see Table 2-2).

There are API functions which are capable to return the maximum stack usage (since call of StartOS()) for each stack (see 5.2.8).

2.3.6.2 Activation

Set "OsStackUsageMeasurement" to TRUE.

2.3.6.3 Usage

The stack usage APIs can be used anywhere in application.

Note
To save OS startup time, the feature can be deactivated in a productive environment.

2.4 Interrupt Concept

2.4.1 Interrupt Handling API

The AUTOSAR OS standard specifies several APIs to disable / enable Interrupts.

DisableAllInterrupts() **EnableAllInterrupts()**	The functions disable all category 1 and category 2 interrupts.
SuspendAllInterrupts() **ResumeAllInterrupts()**	
SuspendOSInterrupts() **ResumeOSInterrupts()**	The functions disable category 2 interrupts only.

2.4.2 Interrupt Levels

The OS defines several interrupt levels.

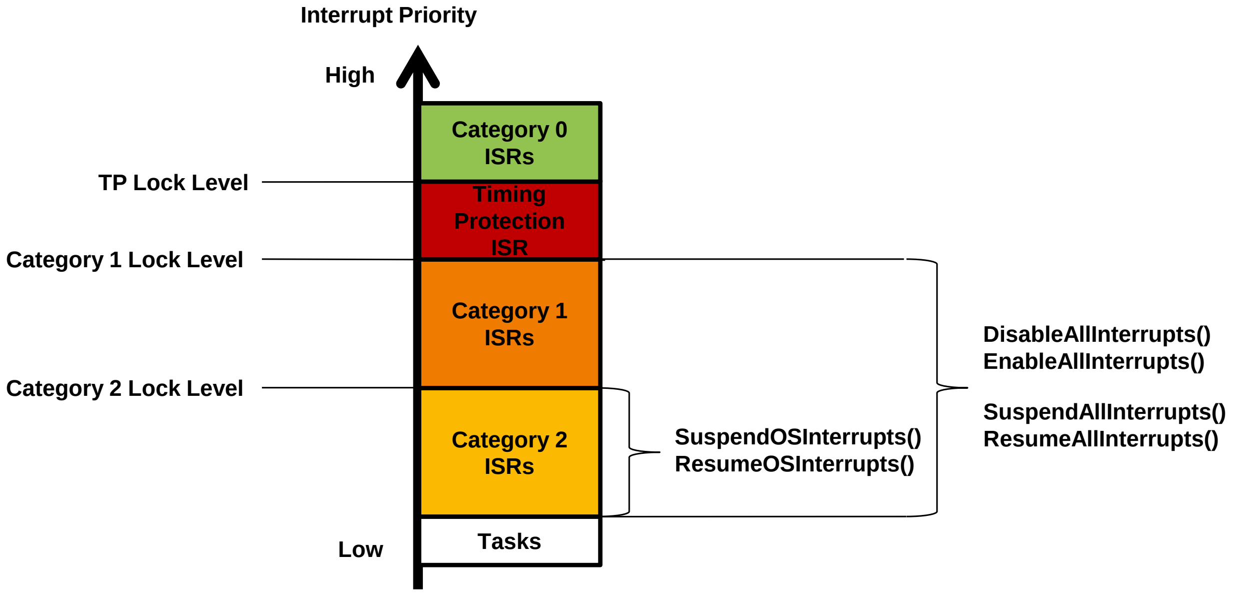

Figure 2-2 Interrupt Lock Levels

> Category 2 ISRs must have a lower priority than category 1 ISRs
> Category 1 ISRs must have a lower priority than the timing protection ISR (within an SC2 / SC4 system)
> The timing protection ISR must have a lower priority than category 0 ISRs (category 0 ISRs are described in detail in chapter 3.14)
> The TP Lock Level cannot be set by the user. Interrupts are disabled up to this level OS internally whenever timing protection is handled.
> Category 0 ISRs are disabled OS internally for very short times only e.g. when performing a stack switch (the locations where category 0 ISRs are locked can be found in chapter 3.14.2.4).

2.4.3 Interrupt Vector Table

The interrupt vector table is generated by MICROSAR Classic OS with respect to the configuration, microcontroller family and used compiler.

In a multi core system multiple vector tables may be generated.

MICROSAR Classic OS generates an interrupt vector for each possible interrupt source.

2.4.4 Nesting of Category 2 Interrupts

2.4.4.1 Description

To keep interrupt latency as low as possible it is possible that

> A higher priority category 2 ISR interrupts a lower priority category 2 ISR.
> A category 1 ISRs interrupts a category 2 ISR (category 1 ISR has always a higher priority)

2.4.4.2 Activation

When setting “OsIsrEnableNesting” to TRUE the category 2 ISR itself is interruptible by higher priority ISRs.

2.4.5 Category 1 Interrupts

2.4.5.1 Implementation of Category 1 ISRs

MICROSAR Classic OS offers a macro for implementing a category 1 ISR. This is a similar mechanism like the macro for a category 2 ISR defined by the AUTOSAR standard.

MICROSAR Classic OS abstracts the needed compiler keywords.

Implement a category 1 ISR

```
OS_ISR1(<MyCategory1ISR>)
{
}
```

2.4.5.2 Nesting of Category 1 ISRs

Since category 1 ISRs are directly called from interrupt vector table without any OS pro- and epilogue, automatic nesting of category 1 ISRs cannot be supported.

The configuration attribute “OsIsrEnableNesting” is ignored for category 1 ISRs.

Nevertheless the interrupts may be enabled during a category 1 ISR to allow interrupt nesting but OS API functions cannot be used for this purpose. The application has to use compiler intrinsic functions or inline assembler statements.

Example

```
OS_ISR1(<MyCategory1ISR>)
{
    __asm(EI); /* enable nesting of this ISR */

    __asm(DI); /* disable nesting before leaving the function */
}
```

2.4.5.3 Category 1 ISRs before StartOS

There may be the need to activate and serve category 1 ISRs before the OS has been started.

The following sequence should be implemented:

1. Call Os_InitMemory()
2. Call Os_Init() (within the function the basic interrupt controller settings are initialized e.g. priorities of interrupt sources).
3. Enable the Interrupt sources of category 1 ISRs by directly manipulating the control registers in the interrupt controller.
4. Enable interrupt handling by calling OS_EnableInterruptsPreStart().

2.4.5.4 Notes on Category 1 ISRs

Expert Knowledge
On platforms which have no automatic stack switch upon interrupt request there will be no stack switch at all if a category 1 ISR occurs. Thus the stack consumption of a category 1 ISR should be added to all stacks which are can be consumed by category 1 ISRs (see 2.2.14 for an overview).

Note
Although the interrupt priorities are initialized by MICROSAR Classic OS there is no API to enable or acknowledge category 1 ISRs. The interrupt control registers have to be accessed directly. For VTT OS, you may use the function CANoeEmuProcessor_UnmaskInterrupt(OS_IRQ_<Shortname>) to enable category 1 interrupts.

Caution
The AUTOSAR OS standard does not allow OS API usage within category 1 ISRs (the only exception is the interrupt handling API).

If a not allowed OS API is called anyway, MICROSAR Classic OS is not able to detect this and the called API may not work as expected.

Caution
Category 1 ISRs are always executed with trusted rights on supervisor level.

Caution

The macro "OS_ISR1" abstracts the appropriate compiler keyword for implementing the interrupt service routine. Thus, the compiler generates code which saves and restore a subset of the general purpose registers.

In certain use cases, e.g. usage of the FPU or nested interrupts, it may require the application to save and restore more registers.

2.4.6 Initialization of Interrupt Sources

Through the OS configuration MICROSAR Classic OS knows the assignment of interrupt sources and priorities to ISRs. In multi core system the core assignment of all ISRs is also known.

Based on these configuration information MICROSAR Classic OS generates data structures for initializing the interrupt controller. It initializes the interrupt priority and its core assignment.

Note

Enabling of interrupt sources:

The OS enables the interrupt sources only for the OS generated timer ISRs.

Other user ISRs can be only be served if the corresponding interrupt sources are enabled by the application.

This should be done by using the interrupt source API (see 5.2.6 for details; function Os_EnableInterruptSource()).

2.4.7 Unhandled Interrupts

Interrupt sources which are not assigned to a user defined ISR are assigned to a default OS interrupt handler which collects those interrupt sources.

Thus interrupt requests from unassigned interrupt sources are handled by the OS. Within OS Hooks (e.g. ProtectionHook()) the application can obtain the source number of the unhandled interrupt request by an OS API (see 5.2.7.1 for details).

In case of an unhandled interrupt request which has occurred within OS code MICROSAR Classic OS calls the PanicHook() because an inconsistent internal state is recognized and the OS does not know how to correctly continue execution.

In case of an unhandled interrupt request which has occurred within critical user sections, i.e. StartupHook, ErrorHook, PreTaskHook, PostTaskHook, Alarm callbacks, IOC receiver callbacks, Timing Hooks, ProtectionHook and ShutdownHook, MICROSAR Classic OS calls the PanicHook() because an inconsistent internal state is recognized and the OS does not know how to correctly continue execution.

In all other cases of an unhandled interrupt request MICROSAR Classic OS calls the ProtectionHook() with the parameter E_OS_SYS_PROTECTION_IRQ.

2.4.8 Unhandled Syscalls

Syscall sources which are not assigned to OS or user handlers are assigned to a default OS syscall handler which collects those exceptions.

Thus syscall requests from unassigned syscall sources are handled by the OS.

In case of an unhandled syscall request which has occurred within OS code MICROSAR Classic OS calls the PanicHook() because an inconsistent internal state is recognized and the OS does not know how to correctly continue execution.

In case of an unhandled syscall request which has occurred within critical user sections, i.e. StartupHook, ErrorHook, PreTaskHook, PostTaskHook, Alarm callbacks, IOC receiver callbacks, Timing Hooks, ProtectionHook and ShutdownHook, MICROSAR Classic OS calls the PanicHook() because an inconsistent internal state is recognized and the OS does not know how to correctly continue execution.

In all other cases of an unhandled syscall request MICROSAR Classic OS calls the ProtectionHook() with the parameter E_OS_SYS_PROTECTION_SYSCALL.

2.5 Exception Concept

2.5.1 Exception Vector Table

The exception vector table is generated by MICROSAR Classic OS with respect to the configuration, microcontroller family and used compiler.

In a multi core multiple vector tables may be generated.

MICROSAR Classic OS generates an exception vector for each possible exception source.

> **Note**
> In a SC3 and SC4 system MICROSAR Classic OS defines OS exception handlers for memory protection errors and for SYSCALL / TRAP instructions.
>
> Exception sources which are used by the OS cannot be configured by the application.

2.5.2 Unhandled Exceptions

Exception sources which are not assigned to user defined exception handlers are assigned to a default OS exception handler which collects those exceptions.

Thus exception requests from unassigned exception sources are handled by the OS. Within OS Hooks the application can obtain the exception number of the unhandled exception request by an OS API (see 5.2.7.3 for details).

Furthermore the address of the instruction that caused the latest exception, can be obtained by a OS API (see 5.2.7.4 for details).

In case of an unhandled exception request which has occurred within OS code MICROSAR Classic OS calls the PanicHook() because an inconsistent internal state is recognized and the OS does not know how to correctly continue execution.

In case of an unhandled exception request which has occurred within critical user sections, i.e. StartupHook, ErrorHook, PreTaskHook, PostTaskHook, Alarm callbacks, IOC receiver callbacks, Timing Hooks, ProtectionHook and ShutdownHook, MICROSAR Classic OS calls the PanicHook() because an inconsistent internal state is recognized and the OS does not know how to correctly continue execution.

In all other cases of an unhandled exception request MICROSAR Classic OS calls the ProtectionHook() with the parameter E_OS_PROTECTION_EXCEPTION.

2.5.3 Configuration

Because of the many similarities to interrupts, exeption handlers are configured like interrupt service routines (ISRs) with only slight differences. The table below shows the attributes which need to be configured differently to interrupt service routines:

Attribute	Value
/MICROSAR/Os/OsIsr/OsIsrInterruptType	EXCEPTION
/MICROSAR/Os/OsIsr/OsIsrCategory	CATEGORY_1 or CATEGORY_0

Table 2-3 Configuration attributes for exceptions

Mind that some other attributes of ISRs have no meaning for exceptions and will simply be ignored. It is hardware dependent, which other attributes have no meaning for exceptions, therefore, they are not described here.

2.5.4 Defining an own exception handler

The definition of an own exception handler is more difficult than the definition of a Category 0 or Category 1 ISR. Mind that an exception may occur even when interrupts are globally disabled and at any time, even while the sytem is in an inconsistent state. Inconsistent states could be an invalid stack pointer, a stack pointer which points to a memory region which is protected against write accesses or any other inconsistency.

In case that the handling of an exception is necessary, please check:

- If it is possible to use the unhandled exception handlers of the OS (by not defining an own handler). These consider the potential inconsistent states and they will call the ProtectionHook. Perhaps, it is possible to do the handling of the exception inside the ProtectionHook. In case a return to the normal program flow shall occur, the usage of exception context manipulation (see chapter 3.13) may be necessary/helpful.
- If the code of the unhandled exception handlers of the OS can be used as example code to circumvent all the potential inconsistent states. So you may copy the ideas of the unhandled exception handler of the OS into your own code. Mind that this may violate the safety manual or the hardware software interface of the OS, as you might have to access hardware registers. This will need special care in safety systems.

2.6 Timer Concept

2.6.1 Description

MICROSAR Classic OS can provide a time base generated from timer hardware located on the microcontroller. This time base can be used to drive alarms and schedule-tables.

2.6.2 Activation

The OS configuration may define an OsCounter Object of type “HARDWARE”. Then a driving hardware must be assigned to “OsDriver” attribute.

2.6.3 Usage

Once the hardware counter is defined it can be assigned to alarms (“OsAlarmCounterRef”) and schedule-tables (“OsScheduleTableCounterRef”).

Such alarms and schedule-tables are driven time based.

Additionally MICROSAR Classic OS provides conversion macros (which are based on the hardware counter configuration) to convert from hardware ticks to time and vice versa (see for 5.2.10 details).

2.6.4 Dependencies

A hardware counter can be driven in two modes:

> Periodical interrupt timer mode (see 2.7)

> High resolution timer mode (see 2.8)

2.7 Periodical Interrupt Timer (PIT)

2.7.1 Description

The timer hardware is set up to generate timer interrupts requests in a strict periodical interval (e.g. 1ms). The interval does not change during OS runtime.

Within each timer ISR MICROSAR Classic OS checks for alarm and schedule-table expirations and execute the configured OS action.

2.7.2 Activation

> Define an OsCounter of type “HARDWARE” and select the timer Hardware in “OsDriver”.

> Set the counter sub-attribute “OsDriverHighResolution” to FALSE.

> The attribute “OsSecondsPerTick” specifies the cycle time of interrupt generation.

> The attribute “OsCounterTicksPerBase” specifies the number of timer counter cycles which are necessary to reach “OsSecondsPerTick”.

Note
The OS will add an appropriate ISR automatically to the configuration.

2.7.3 Driver Configuration

Caution
16 Bit FRT timers must not be configured as a driver for a PIT.

2.8 High Resolution Timer (HRT)

2.8.1 Description

The timer hardware is set up to generate one timer interrupt request when an alarm or schedule-table action shall be executed.

Within each timer ISR MICROSAR Classic OS performs that action, calculates the timer interval for the next action and reprograms the timer hardware with the new expiration time.

2.8.2 Activation

- Define an OsCounter of type “HARDWARE” and select the timer Hardware in “OsDriver”.
- Set the counter sub-attribute “OsDriverHighResolution” to TRUE.
- The attribute “OsSecondsPerTick” specifies the cycle time of the timer counter.
- The attribute “OsCounterTicksPerBase” must be set to “1”.
- The attribute “OsCounterMaxAllowedValue” must be set to 0x3FFFFFFF

Note
The OS will add an appropriate ISR automatically to the configuration.

Caution
To avoid corruption of the OS time base, the HRT ISR must not be delayed longer than a half hardware counter cycle.

For a 16 Bit hardware timer for instance, a half hardware counter cycle is the time needed to count from 0 to 0x7FFF.

2.9 PIT versus HRT

	PIT	HRT
Interrupt Requests are generated …	▶ Strictly periodical	▶ On demand
Precision of Alarms / Schedule-tables	▶ Only multiples of the attribute OsSecondsPerTick are possible for alarm / schedule-table times.	▶ Any times are possible. With precision of the cycle time of the used timer hardware.
Interrupt Load	▶ Generates a constant interrupt load which is equally distributed over runtime.	▶ Interrupt load is not equally distributed over runtime. ▶ Interrupt bursts may be possible.

Table 2-4 PIT versus HRT

2.10 Startup Concept

The following figure gives an overview of the different startup phases of the OS. It also shows which OS API functions are available in the different phases.

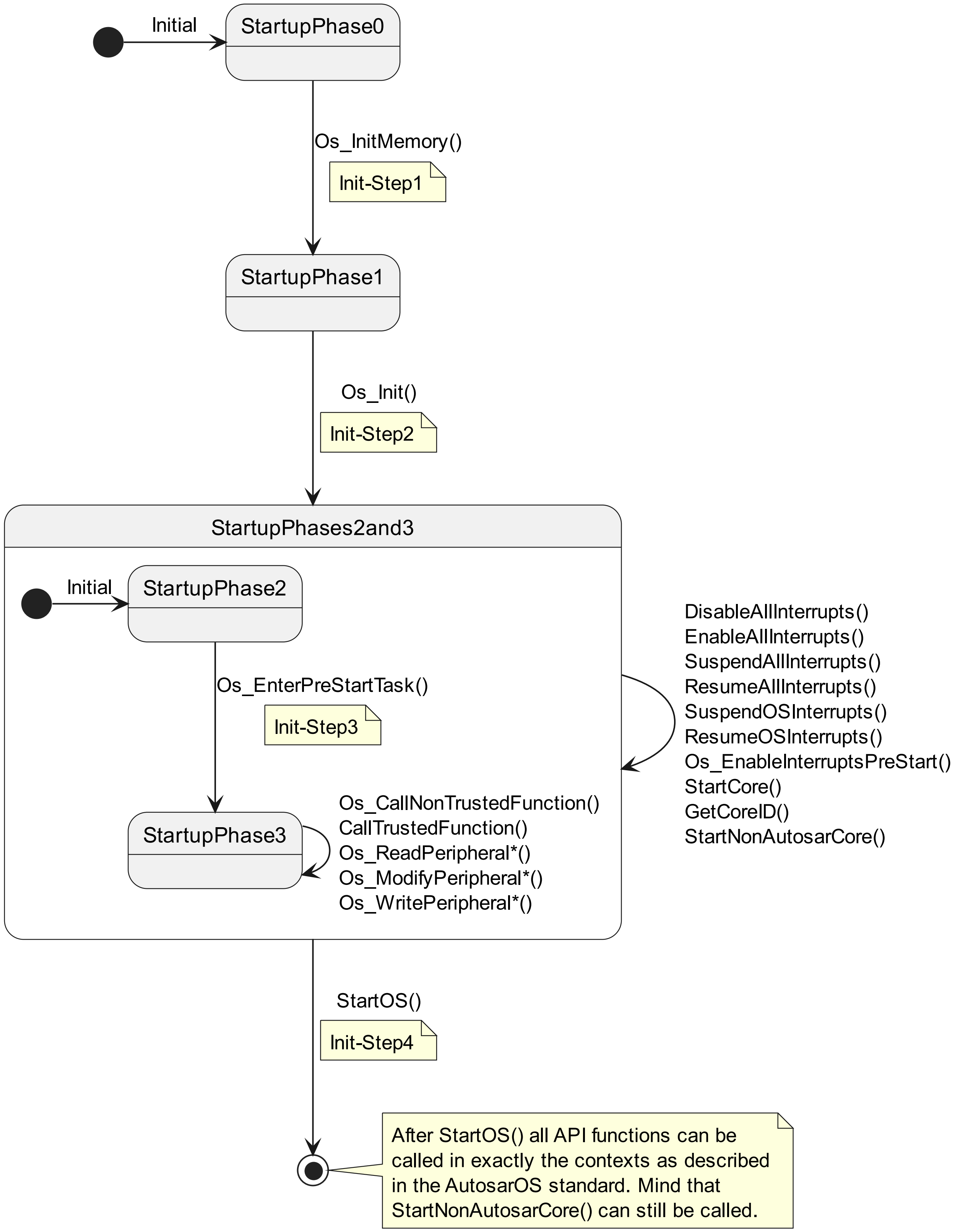

Figure 2-3 API functions during startup

2.11 Single Core Startup

This chapter shows some examples how MICROSAR Classic OS is started as single core OS.

2.11.1 Single Core Derivatives

OS single core startup on a single core derivative

```
void main (void)
{
   Os_InitMemory();
   Os_Init();
   StartOS(OSDEFAULTAPPMODE);
}
```

2.11.2 Multi Core Derivatives

2.11.2.1 Examples for SC1 / SC2 Systems

OS single core startup on a multi core derivative

```
void main (void)
{
   StatusType rv;

   Os_InitMemory();
   Os_Init();

   switch(GetCoreID())
   {
      case OS_CORE_ID_MASTER:
         StartNonAutosarCore(OS_CORE_ID_1, &rv); /* call of StartNonAutosarCore is
                                                    optional the other core may also be
                                                    held in reset */
         StartOS(OSDEFAULTAPPMODE);
         break;
      case OS_CORE_ID_1:
         /* don't call StartOS; do something else */
         break;
      default:
         break;
   }
}
```

The example starts a single core OS on the master core of a multi core derivative.

OS single core startup on a multi core derivative

```
void main (void)
{
   StatusType rv;

   Os_InitMemory();
   Os_Init();

   switch(GetCoreID())
   {
      case OS_CORE_ID_MASTER:
         StartCore(OS_CORE_ID_1, &rv);
         /* don’t call StartOS; do something else */
         break;
      case OS_CORE_ID_1:
         StartOS(OSDEFAULTAPPMODE);
         break;
      default:
         break;
   }
}
```

The example starts a single core OS on the slave core of a multi core derivative

2.11.2.2 Examples for SC3 / SC4 Systems

Caution

The function GetCoreID requires a trap into the OS to be functional. Since the OS does not initialize any trap tables on non-AUTOSAR cores GetCoreID cannot be used on such cores.

Therefore it is not possible to use the API function GetCoreID within the main function. A user function (e.g. UsrGetCoreID) is necessary which distinguishes the correct core ID.

OS single core startup on a multi core derivative

```
void main (void)
{
   StatusType rv;

   Os_InitMemory();
   Os_Init();

   switch(UsrGetCoreID())
   {
      case 0:
         StartNonAutosarCore(OS_CORE_ID_1, &rv); /* call of StartNonAutosarCore is
                                                    optional the other core may also be
                                                    held in reset */
         StartOS(OSDEFAULTAPPMODE);
         break;
      case 1:
         /* don't call StartOS; do something else */
         break;
      default:
         break;
   }
}
```

The example starts a single core OS on the master core of a multi core derivative.

2.12 Multi Core Startup

Within a multi core system each core has the following possibilities when entering the main function:

1. Mandatory: call to Os_InitMemory() and Os_Init().
2. Optional: calls to StartCore() to start additional cores under control of MICROSAR Classic OS.
3. Optional: calls to StartNonAutosarCore() to start additional cores which are independent of MICROSAR Classic OS.
4. Optional: call StartOS() to start MICROSAR Classic OS on the core

    For a slave core this is only possible if the core once has been started with StartCore() API from another core.

    For the master core this is only possible if the core itself is configured to be an AUTOSAR core.

2.12.1 Example for SC1 / SC2 Systems

OS multi core startup

```
void main (void)
{
   StatusType rv;

   Os_InitMemory();
   Os_Init();

   switch(GetCoreID())
   {
      case OS_CORE_ID_MASTER:
         StartCore(OS_CORE_ID_1, &rv);
         StartCore(OS_CORE_ID_2, &rv);
         StartOS(OSDEFAULTAPPMODE);
         break;
      case  OS_CORE_ID_1:
         StartOS(DONOTCARE);
         break;
      case  OS_CORE_ID_2:
         StartCore(OS_CORE_ID_3, &rv);
         StartOS(DONOTCARE);
         break;
      case  OS_CORE_ID_3:
         StartOS(DONOTCARE);
         break;
      default:
         break;
   }
}
```

The example shows a possible startup sequence for a quad core system.

2.12.2 Examples for SC3 / SC4 systems

2.12.2.1 Only with AUTOSAR Cores

OS multi core startup

```
void main (void)
{
   StatusType rv;

   Os_InitMemory();
   Os_Init();

   switch(GetCoreID())
   {
      case OS_CORE_ID_MASTER:
         StartCore(OS_CORE_ID_1, &rv);
         StartCore(OS_CORE_ID_2, &rv);
         StartOS(OSDEFAULTAPPMODE);
         break;
      case  OS_CORE_ID_1:
         StartOS(DONOTCARE);
         break;
      case  OS_CORE_ID_2:
         StartCore(OS_CORE_ID_3, &rv);
         StartOS(DONOTCARE);
         break;
      case  OS_CORE_ID_3:
         StartOS(DONOTCARE);
         break;
      default:
         break;
   }
}
```

The example shows a possible startup sequence for a quad core system. All cores are configured to be AUTOSAR cores.

2.12.2.2 Mixed Core System

Caution

The function GetCoreID requires a trap into the OS to be functional. Since the OS does not initialize any trap tables on non-AUTOSAR cores GetCoreID cannot be used on such cores.

Therefore it is not possible to use the API function GetCoreID within the main function. A user function (e.g. UsrGetCoreID) is necessary which distinguishes the correct core ID.

OS multi core startup

```
void main (void)
{
   StatusType rv;

   Os_InitMemory();
   Os_Init();

   switch(UsrGetCoreID())
   {
      case 0:
         StartNonAutosarCore(OS_CORE_ID_1, &rv);
         StartCore(OS_CORE_ID_2, &rv);
         StartOS(OSDEFAULTAPPMODE);
         break;
      case 1:
         /* not an AUTOSAR core; do something else */
         break;
      case 2:
         StartCore(OS_CORE_ID_3, &rv);
         StartOS(DONOTCARE);
         break;
      case 3:
         StartOS(DONOTCARE);
         break;
      default:
         break;
   }
}
```

The example shows a possible startup sequence for a quad core system. Three cores are AUTOSAR cores and one core is a non-AUTOSAR core.

2.13 Error Handling

MICROSAR Classic OS is able to detect and handle the following types of errors:

Application Errors …	▶ Are raised if the OS could not execute a requested OS API service correctly. Typically the OS API is used wrong (e.g. invalid object ID). ▶ Do not corrupt OS internal data. ▶ Will result in call of the global ErrorHook() for centralized error handling (if configured). ▶ Will result in call of an application specific ErrorHook (if configured). ▶ May not induce shutdown / terminate reactions. Instead the application may continue execution by simply returning from the ErrorHooks.
Protection Errors …	▶ Are raised if the application violates its configured boundaries (e.g. memory access violations, timing violations). ▶ Do not corrupt OS internal data. ▶ Are raised upon occurrence of unhandled exceptions and interrupts. ▶ Will result in call of the ProtectionHook() where a shutdown or terminate handling (with or without restart) is induced. ▶ If termination is induced but the feature “OsForcibleTermination” has not been configured, the OS goes into shutdown. In this case the ShutdownHook() is called with the error code E_OS_SYS_DISABLED (if configured). ▶ If shutdown is induced the ShutdownHook() is called (if configured). ▶ If no ProtectionHook() is configured shutdown is induced.
Kernel Errors …	▶ Are raised if the OS cannot longer assume the correctness of its internal data (e.g. memory access violation during ProtectionHook()) ▶ The OS will disable all interrupts and call the Os_PanicHook() to inform the application. ▶ Afterwards the OS enters an infinite loop.

Table 2-5 Types of OS Errors

2.13.1 Service Protection Error Handling

If MICROSAR Classic OS Service Protection is enabled by configuration (OsServiceProtection), additional OS Service checks are executed. If an OS service implements a Service Protection check, the according error code is listed in the API description (see chapter 5 API Description).

Reference

All possible error codes and the hook (ErrorHook() or ProtectionHook()) in which they are reported can be looked up in the header file `Os_Types.h`.

The following checks are performed:

2.13.1.1 Access Check

The Access Check verifies, that the calling application has access to the OS object given as parameter to the OS API. The calling application must be part of the AccessingApplication list of the OS object.

The following error code is reported: E_OS_ACCESS

2.13.1.2 Accessibility Check

The Accessibility Check verifies, that the application which owns the OS object given as parameter to the OS API is accessible.

A application is the owner of an OS object if it is referenced in the according OS object list.

A application is accessible after the OS has been initialized (after StartOs() has been called).

A application is not accessible, if it has been terminated. A terminated application can be configured as restartable. After the application has been restarted, a call to the OS API AllowAccess() is needed to make the application accessible again.

The following error code is reported: E_OS_ACCESS

2.13.1.3 Owner Check

The Owner Check verifies, that the calling application is the owner of the OS object given as parameter to the OS API.

The following error code is reported: E_OS_ACCESS

2.13.1.4 Interrupt API Check:

The Interrupt API Check verifies, that the AUTOSAR Interrupt APIs are called correctly regarding order and nesting. The following checks are performed:

The following error code is reported: E_OS_SYS_API_ERROR

Nesting Check:

- Check that DisableAllInterrupts is NOT called after DisableAllInterrupts, SuspendAllInterrupts or SuspendOsInterrupts.
- Check that SuspendAllInterrupts is NOT called after DisableAllInterrupts.
- Check that SuspendOsInterrupts is NOT called after DisableAllInterrupts.
- Check that ResumeAllInterrupts is NOT called after SuspendOsInterrupts (this is also an order check).
- Check that ResumeOsInterrupts is NOT called after SuspendAllInterrupts (this is also an order check).

Order Check:

- Check that EnableAllInterrupts is called after a previous call of DisableAllInterrupts.
- Check that ResumeAllInterrupts is called after a previous call of SuspendAllInterrupts.
- Check that ResumeOsInterrupts is called after a previous call of SuspendOsInterrupts.
- Check that Os_EnableInterruptsPreStart is NOT called after StartOS.

2.14 Error Reporting

MICROSAR Classic OS supports error reporting according to the AUTOSAR [1] and OSEK OS [2] standard.

This includes

- StatusType return values of OS APIs
- Parameter passing of error codes error to ErrorHook()
- Service ID information provided by the macro OSErrorGetServiceId()
- Parameter access macros (e.g. OSError_ActivateTask_TaskID())

2.14.1 Extension of Service IDs

MICROSAR Classic OS introduces new service IDs for own services.

Reference

All service IDs are listed in the OS header file `Os_Types.h` and may be looked up in the enum data type `OSServiceIdType`.

2.14.2 Extension of Error Codes

MICROSAR Classic OS introduces new 8 bit error codes which extend the error codes which are already defined by AUTOSAR OS and OSEK OS standard.

Type of Error	Related Error Code	Value
An internal OS buffer used for cross core communication is full.	E_OS_SYS_OVERFLOW	0xF5
A forcible termination of a kernel object has been requested e.g. terminate system applications.	E_OS_SYS_KILL_KERNEL_OBJ	0xF6
An OS-Application has been terminated with requested restart but no restart task has been configured.	E_OS_SYS_NO_RESTARTTASK	0xF7
The application tries to use an API cross core, but the target core has not been configured for cross core API	E_OS_SYS_CALL_NOT_ALLOWED	0xF8
The triggered cross core function is not available on the target core.	E_OS_SYS_FUNCTION_UNAVAILABLE	0xF9
A syscall instruction has been executed with an invalid syscall number.	E_OS_SYS_PROTECTION_SYSCALL	0xFA
An unhandled interrupt occurred.	E_OS_SYS_PROTECTION_IRQ	0xFB
The interrupt handling API is used wrong.	E_OS_SYS_API_ERROR	0xFC
Internal OS assertion (not issued to customer).	E_OS_SYS_ASSERTION	0xFD

A system timer ISR was delayed too long.	E_OS_SYS_OVERLOAD	0xFE
An OS internal functionality could not be done in time.	E_OS_SYS_TIMEOUT	0xFF

Table 2-6 Extension of Error Codes

Reference
All error codes and their values can be looked up in the header file `Os_Types.h`.

2.14.3 Detailed Error Codes

MICROSAR Classic OS provides detailed error code to extend the standard error handling of AUTOSAR to uniquely identify each possible OS error.

The detailed error code is assembled from AUTOSAR StatusType error code and unique error code.

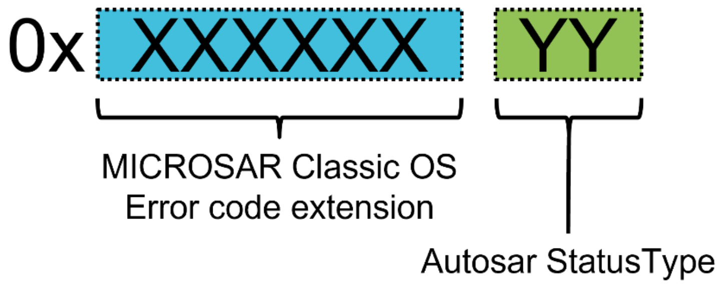

Figure 2-4 MICROSAR Classic OS Detailed Error Code

Within OS Hook routines the error code can be obtained by calling Os_GetDetailedError() (see 5.2.7.1 for details).

Note
Vector OS experts always ask about the detailed error codes when supporting customers in case of OS errors.

Reference
The detailed error codes are listed in the file `Os_Types.h` and may be looked up in the enum data type `Os_StatusType`.

Each detailed error code is preceded by a descriptive comment.

2.15 Multi Core Concepts

2.15.1 Scheduling and Dispatching

MICROSAR Classic OS implements independent schedulers and dispatchers for each core.

2.15.2 Multi Core Data Concepts

The multi core data concept of MICROSAR Classic OS tries to avoid concurrent writing accesses between cores.

Although cores may read all OS data of all cores, write accesses to OS data are only performed locally from the owning core.

This data concept allows optimized linking:

- The data of a particular core may be linked into fast accessible memory
- The data of a particular core may be linked into cached memory

Only the variables related to spinlocks have to be linked into global memory which must be accessible by all participating cores.

2.15.3 X-Signals

To realize cross core service APIs MICROSAR Classic OS offers the X-Signal concept (see 3.9 for details). Mind that the X-Signal concept is used internally by the OS cross core API functions only. X-Signals do not provide a service to the application directly.

2.15.4 Master / Slave Core

In a real master / slave multi core architecture only one core is started upon reset. This is the master core. All other cores are held in a reset state and must be explicitly started by the master core. These are slave cores.

There are also multi core systems which starts all cores simultaneously. There is no hardware master / slave classification.

MICROSAR Classic OS is capable to deal with both concepts. In a system with equal cores the OS emulates master / slave behavior according to the core configurations.

2.15.5 Hardware Init Core

To initialize the system peripherals used by the OS (e.g. System MPU, Interrupt Controller), MICROSAR Classic OS uses a dedicated so called Hardware Init Core.

MICROSAR Classic OS offers the possibility to configure one core as Hardware Init Core ("/MICROSAR/Os/OsOS/OsHardwareInitCore"). If the user does not configure a specific core, the Master Core is used as Hardware Init Core.

In safety-critical environments it is recommended to configure the core with the highest diagnostic coverage as Hardware Init Core.

2.15.6 Startup of a Multi Core System

The startup of a multi core system is described in detail in 2.12.

MICROSAR Classic OS offers the possibility to configure a startup symbol for each core. Within a real master / slave system the OS needs this information for starting the slave cores.

2.15.7 Spinlocks

Synchronization of cores is done by

- OS Spinlocks (see [1]) or
- Optimized spinlocks (see 3.1)

2.15.7.1 Linking of Spinlocks

To achieve freedom from interference between cores with different diagnostic coverage capability, spinlocks are linked into different memory sections.

An MPU may be used to allow access from only specific cores or specific OS applications.

The used memory sections depend on the feature „OsForcibleTermination“

<table>
<tr><th></th><th>OS spinlocks</th><th>Optimized spinlocks</th></tr>
<tr><td>OsForcibleTermination = TRUE</td><td rowspan="2">Spinlocks variables are linked into a core shared section</td><td>Spinlock variables are linked into a core shared section</td></tr>
<tr><td>OsForcibleTermination = FALSE</td><td>Spinlock variables are linked into an application shared section</td></tr>
</table>

Table 2-7 Linking of spinlocks

2.15.8 Cache

Due to cache coherency problems spinlock variables and other application variables which are shared among cores must not be cached.

2.15.9 Shutdown

2.15.9.1 Shutdown of one Core

If ShutdownOS() is called on one core, it induces shutdown actions.

- The core terminates all its applications
- Application specific ShutdownHooks are called
- The global ShutdownHook() is called
- Interrupts are disabled
- An endless loop is entered

2.15.9.2 Shutdown of all Cores

Upon call to ShutdownAllCores() synchronized shutdown of the system is invoked. An asynchronous X-Signal is used for this purpose.

Synchronized shutdown is described in [1].

2.15.9.3 Shutdown during Protection Violation

If the ProtectionHook() returns with “PRO_SHUTDOWN” a shutdown of all cores is invoked.

2.16 Debugging Concepts

2.16.1 Description

MICROSAR Classic OS offers several utilities to support OS debugging.

ORTI	MICROSAR Classic OS generates an ORTI debug file (**O**SEK **R**un**T**ime **I**nterface). Many debuggers are capable of loading such ORTI files and provide comfortable debug means based upon the OS configuration. See chapter 2.16.3 for details.
Timing Hooks	MICROSAR Classic OS provides macros which may be used for debugging purposes (also suitable for third party tools). See chapter 3.10 for details.
ARTI	MICROSAR Classic OS supports the ARTI (**A**UTOSAR **R**un-**T**ime **I**nterface) debugging functionality according to [11]. Which unifies the following debug features in one interface: ▶ Variable tracing (similar to ORTI) ▶ OS State tracing via hooks (comparable to the Timing Hooks) ▶ Service tracing via hooks (one hook for each OS API entry and exit) See chapter 2.16.4 for details.

2.16.2 Activation

ORTI and TimingHooks may be switched on within the following container:

`/MICROSAR/Os/OsOS/OsDebug`

ARTI support can be activated via the following parameters:

- Enable ARTI support in general:

  `/MICROSAR/Os/OsOS/OsUseArti`

- Enable different sets of ARTI hooks:

  `/MICROSAR/Os/OsOS/OsDebug/OsArtiHooks`

  See chapter 2.16.4.1 for details and important notes on ARTI Hooks.

Note
There is an additional switch within the “OsDebug” container. It enables OS assertions. They are intended for OS internal test purposes. If activated the OS performs additional runtime checks on its own internal states.

2.16.3 ORTI Debugging

ORTI is the abbreviation of “OSEK Runtime Interface”.

When ORTI debugging is activated MICROSAR Classic OS generates additional files with “`.ort`” extension. These files contain information about the whole OS configuration. They are intended to be read by a debugger.

The debugger uses the information from the ORTI files to display static and runtime information about OS objects e.g. task states.

ORTI versions supported by MICROSAR Classic OS:

ORTI 2.2	As described in the OSEK standard [3] and [4]
ORTI 2.3	Unofficial “Standard” based upon ORTI 2.2. It does contain extensions for multi core OS and was proposed by “Lauterbach Development Tools” (described in [5]).

Both ORTI versions are capable to be used within single core and multi core systems.

Note for ORTI 2.2 multi core debugging

For each configured AUTOSAR core there is one separate ORTI file.

For multi core debugging, the debugger software must be capable to read several ORTI files.

Note for ORTI 2.3 multi core debugging

The debug information for all configured cores is aggregated in one file.

Note

Basically debuggers are capable to display the stack consumption for each stack (OsStackUsageMeasurement must be switched on).

Please note that uninitialized OS stacks may show 100% stack usage within ORTI debugging. Reliable information can only be given after the OS has initialized all stacks.

Caution

MESSAGE objects and CONTEXT information specified by ORTI 2.2 Standard are not supported in MICROSAR Classic OS.

Caution
The following OS services are not traced by ORTI service tracing:
- GetSpinlock() (for optimized spinlocks)
- TryToGetSpinlock() (for optimized spinlocks)
- ReleaseSpinlock() (for optimized spinlocks)
- IOC API
- Os_GetVersionInfo()
- Os_Init()
- Os_InitMemory()

2.16.4 ARTI Debugging

When ARTI debugging is activated, MICROSAR Classic OS adds relevant data for variable tracing to the ARTI module configuration according to [11]:

`/AUTOSAR/EcucDefs/Arti/ArtiOs`

The same applies when ARTI hook macros are enabled, these are added to:

`/AUTOSAR/EcucDefs/Arti/ArtiValues/ArtiHook`

2.16.4.1 ARTI Hooks

In addition to the ARTI specification, MICROSAR Classic OS provides the possibility to activate or deactivate each set of ARTI hook macros separately. Where a set of hooks is defined by the specified ARTI event classes (e.g. `AR_CP_OS_SPINLOCK`). This is useful to decrease system runtime or code size. To enable a set of hook macros, the corresponding event class shall be added to the `OsArtiHooks` parameter.

ARTI hook prototypes, provided by the application, must match with the specification from [11]. Additionally, the extern declarations of all possible ARTI hook macros must be provided via the file ***arti.h***.

Caution
Enabling or disabling ARTI hook macros via the `OsArtiHooks` parameter only controls the invocation of the corresponding hooks. **However, the extern declarations for all possible ARTI hooks must always be provided by the application via *arti.h*.**

Caution
The code size of the ARTI hook implementations must be kept as small as possible. As these hooks are typically inlined, their code will be copied to several locations in the OS. Especially for service call hooks, this can significantly increase the code size. For more complex tasks within an ARTI hook, an external function call is preferred.

2.17 Memory Protection

MICROSAR Classic OS uses memory protection facilities of a processor to achieve freedom from interference between OS applications and cores. Depending on the platform, the MICROSAR Classic OS uses memory protection units (MPU) or memory management units (MMU). These hardware modules are responsible for monitoring all memory accesses made by CPU and/or peripheral devices and triggering an exception upon detection of an illegal memory access.

As from the OS perspective the configuration of MPUs and MMUs are mostly the same, the descriptions in this chapter are valid for MPUs and MMUs, if not explicitely stated otherwise.

Caution
MICROSAR Classic OS does NOT use memory protection facilities to protect OS Applications, which run in privileged CPU mode, against each other or the OS. An OS-Application is executed in privileged CPU mode, if the parameter OsApplicationIsPrivileged is set TRUE.

2.17.1 Usage of the MPUs / MMUs

The memory protection peripherals (MPUs / MMUs) are used to achieve freedom from interference between applications / tasks / ISRs on the same core. The basic concept is that access rights of these runtime entities (read/write/execute) have to be granted explicitly to software parts.

This is done with the following steps:

Step	Toolchain phase
Set up an SC3 system	Configuration of OS
Configure memory regions	
Assign the memory regions to a memory protection peripheral	
Assign the memory regions to OS applications / Tasks / ISRs (optional)	
Use the AUTOSAR MemMap mechanism to place code, constants and variables into appropriate sections (see 4.4.1.1)	Compilation
Use OS generated linker command files to locate the sections into memory (see 0)	Linkage

Note
Which memory protection peripherals are reserved for use by MICROSAR Classic OS depend on the target hardware. Each memory protection peripheral used by MICROSAR Classic OS can be identified by the corresponding registers that are described in 4.3. Memory protection peripherals reserved for MICROSAR Classic OS may not be used for anything else.

2.17.2 Configuration Aspects

A memory region is typically configured by specifying

> A start and end address by number, or by linker labels (see 4.4.3 for OS generated section labels)

> Access rights to this region (a pre-defined set of access rights is referable)

> The validity of the region by ID (e.g. PID / ASID / Protection Set)

> To which memory protection peripheral the region belongs

> The owner of the region

The owner of the memory region distinguishes the runtime behavior of the memory region in the hardware peripheral (whether the region is static or dynamic).

Note
The start and end addresses of configured memory region should always be a multiple of the granularity of the hardware memory protection peripheral.

Note
If an MPU is used, the number of available hardware MPU regions is limited by hardware!

MICROSAR Classic OS checks during code generation that the overall number of configured memory regions does not exceed the number of available hardware MPU regions.

Caution
For MMUs, the configured regions must not overlap. Further configuration constraints (e.g. granularity and alignment) are hardware dependent and therefore described in [9].

2.17.2.1 Static Memory Regions

If no owner is specified, MICROSAR Classic OS initializes a memory region in the hardware peripheral to be static. It is never reprogrammed during runtime of the OS. It is valid for all software parts.

> **Note**
> The validity of a static memory region may be restricted by using ID (e.g. PID / ASID / ProtectionSet).

2.17.2.2 Dynamic Memory Regions

If an owner is specified for a memory region, MICROSAR Classic OS initializes a hardware memory region in the hardware peripheral to be dynamically reprogrammed during OS runtime. Whenever the owner of the memory is active during runtime a specific memory region in the hardware peripheral is programmed with the configured values of the memory region.

Memory regions which are assigned to an OS application are reprogrammed whenever the OS application is switched.

Memory regions which are assigned to tasks or ISRs are reprogrammed with each thread switch.

2.17.2.3 Freedom from Interference

MICROSAR Classic OS is able to encapsulate OS application data, task private data and ISR private data. This does also depend on the owner of the memory region.

<table>
<tr><th>Memory Region Owner</th><th>Access Granted To</th><th>Access Denied For</th></tr>
<tr><td>OS application</td><td>Runtime objects of this OS application
> Tasks
> ISRs
> IOC callbacks
> Non-trusted functions
> Application specific hooks</td><td>> Other non-trusted OS applications and their objects</td></tr>
<tr><td>Task</td><td>> The owning task</td><td rowspan="2">> Other non-trusted OS applications and their objects
> Other runtime objects of the same OS application</td></tr>
<tr><td>ISR</td><td>> The owning ISR</td></tr>
</table>

2.17.3 Stack Monitoring

MICROSAR Classic OS automatically provides one memory region to monitor the current stack. This is the default handling in SC3 and SC4 systems. See 2.3.5 for details.

Note
If an MPU is used, the memory area where the stacks are linked to has to be configured as read-only region for supervisor- and user-mode.

Caution
Memory regions must not be configured to allow write access into any stack region. Otherwise, the OS cannot ensure stack data integrity.

2.17.4 Protection Violation Handling

Upon any memory protection violation exception the OS first switches to the kernel stack and then informs the application.

In case of a memory protection violation exception which has occurred within OS code MICROSAR Classic OS enters a Kernel Panic.

In case of a memory protection violation exception which has occurred within critical user sections, i.e. PreTaskHook, PostTaskHook, Alarm callbacks, Timing Hooks, ProtectionHook and ShutdownHook, MICROSAR Classic OS calls the PanicHook().

In all other cases of a memory protection violation exception MICROSAR Classic OS calls the ProtectionHook() with the parameter E_OS_PROTECTION_MEMORY.

2.17.5 Optimized / Fast MPU Handling

If the number of application / task / ISR specific memory regions is small, MICROSAR Classic OS may have the possibility to initialize the MPU entirely with static MPU regions.

By utilizing memory protection identifiers different access rights may still be achieved between different applications.

MICROSAR Classic OS switches access rights by simply switching the protection identifier. This will result in a very fast MPU handling.

> Configure only memory regions which are static (no owner is assigned).

> Use “OsMemoryRegionIdentifier” to assign a protection identifier to that region.

> Assign either OS applications or Tasks and ISRs to use a specific protection identifier (OsAppMemoryProtectionIdentifier, OsTaskMemoryProtectionIdentifier, OsIsrMemoryProtectionIdentifier)

Note
Depending on the used platform protection identifiers are also referred as PID (MPC), ASID (RH850) or protection sets (TriCore). But the basic technique is the same.

2.17.6 Recommended Configuration

MICROSAR Classic OS offers a recommended MPU configuration which contains a basic setup.

It configures the MPU to achieve the access rights as follows

Access Rights	Trusted Software	Non-Trusted Software
Executable rights to whole memory	X	X
Read access to whole RAM / ROM	X	X
Write access to whole RAM (except stack regions)	X	-
Read / Write access to peripheral registers	X	-
Read / Write access to global shared memory	X	X
Write access to current active stack	X	X

Table 2-8 Recommended Configuration MPU Access Rights

Caution
As in case of an MMU, the memory regions must not overlap, it is (almost) impossible to configure memory regions which grant access rights to the entire memory (e.g. “read all” or “execute all” regions). It is rather recommended to create small context specific memory regions for the memory a context requires access to with access rights as described in Table 2-8.

2.18 Memory Access Checks

2.18.1 Description

AUTOSAR OS specifies functions for checking memory access rights of an ISR or task to a specific memory region.

> CheckTaskMemoryAccess
> CheckISRMemoryAccess

2.18.2 Activation

No explicit activation of these API service functions necessary. They are provided automatically by the OS.

2.18.3 Usage

The API service functions `CheckTaskMemoryAccess()` and `CheckISRMemoryAccess()` work on additional configuration data which has to be provided by the user.

Therefore additional regions (`OsAccessCheckRegion`) may be configured. Tasks and ISRs may be assigned to each access check region.

Note
All memory access checks are based upon the configured `OsAccessCheckRegion` objects. They are not based upon current MPU values during runtime!
`OsAccessCheckRegions` and `OsMemoryRegions` contain redundant information.

2.18.4 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

2.19 Timing Protection Concept

2.19.1 Description

To implement timing protection, MICROSAR Classic OS needs a timer hardware which is able to generate an interrupt with high priority. This interrupt is never disabled by the OS interrupt handling API.

Two concepts may be implemented within MICROSAR Classic OS:

- The timing protection interrupt request is non-maskable (NMI request)
- The timing protection interrupt request is maskable

The consequences of both concepts are shown in the comparison:

	Timing Protection IRQ is Maskable	Timing Protection IRQ is NMI
Level of timing protection IRQ	The level of the interrupt source is chosen to be higher than the highest category 1 ISR.	The exception source has no interrupt level.

Caution
Any category 1 ISR bypasses the OS. For this reason such an ISR may get terminated in case it is executed, while the budget of a monitored entity is exhausted.

Thus the AUTOSAR OS specification advises not to use category 1 ISRs within a system which uses timing protection.

Caution
In case of an inter-arrival time violation MICROSAR Classic OS does currently not provide the information which task or ISR did violate its inter-arrival time. GetTaskID() and GetISRID() return the current task / ISR. The suppressed task / ISR ID is not returned by these APIs.

2.19.2 Activation

Timing protection features are activated by setting the scalability class to SC2 or SC4 (OsScalabilityClass).

Afterwards timing protection containers may be configured for tasks or ISRs (OsTaskTimingProtection / OsIsrTimingProtection). Observed times are configured within these containers.

Note
The OS will add an appropriate ISR automatically to the configuration.

Caution
To avoid corruption of the OS Timing Protection facility, the Timing Protection ISR must not be delayed longer than one hardware counter cycle. For a 16 Bit hardware timer for instance, one hardware counter cycle is the time needed to count from 0 to 0xFFFF.

2.19.3 Usage

Once the timing protection feature is active tasks and ISRs are observed automatically by the OS.

Observation of a particular OS object (task / ISR) only takes place if any execution budgets or locking times are configured for this object.

2.20 IOC

2.20.1 Description

The Inter OS-Application Communicator (IOC) is responsible for data exchange between OS applications. It handles two important tasks

- Data exchange across core boundaries
- Data exchange across memory protection boundaries

Parts of the IOC API services are generated.

MICROSAR Classic OS always tries to generate IOC API services and data structures to minimize resource usage.

Especially the runtime of IOC API services is influenced by the configuration of IOC objects. For the customer it is important how configuration aspects minimize the IOC runtime.

For each IOC object MICROSAR Classic OS decides during runtime whether

- Interrupt locks
- Spinlocks

Are used or not.

2.20.2 Unqeued (Last Is Best) Communication

Note
Whenever the data of a last is best IOC object can be written / read atomically (integral data type) no spinlocks are used at all.

2.20.2.1 1:1 Communication Variant

	Sender and Receiver are located on the same core	Sender and Receiver are located on the different cores
Interrupt Locks	Used	Not used
Spinlocks	Not Used	Used
System Call Traps	Not Used	Not Used

2.20.2.2 N:1 Communication Variant

	Sender and Receiver are located on the same core	Sender and Receiver are located on the different cores
Interrupt Locks	Used	Not used
Spinlocks	Not Used	Used
System Call Traps	Used	Used

2.20.2.3 N:M Communication Variant

	Sender and Receiver are located on the same core	Sender and Receiver are located on the different cores
Interrupt Locks	Used	Not used
Spinlocks	Not Used	Used
System Call Traps	Used	Used

2.20.3 Queued Communication

For 1:1 and N:1 Communication the following table is applied:

	Sender and Receiver are located on the same core	Sender and Receiver are located on the different cores
Interrupt Locks	Not Used	Not used
Spinlocks	Not Used	Not Used
System Call Traps	Not Used	Not Used

2.20.4 Notification

MICROSAR Classic OS provides configurable receiver callback functions for notification purposes.

Note
In case an IOC object has a configured receiver callback function a system call trap is needed in any case.

2.20.5 Particularities

2.20.5.1 N:1 Queued Communication

N:1 queued communication is realized with multiple sender queues. The receiver application does an even multiplexing on all sender queues when calling the receive function (see figure).

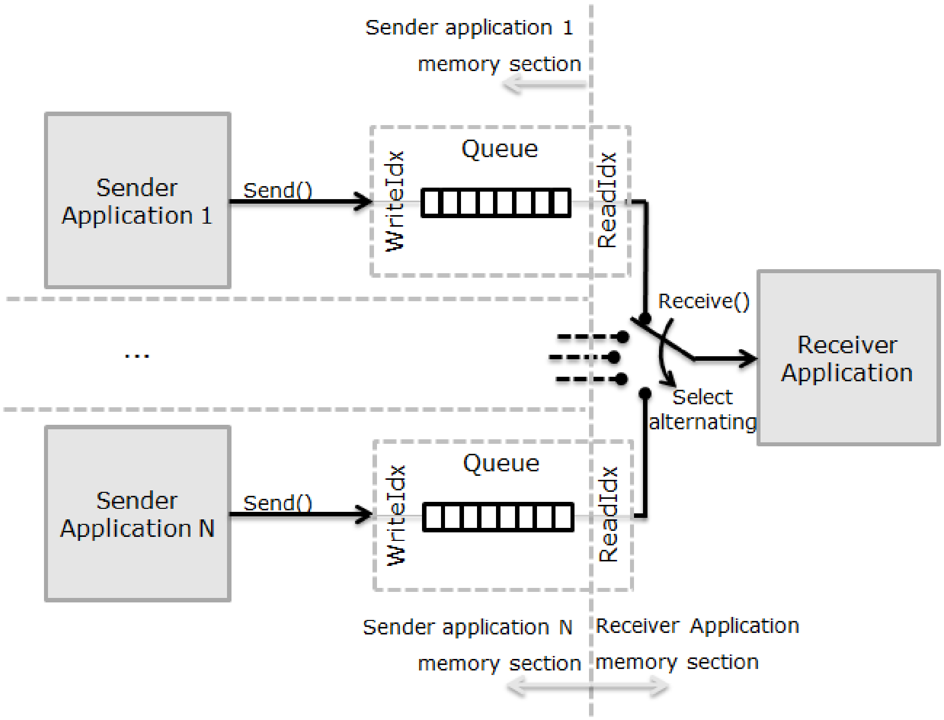

Figure 2-5 N:1 Multiple Sender Queues

2.20.5.2 IOC Spinlocks

Note
During generation of OS data structures, if MICROSAR Classic OS detects that a spinlock is needed for a particular IOC object, it automatically creates a spinlock object within the OS configuration.

2.20.5.3 Notification

Based on the core assignment of sender and receiver of an IOC object, two possible scenarios for callback handling are possible.

Sender and Receiver are located on the same core	> The callback notification function is called within the IOC send function
Sender and Receiver are located on different cores	> The sender triggers an X-Signal request on the receiving core > The callback notification function is called within the X-Signal ISR

Note

- All callback functions are using the cores IOC receiver pull callback stack.
- During execution of the IOC receiver pull callback function category 2 ISRs are disabled.
- Within IOC receiver pull callback functions only other IOC API functions and interrupt dis/enable API functions are allowed.

2.20.5.4 Complex Data Types

Note

If `OsIocDataType` or `OsIocDataTypeRef` of an IOC object is a complex data type, MICROSAR Classic OS uses a memcpy function of the VStdLib Module for data transfer and initialization.

See VStdLib Technical Reference [8].

2.21 Trusted OS Applications

Trusted OS Applications are basically executed in supervisor mode. They can have read/write access to nearly the whole memory (except stack regions).

MICROSAR Classic OS allows to restrict the access rights of trusted OS Applications. Trusted OS Applications may run with memory protection in non-privileged mode.

2.21.1 Trusted OS Applications with Memory Protection

2.21.1.1 Description

Runtime objects (Tasks / ISRs / Trusted functions) of trusted OS applications with enabled memory protection have the following behavior

- They run in user mode
- Memory access has to be granted explicitly (in the same way as for a non-trusted OS application)
- The MPU is re-programmed whenever software of the OS application is executed.

Note

- API runtimes for OS applications which run in user mode are longer.

2.21.1.2 Activation

Set “OsTrustedApplicationWithProtection” to TRUE.

2.21.1.3 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

2.21.2 Trusted Functions

Note
> The interrupt state of the caller is preserved when entering the trusted function.
> The trusted function may manipulate the interrupt state by using OS services. The changed interrupt state is preserved upon return from the trusted function.
> The trusted function is executed with the memory protection and processor mode settings of the owner Application.

Caution
Trusted functions of OS Applications with protection have limited memory access rights but have still full access to the stack of the caller.

Caution
Nesting level of trusted functions is limited to 255.

The application has to ensure that this limitation is held. There is no error detection within the OS.

2.22 OS Hooks

2.22.1 Runtime Context

MICROSAR Classic OS implements the runtime context and accessing rights of OS Hooks according to the following table

<table>
<tr><th>Hook Name</th><th>Processor Mode</th><th>Access Rights</th><th>Interrupt State</th></tr>
<tr><td>StartupHook</td><td rowspan="4">Supervisor</td><td rowspan="4">Trusted</td><td rowspan="2">Category 2 lock level</td></tr>
<tr><td>ErrorHook</td></tr>
<tr><td>ShutdownHook</td><td rowspan="2">TP lock level</td></tr>
<tr><td>ProtectionHook</td></tr>
<tr><td>StartupHook_<OS application name></td><td colspan="2" rowspan="3">Depending on the configuration of the owning OS application</td><td rowspan="2">Category 2 lock level</td></tr>
<tr><td>ErrorHook_<OS application name></td></tr>
<tr><td>ShutdownHook_<OS application name></td><td>TP lock level</td></tr>
<tr><td>Os_PanicHook</td><td>Supervisor</td><td>Trusted</td><td>TP lock level</td></tr>
<tr><td>PreTaskHook</td><td>Supervisor</td><td>Trusted</td><td>TP lock level</td></tr>
<tr><td>PostTaskHook</td><td>Supervisor</td><td>Trusted</td><td>TP lock level</td></tr>
<tr><td>AlarmCallbacks</td><td>Supervisor</td><td>Trusted</td><td>Category 1 lock level</td></tr>
<tr><td>IOC receiver pull callbacks</td><td colspan="2">Depending on the configuration of the owning OS application</td><td>Category 2 lock level</td></tr>
</table>

2.22.2 Nesting behavior

It is possible that OS hooks may be nested by other OS hooks according to the following table

Nested by OS Hook	ErrorHook(s)	ProtectionHook	StartupHook(s)	ShutdownHook(s))	IOC Callbacks
ErrorHook(s)	Not possible	possible	Not possible	possible	possible
ProtectionHook	Not possible	Not possible	Not possible	possible	possible
StartupHook(s)	possible	possible	Not possible	possible	possible
ShutdownHook(s)	Not possible	Not possible	Not possible	Not possible	possible
IOC Callbacks	possible	possible	Not possible	possible	Not possible

2.22.3 Hints

Caution
Within OS Hooks the interrupts must not be enabled again!

Caution
Hooks must never be called by application code directly.

Note for SC2 or SC4
Hooks don’t have any own runtime budgets. OS Hooks consume the budget of the current task / ISR.

Caution: Protection violations during Pre- and PostTaskHook

- In case of a memory violation during execution of Pre-/PostTaskHook, the OS will always end up in PanicHook.
- In case of an unhandled exception or an unhandled interrupt during execution of Pre-/PostTaskHook, the OS will always end up in PanicHook.
- After termination of a task a timing violation in the according PostTaskHook could not be detected by the OS.

3 Vector Specific OS Features

This chapter describes functions which are available only in MICROSAR Classic OS. They extend the standardized OS functions from the AUTOSAR and OSEK OS standard [1] [2].

3.1 Optimized Spinlocks

3.1.1 Description

For core synchronization in multi core systems, MICROSAR Classic OS offers (beneath the AUTOSAR specified OS spinlocks) additional optimized spinlocks.

They are able to reduce the runtime of the Spinlock API. Configuration is also easier.

AUTOSAR specified OS spinlocks cannot cause any deadlocks between cores (see unique order of nesting OS spinlocks in AUTOSAR OS standard). Therefore some error checks on OS configuration data are necessary.

The error checks are not performed with optimized spinlocks.

	OS Spinlocks	Optimized Spinlocks
Deadlocks	No deadlocks possible	Deadlocks are possible
Runtime	Longer runtime due to more error checks	Smaller runtime due to less error checks
Configuration	OsSpinlockSuccessor must be configured if spinlocks must be nested	OsSpinlockSuccessor need not to be configured
Nesting	Can be nested by other OS spinlocks	Nesting of optimized spinlock should be avoided or at least be used with caution
Non-Forcible Thread Termination	Automatically released if still locked by terminating thread	Not released automatically
Linking	OS and optimized spinlock variables are placed into different dedicated memory sections (see 4.4.1).	

Table 3-1 Differences of OS and Optimized Spinlocks

3.1.2 Activation

The spinlock attribute “OsSpinlockLockType” may be set to “OPTIMIZED”.

The “OsSpinlockSuccessor” attribute should not be configured for an optimized spinlock.

3.1.3 Usage

Once a spinlock object is configured to be an optimized spinlock the application may use the Spinlock API as usual. The Spinlock service functions are capable to deal with optimized and OS spinlocks.

3.2 Barriers

3.2.1 Description

MICROSAR Classic OS offers a feature to synchronize tasks from different cores using a barrier object. The calling task of the synchronization API method blocks until all other tasks participating in the same barrier have also called the synchronization method.

3.2.2 Activation

Within OS configuration “Barrier” objects may be specified. A barrier consists of a list of tasks which participate in the barrier.

> **Note**
> Only one task per core may be assigned to a barrier object. The assigned task must also be the task that calls the API.

3.2.3 Usage

If one or more barriers are configured, participating tasks may call Os_BarrierSynchronize() with a BarrierID they are participating in. A task can participate in multiple barriers.

Multiple tasks of one core may not participate in the same barrier.

After a task calls Os_BarrierSynchronize() for a specific barrier, it blocks until all other participants of the same barrier have also called Os_BarrierSynchronize() with the same BarrierID.

If a participating core has not been started, the participating task of that core will not be considered as participant of the barrier.

> **Note**
> Os_BarrierSynchronize() does not disable the interrupts internally. Therefore, higher priority threads may preempt the calling task.
>
> Threads with lower priority will not be executed until the synchronization is complete.
>
> If the core should stop execution until the barrier synchronization is completed, the user has to disable the interrupts before calling the API.

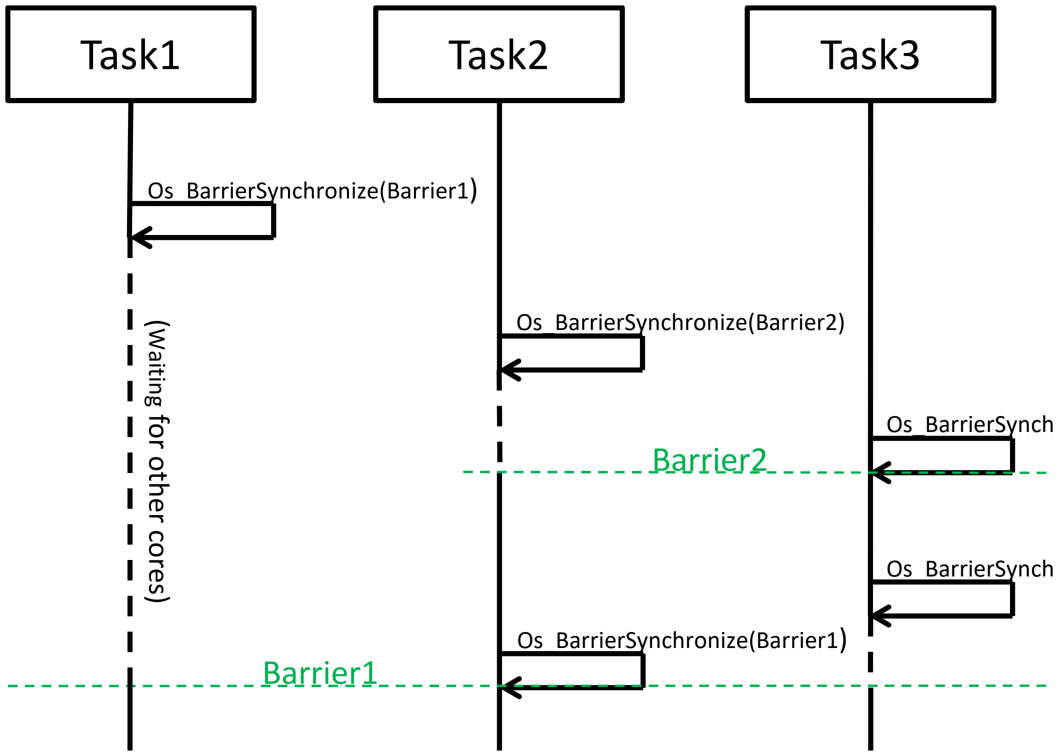

Figure 3-1 Barriers

Caution
A deadlock may occur if one task has called Os_BarrierSynchronize() and one of the other participants does not call Os_BarrierSynchronize() for the same barrier. All participants must call the API method the same number of times to ensure deadlock free scheduling.

Caution
A deadlock occurs when the API method is called for a barrier of which one of the participants was killed.

3.3 Peripheral Access API

3.3.1 Description

MICROSAR Classic OS offers peripheral access services for manipulating registers of peripheral units. The application may delegate such accesses to the OS in case that its own accessing rights are not sufficient to manipulate specific peripheral registers.

3.3.2 Activation

The API service functions themselves do not need any activation.

But within the OS configuration “OsPeripheralRegion” objects may be specified. They are needed for error and access checking by the OS.

An OsPeripheralRegion object consists of the start address, end address and a list of OS applications which have accessing rights to the peripheral region.

> **Note**
> Access to a peripheral region is granted if the following constraint is held
> Start address of peripheral region <= Accessed address <= End address of peripheral region

3.3.3 Usage

Once peripheral regions are configured they may be passed to the API functions.

> **Reference**
> The API service functions themselves are described in chapter 5.2.2.

3.3.4 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

3.3.5 Alternatives

The access rights to peripheral registers may also be granted by configure an additional MPU region for the accessing OS application.

3.3.6 Common Use Cases

The peripheral access APIs may be used …

> … if the accessing OS application runs in user mode but the register to be manipulated can only be accessed in supervisor mode.

> … if the application does not want to spend a whole MPU region to grant access rights.

3.4 Trusted Function Call Stubs

3.4.1 Description

Since the OS service CallTrustedFunction() is very generic, there is the need to implement a stub-interface which does the packing and unpacking of the arguments for trusted functions.

MICROSAR Classic OS is able to generate these stub functions.

3.4.2 Activation

The OS application attribute “OsAppUseTrustedFunctionStubs” must be set to TRUE. Data types must be defined in the header file which is referred by “OsAppCalloutStubsIncludeHeader”.

3.4.3 Usage

A particular trusted function is called with the following syntax:

```
<configured return type> Os_Call_<trusted function name>
(<configured parameters>);
```

Parameter packing, unpacking and return value handling is done by the stub function.

3.4.4 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

3.5 Non-Trusted Functions (NTF)

3.5.1 Description

Service functions which are provided by non-trusted OS applications are called non-trusted functions. They have the following characteristics:

> They run in user mode.

> They run with the MPU access rights of the owning OS application.

> They perform a stack switch to specific non-trusted function stacks.

> They run on an own secured stack.

> They can safely provide non-trusted code to other OS applications.

> Parameters are passed to the NTF with a reference to a data structure provided by the caller.

> Returning of values is only possible if the caller passes the non-trusted functions parameters as pointer to global accessible data.

> If the ProtectionHook returns with PRO_TERMINATEAPPL / PRO_TERMINATEAPPL_RESTART in a NTF, the caller application is forcibly terminated, rather than the application that provides the NTF (equally to trusted functions). In case of nested calls to trusted / non-trusted functions, the first caller application is forcibly terminated.

3.5.2 Activation

They are defined within an OsApplication container (“OsApplicationNonTrustedFunction”). The attribute “OsTrusted” for this OS application must be set to FALSE.

3.5.3 Usage

Similar to the CallTrustedFunction() API of the AUTOSAR OS standard MICROSAR Classic OS implements an additional service which is called Os_CallNonTrustedFunction() (see chapter 5.2.4 for Details).

Configured non-trusted functions are called with this API.

Note

> The interrupt state of the caller is preserved when entering the non-trusted function

> The non-trusted function may manipulate the interrupt state by using OS services. The changed interrupt state is preserved upon return from the non-trusted function.

Caution

Non-trusted functions currently cannot be terminated without termination of the caller.

3.5.4 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

3.6 Fast Trusted Functions

3.6.1 Description

MICROSAR Classic OS offers the feature of runtime optimized trusted functions (fast trusted functions).

The speedup of the runtime is achieved by removing most of the OS error checks, the application switch and the MPU reprogramming.

Fast trusted functions have the following characteristics:

> They may be called with disabled interrupts.
> They run in supervisor mode.
> They run with the application ID of the caller.
> They run on the stack of the caller.
> They run with the MPU settings of the caller.
> Parameters are passed to the fast trusted function with a reference to a data structure provided by the caller.

Caution
Calls to other OS API services are not allowed within a fast trusted function!

3.6.2 Activation

They are defined within an OsApplication container (“OsApplicationFastTrustedFunction”). The attribute “OsApplicationIsPrivileged” for this OS application must be set to TRUE.

3.6.3 Usage

Similar to the CallTrustedFunction() API of the AUTOSAR OS standard MICROSAR Classic OS implements an additional service which is called Os_CallFastTrustedFunction() (see chapter 5.2.5 for Details).

Configured fast trusted functions are called with this API.

3.6.4 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

3.7 Interrupt Source API

3.7.1 Description

MICROSAR Classic OS offers additional API services for category 2 ISRs and their respective interrupt sources.

The services include

> Enable of an interrupt source
> Disable of an interrupt source
> Clearing of the interrupt pending bit
> Checking if the interrupt source is enabled
> Checking of interrupt pending bit status
> Enabling of all interrupt sources

(See 5.2.6 for API details).

3.8 Pre-Start Task

3.8.1 Description

MICROSAR Classic OS offers the possibility to provide a set of OS API functions for initialization purposes before StartOS() has been called.

Therefore a pre-start task may be configured which is capable to run before the OS has been started. Within this task stack protection is enabled and particular OS APIs can be used.

The table in 5.2.15 lists the OS API functions which may be used within the Pre-Start task.

3.8.2 Activation

> Define a basic task and assign the task to a trusted and privileged application.

> Within a core object this basic task has to be referred to be the pre-start task of this core (attribute “OsCorePreStartTask”). Only one pre-start task per core is possible.

> Start the OS as described below

3.8.3 Usage

1. Execute Startup Code
2. Call `Os_InitMemory()`
3. Call `Os_Init()`
4. Call `Os_EnterPreStartTask()` (see 5.2.3 for Details)
5. The OS schedules and dispatches to the task which has been referred as pre-start task.
6. The pre-start task has to be left by a call to `StartOS()`

Caution
The pre-start task may only be active once prior to StartOS() call.

Caution
Within the pre-start task the getter OS API services (e.g. GetActiveApplicationMode()) neither return a valid result nor a valid error code.

Caution
If MICROSAR Classic OS encounters an error within the pre-start task, only the global hooks (ErrorHook(), ProtectionHook() and ShutdownHook()) are executed. OS application specific hooks won't be executed.

Consider that the StartupHook() did not yet run when the Pre-Start Task is executed.

Caution
If the Pre-Start Task is used, global hooks have to consider that the OS might not be completely initialized. OS APIs which are allowed after normal initialization (e.g. TerminateApplication()) are not allowed within global hooks, if the error occurred in the Pre-Start Task.

Caution
If the ProtectionHook() is triggered within the Pre-Start Task, the OS ignores its return value. The only valid return value is PRO_SHUTDOWN.

3.8.4 Dependencies

This feature is of significance in SC3 and SC4 system with active memory protection.

3.9 X-Signals

3.9.1 Description

MICROSAR Classic OS uses cross core signaling (X-Signals) to realize API service calls between cores.

Note
X-Signals are used internally in the OS. The OS uses them to make the usual OS API-services (like task activation, event setting, alarm start, …) work cross core. In order to achieve that, the integrator has to configure (and understand) X-Signals although they do not directly provide any additional services to the application.

The next figure shows the basic principles of an X-Signal

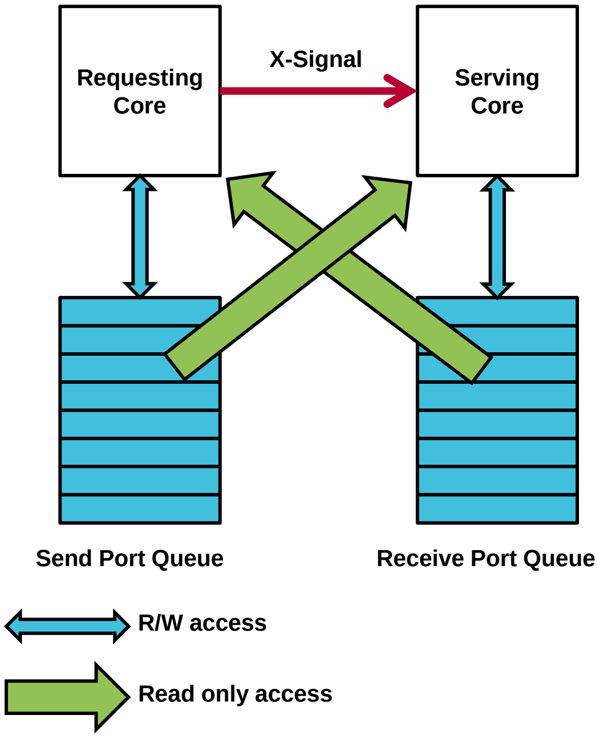

Figure 3-2 X-Signal

Whenever a core executes a service API cross core it writes this request into its own send port queue. Then it signals this request by an interrupt request (X-Signal) to the serving core.

The serving core reads the request from the send port queue and executes the requested service API. The result of the service API is provided in the receive port queue.

X-Signals have the following characteristics:

> An X-Signal is a unidirectional request from one core to another (1:1).

> For each core interconnection one X-Signal is needed.

> All accesses to the (sender / receiver) port queues are lock free.

> Queue Sizes must be configured.
> The Queues may be protected by MPU to achieve freedom of interference between cores.
> X-Signals may be configured to offer only a subset of possible cross core API services. Not configured API services are refused to be served.
> The API error codes for cross core API services are extended.
  - Additional error codes for queue handling.
  - Additional error code if the requested service is refused to be served.
> X-Signals can be configured to be synchronous or asynchronous.

	Synchronous X-Signal	Asynchronous X-Signal
Call behavior	After the cross core service API has been requested the requester core goes into active waiting loop and polls for the result from the server core (remote procedure call). **Note:** During active wait the interrupts are enabled.	After the cross core service API has been requested the requester core continues its own program execution.
Error signaling	Error handling is induced on the requester core immediately, if the polled API result is not E_OK.	Error handling is induced with the next X-Signal request on the requester core, if the result of the previously requested API is not E_OK. **Note**: Upon potential errors of the previously requested API the current application ID on sender and receiver side meanwhile may have changed.
AUTOSAR standard compliance	Compliant to the AUTOSAR Standard	Deviation to the AUTOSAR Standard

Table 3-2 Comparison between Synchronous and Asynchronous X-Signal

Note
Any cross core “getter” APIs e.g. GetTaskState() are always executed with a synchronous X-Signal.

Note
The sender core as well as the receiver core may cause protection violations. Protection error handling is performed on the core where the violation is detected.

Note
When a cross core API is induced by an X-Signal, all static error checks (e.g. validity of parameters) are done on the caller side.

All dynamic error checks (which depend on runtime states) are executed on the receiver side.

Caution
For correct X-Signal function it is essentially that a sender core of an X-Signal must have read access to the receiver core data structure. Especially if the data is mapped into core local RAM.

There are some platforms e.g. RH850 which does not grant cross core read access to core local RAM out of reset. Within such platforms it is the duty of the application to set up these cross core read accesses before the OS is started.

3.9.1.1 Notes on Synchronous X-Signals

The priority of the receiver ISR determines which other category 2 ISRs of one core may use cross core API services and are allowed to call GetSpinlock().

Additionally category 2 ISRs may only use cross core API services and call GetSpinlock() if they allow nesting. Note that an ISR that gets a spinlock with an interrupt blocking "OsSpinlockLockMethod" is not nestable.

The following table gives an overview.

Logical Priority	ISR Nesting	Synchronous Cross Core API Calls / Calling GetSpinlock()
ISR with higher priority than X-Signal priority	ISR nesting is allowed	Not allowed
ISR with higher priority than X-Signal priority	ISR nesting is disabled	Not allowed
X-Signal ISR priority	-	-
ISR with lower priority than X-Signal priority	ISR nesting is allowed	Allowed
ISR with lower priority than X-Signal priority	ISR nesting is disabled	Not allowed

Table 3-3 Priority of X-Signal receiver ISR

Caution
If the priority and nesting requirements from the previous table are not fulfilled there may be deadlocks within a multicore system!

3.9.1.2 Notes on Mixed Criticality Systems

MICROSAR Classic OS checks application access rights on sender and on receiver side. This increases isolation of safety-critical parts in mixed criticality systems (e.g. protect a lockstep core from a non-lockstep core).

Consider that these application access checks are not performed for ShutdownAllCores(). Thus switching off the usage of ShutdownAllCores API for non-lockstep cores is recommended. This can be done within the X-Signal configuration.

3.9.2 Activation

X-Signals must be configured explicitly in a multi core environment. See chapter 4.6 for details.

3.10 Timing Hooks

3.10.1 Description

MICROSAR Classic OS supports timing measurement and analysis by external tools. Therefore it provides timing hooks. Timing hooks inform the external tools about several events within the OS:

- Activation and arrival of a task:
  - These allow an external tool to trace all activations of tasks as well as further arrivals (e.g. setting of an event or the release of a semaphore with transfer to another task).
  - This allows external tools to visualize activations and arrivals of tasks to measure the time between them in order to do a schedulability analysis.
- Context switch:
  - These allow an external tool to trace all context switches between tasks and ISRs.
  - This allows external tools to visualize the scheduling of tasks and ISRs and measure their execution time.
- Locking of interrupts, resources or spinlocks:
  - These allow an external tool to trace locks. This is important as locking times of tasks and ISRs influence the execution of other tasks and ISRs. The kind of influence is different for different locks.
- Forcible termination of tasks and ISRs:
  - These allow an external tool to trace killing of tasks and ISRs. So abnormal behavior of the application can be monitored (e.g. timing violations by a task or ISR).

The timing hooks are called within MICROSAR Classic OS code. For hooks, which are not implemented by the user, empty hook definitions are provided.

3.10.2 Activation

An include header has to be specified in the attribute “OsTimingHooksIncludeHeader” located in the “OsDebug” container.

3.10.3 Usage

The timing hooks may be implemented in the configuration specified header. All available macros are introduced in chapter 5.2.12.

Caution
Within the timing hooks trusted access rights are active e.g. access rights to OS variables.

Note: Protection violations during Timing Hooks
If any protection violation occurs during any of the timing hooks the OS will always go into shutdown!

The return value of the ProtectionHook (e.g. PRO_TERMINATEAPPL) will be ignored and overwritten by the OS to PRO_SHUTDOWN.

3.11 Kernel Panic

If MICROSAR Classic OS recognizes an inconsistent internal state it enters the kernel panic mode. In such cases, the OS does not know how to correctly continue execution. Even a regular shutdown cannot be reached. E.g.:

- The protection hook itself causes errors
- The shutdown hook itself causes errors

MICROSAR Classic OS goes into freeze as fast as possible

1. Disable all interrupts
2. Inform the application about the kernel panic by calling the Os_PanicHook() (see 5.2.13)
3. Enter an endless loop

Caution

- The OS cannot recover from kernel panic.
- ProtectionHook() is not called
- ErrorHook() is not called
- There is no stack switch. The Os_PanicHook() runs on the current active stack

3.12 Generate callout stubs

3.12.1 Description

MICROSAR Classic OS offers the feature to generate the function bodies of all configured OS hook functions (all global hooks and application specific hooks).

The function bodies are generated into the file “Os_Callout_Stubs.c”.

3.12.2 Activation

The Configuration attribute “OsGenerateCalloutStubs” has to be set to TRUE.

3.12.3 Usage

Once the C-File has been generated it may be altered by the user. Code parts between certain special comments are permanent and won’t get lost between two generation processes.

If a hook is switched off, the corresponding function body is also removed. But the user code (between the special comments) is preserved. Once the hook is switched on again, the preserved user code is also restored.

Example

```
FUNC(void, OS_STARTUPHOOK_CODE) StartupHook(void)
{
/**********************************************************************
 * DO NOT CHANGE THIS COMMENT!           <USERBLOCK OS_Callout_Stubs_StartupHook>
 *********************************************************************/

   /* user code starts here */
   /* code between those comments is preserved even if the file is newly generated
      Or even if the hook is switched off in the meanwhile */

/**********************************************************************
 * DO NOT CHANGE THIS COMMENT!           </USERBLOCK>
 *********************************************************************/

}
```

3.13 Exception Context Reading and Manipulation

3.13.1 Description

MICROSAR Classic OS offers the feature to read and modify the interrupted context in case of a hardware exception. This feature shall be applied in ProtectionHook in the combination with PRO_IGNORE_EXCEPTION as the return value. One typical use case for this feature is to recover from an ECC error in memory.

Reading of the interrupted context can also be used for debugging and logging. Thus the user is able to gather specific information about the interrupted context after an exception occurred.

3.13.2 Usage

The following figure shows the usage of this feature.

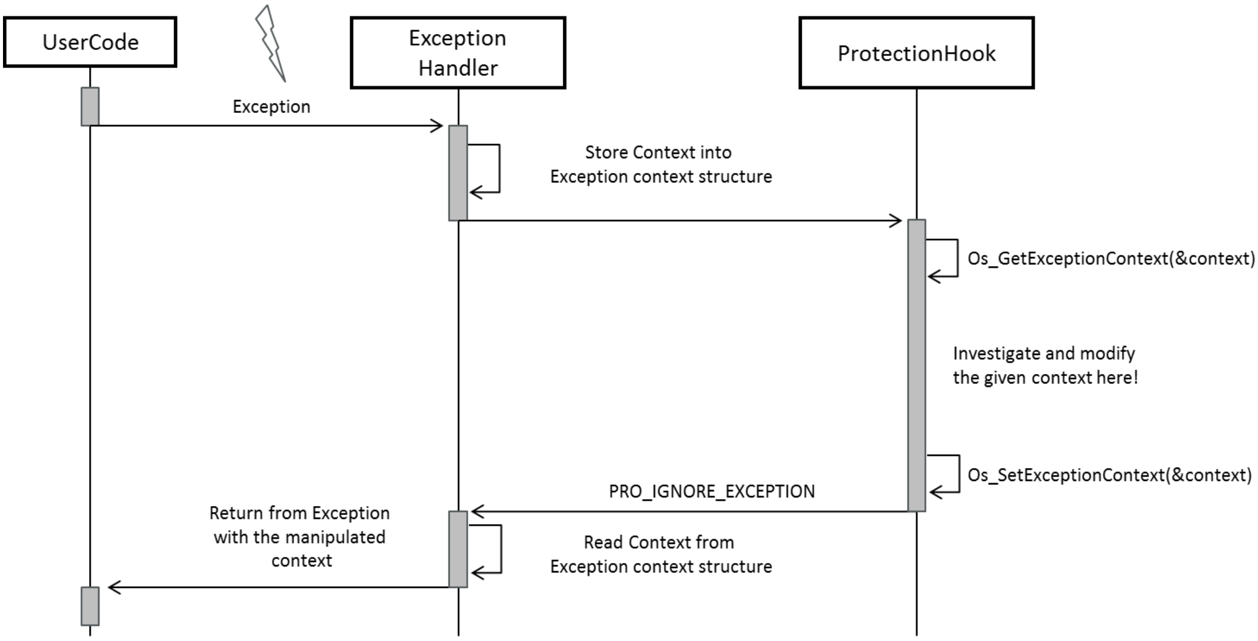

Figure 3-3 Usage of manipulating exception context

Inside ProtectionHook the user first needs to call Os_GetExceptionContext() to read the previous context. Then the context may be investigated and modified according to user requirements. For instance, the program counter may be adapted to the instruction, which is to be executed directly after the exception. Note that the content of the context is depending on the platform. In general, the context contains all the processor registers and some other relevant information. More detailed information can be found in the static code, where the type Os_ExceptionContextType is defined. Finally, the modified context can be written back via Os_SetExceptionContext(). When ProtectionHook returns with PRO_IGNORE_EXCEPTION, the processor continues its execution with the manipulated context.

Note
Currently this feature isn't available for all derivatives supported by the MICROSAR Classic OS. On some derivatives only reading of the interrupted context is possible. For more information please refer to the platform specific Technical Reference [9].

Caution
Exception context manipulation may only be used within the ProtectionHook when the error status is either E_OS_PROTECTION_MEMORY or E_OS_PROTECTION_EXCEPTION.

Caution
The MICROSAR Classic OS is not able to guarantee the correctness of the context structure for each possible situation. Thus, the user of MICROSAR Safe needs to assure the functionality for his use case on his own. For this, the values of the members of the structure Os_ExceptionContextType need to be checked for correctness. In case of Os_SetExceptionContext usage, the register values after return need to be checked against the values in the context structure.
Os_GetExceptionContext usage can be checked by preparing a context, triggering an exception and checking the returned values by a runtime test or with the help of a debugger. For Os_SetExceptionContext a breakpoint may be set to the return address in order to check the restored register values against the values in the context structure.

3.14 Category 0 Interrupts

3.14.1 Description

MICROSAR Classic OS implements category 0 ISRs to have minimal interrupt latency time especially in SC2 or SC4 systems. This is an extension to the OS standard.

3.14.2 Usage

3.14.2.1 Implement Category 0 ISRs

MICROSAR Classic OS offers a macro for implementing a category 0 ISR. This is a similar mechanism like the macro for a category 2 ISR defined by the AUTOSAR standard.

MICROSAR Classic OS abstracts the needed compiler keywords.

Implement a category 0 ISR

```
OS_ISR0(<MyCategory0ISR>)
{
}
```

3.14.2.2 Nesting of Category 0 ISRs

Since category 0 ISRs are directly called from interrupt vector table without any OS pro- and epilogue, automatic nesting of category 0 ISRs cannot be supported.

The configuration attribute “OsIsrEnableNesting” is ignored for category 0 ISRs.

Nevertheless the interrupts may be enabled during a category 0 ISR to allow interrupt nesting but OS API functions cannot be used for this purpose. The application has to use compiler intrinsic functions or inline assembler statements.

Example

```
OS_ISR0(<MyCategory0ISR>)
{
   __asm(EI); /* enable nesting of this ISR */

   __asm(DI); /* disable nesting before leaving the function */
}
```

3.14.2.3 Category 0 ISRs before StartOS

There may be the need to activate and serve category 0 ISRs before the OS has been started.

The following sequence should be implemented:

> Call Os_InitMemory()

> Call Os_Init() (within the function the basic interrupt controller settings are initialized e.g. priorities of interrupt sources).

> Enable the interrupt sources of category 0 ISRs by directly manipulating the control registers in the interrupt controller.

> Enable the interrupts by directly manipulating the global interrupt flag and / or current interrupt priority to allow the category 0 ISRs

3.14.2.4 Locations where category 0 ISRs are locked

Category 0 interrupts are disabled OS internally for very short times only.

The following list mentions the locations of these locks:

> Inside APIs that cause a context switch e.g. TerminateTask()

> Partial termination due to exception handled by ProtectionHook

> On Interrupt, Exception and Trap entry and return

> OS initialization routines inside Os_Init() and StartOS()

3.14.3 Notes on Category 0 ISRs

Expert Knowledge
On platforms which have no automatic stack switch upon interrupt request there will be no stack switch at all if a category 0 ISR occurs. Thus the stack consumption of a category 0 ISR should be added to all stacks which can be consumed by category 0 ISRs (see 2.2.14 for an overview).

Expert Knowledge
Category 0 ISRs are consuming timing protection budgets (execution budgets and locking times) of the interrupted Task or category 2 ISR

Note
Although the interrupt priorities are initialized by MICROSAR Classic OS there is no API to enable or acknowledge category 0 ISRs. The interrupt control registers have to be accessed directly.

Caution
If the timing protection interrupt occurs during the runtime of a category 0 ISR, its execution (the timing protection violation handling/protection hook) is delayed until the category 0 ISR has finished.

Caution
MICROSAR Classic OS does not allow OS API usage within category 0 ISRs.

If any OS API is called anyway, MICROSAR Classic OS is not able to detect this and the called API may not work as expected.

Caution
Category 0 ISRs are always executed with trusted rights on supervisor level.

Caution
A category 0 ISR may never lower the interrupt priority of the CPU or the interrupt controller.

Caution
Category 0 ISRs may still occur in case of a shutdown of the OS or even in case the OS has entered the panic hook.

Caution
Be aware that a category 0 ISR will interrupt category 2 ISRs even if they are configured to be non-nestable!

Caution
If the owner application of a category 0 ISR is terminated for any reason, assigned category 0 ISRs are not disabled.

Caution
The macro “OS_ISR0” abstracts the appropriate compiler keyword for implementing the interrupt service routine. Thus the compiler generates code which saves and restore a subset of the general purpose registers.

In certain use cases e.g. usage of the FPU or nested interrupts it may require the user application to save and restore more registers.

3.15 Floating Point Context Extension

3.15.1 Description

If several tasks or ISRs use FPU operations, there is the need to save
and restore dedicated FPU registers upon context switches.
e.g. If a task, which uses the FPU, is preempted by another task or ISR which also uses the FPU as well.

MICROSAR Classic OS offers the feature to configure save and restoration of the related floating-point registers upon context switch.

3.15.2 Usage

The parameter OsFpuUsage determines the scale of the feature:

> ALL: Dedicated FPU registers are saved upon each context switch

> INDIVIDUAL: Dedicated FPU registers are saved only for selected tasks or ISRs

> NONE: No dedicated FPU registers are saved upon context switches

The FPU configuration must be already set up by the user for each core before Os_Init() is called.

Note
On platforms with dedicated FPU registers, the OsFpuUsage values ALL and INDIVIDUAL require additional memory and runtime for FPU context handling.

See [9] for details.

Caution
If OsFpuUsage is enabled by configuration, the hardware FPU support must be enabled by appropriate compiler options on some platforms too.

3.16 User defined processor state

3.16.1 Description

MICROSAR Classic OS offers the user the possibility to change the processor state according to his needs by altering the flags which are NOT under control of the OS.
The OS never changes such flags, but it saves and restores them during a context switch.

> **Note**
> State register flags which are under control of the OS can be looked up in the corresponding platform HSI chapter (see 4.3).

3.16.2 Usage

The processor state register should be initialized by the user before Os_Init() is called. MICROSAR Classic OS will transfer the settings into every new context.

Thus, if an ISR interrupts an OS Task, the ISR runs with the settings of the interrupted Task, as these are simply transferred to the new context. Nevertheless, MICROSAR Classic OS saves the current processor state of the Task and restores it completely after the ISR returns. This means that even if the ISR changes the according flags, they are not transferred back to the task.

It is recommended to never change the preconfigured values of the user bits during runtime of the OS. If this is nevertheless required, the user has to ensure the correctness of the flags for each Task, ISR and Hook.

3.17 Interrupt Mapping

3.17.1 Description

MICROSAR Classic OS offers the user the possibility to map certain interrupts to a hardware defined type. These interrupts are routed to the respective hardware specific interrupt controller.

3.17.2 Usage

The optional parameter OsIsrInterruptMapping can be used to configure a mapped interrupt. By default, this parameter is left empty and thus no mapping does apply.
More details about interrupt mapping and its configuration can be found in [9].

3.18 Time Slice Scheduling

3.18.1 Description

MICROSAR Classic OS offers an additional time slice scheduling strategy for tasks on the same priority level besides the FIFO strategy specified by AUTOSAR. With this method, tasks will be scheduled in a round robin like manner at configurable points in time.

3.18.2 Activation

Time slice scheduling can be enabled for a task by adding the container OsTaskRoundRobinScheduling to it. This container holds the parameter to configure the number of time slices a task may consume. To activate time slice scheduling, it is required to add exactly one alarm with the action OsAlarmScheduleEventRoundRobin for each core, which has round robin tasks configured.

3.18.3 Usage

Time slice scheduling uses an alarm whose callback function increases the number of time slices the currently running task consumed. After the task consumed the configured amount of time slices scheduling will take place. Therefore, time slice scheduling will only work if the alarm is set up to trigger time slicing events cyclically and has been started.

If a task uses time slice scheduling and its attribute OsTaskSchedule is set to NON or the task uses an internal resource, scheduling will not happen automatically. In these cases it is required to call the OS API Schedule(). However, time slice scheduling will still only take place if the task consumed all of its configured time slices.

If the number of configured time slices is consumed while the task holds a resource or a spinlock with lock method LOCK_WITH_RES_SCHEDULER, scheduling will not happen immediately. It will be delayed until the task releases the resource/spinlock.

Note
When enabling time slice scheduling for a task all tasks that are located on the same core and use the same priority level also have to use time slice scheduling.

Caution
Time slice scheduling as implemented in MICROSAR Classic OS is not meant to provide exact execution budgets to the round robin tasks. It only just counts time slicing events for the currently running task and performs scheduling if the configured number has been reached. This means:

- Runtime of ISRs is accounted for the interrupted task.
- Runtime of preempting tasks is accounted for the preempted task if the preemption is short enough that no time slicing event occurs meanwhile.
- A full time slicing event is accounted for the currently running task even if it has just become active and other tasks have consumed most of the time between the last and the current time slicing event.
- Disabling interrupts may delay time slicing events.

3.18.3.1 Usage with OS resources

OS resources are commonly used to implement critical sections. With a call of GetResource(), the OS will increase the running priority of the calling task to the highest priority of all tasks which are configured to use the resource. With the AUTOSAR standard scheduling strategy, this will effectively prevent any other task, which is configured to use the same OS resource, to enter the running state. With time slice scheduling however, a round robin scheduling is still possible if the highest priority task which is configured to use the OS resource has time slicing configured. In that case, a task on the same priority level will be able to take the same OS resource. That may happen by a call of GetResource() or, in case of an internal resource, simply by scheduling to that task. As a consequence, a resource may be taken more than once (no possible implementation of a critical section anymore) or GetResource() may return with an error message (dependent on the configuration).

Caution

OS resources may only be used to implement critical sections for round robin tasks if the highest priority task, configured to access the resource, has higher priority than the round robin tasks.

3.19 Interrupt Only Use Case

3.19.1 Description

MICROSAR Classic OS offers a usage scenario, where the OS is only responsible for interrupt and exception handling. The only OS objects which shall be configured are category 1 interrupts.

The OS prepares the core and the interrupt controller to handle the configured category 1 interrupts. Furthermore it implements a mechanism to handle unconfigured interrupts and exceptions.

3.19.2 Activation

The Configuration attribute “OsUseCase” has to be set to INTERRUPT_ONLY.

3.19.3 Usage

To use the “Interrupt Only” use case, the OS service Os_InitInterruptOnly() needs to be called. This service configures the interrupt controller and the according AUTOSAR core to be able to handle the configured category 1 interrupts. For details regarding category 1 interrupts please also refer to chapter 2.4.5.

The following sequence should be implemented:

1. Call Os_InitInterruptOnly() (within the function,the basic interrupt controller settings are initialized, e.g. priorities of interrupt sources).
2. Enable the Interrupt sources of category 1 ISRs by directly manipulating the control registers in the interrupt controller.
3. Enable interrupt handling by calling Os_EnableInterruptsPreStart()

In case an unconfigured interrupt or exception occurs, an endless loop is entered. To execute user code before entering the endless loop, the user of MICROSAR Classic OS can configure the PanicHook.

Caution
The MICROSAR Classic OS does not allow OS API usage within the “Interrupt Only” use case.

The only exception is the interrupt handling API:

> DisableAllInterrupts()

> EnableAllInterrupts()

> Os_EnableInterruptsPreStart()

If any other OS API is called, MICROSAR Classic OS is not able to detect this and the called API may result in an unpredictable system behavior.

3.20 OS_APPMODE_ANY

3.20.1 Description

MICROSAR Classic OS offers a preconfigured application mode (AppMode) with the name "OS_APPMODE_ANY". This AppMode can be used to configure that a startup object (Task, Alarm, ScheduleTable) is automatically started in any AppMode. This way the object will be started automatically no matter which AppMode is passed to StartOS().

3.20.2 Usage

Reference the AppMode "OS_APPMODE_ANY" in the autostart configuration of all the objects that should be automatically started in every AppMode:

- Use the reference "OsAlarmAppModeRef" for an Alarm
- Use the reference "OsScheduleTableAppModeRef" for a ScheduleTable
- Use the reference "OsTaskAppModeRef" for a Task

Caution
The AppMode "OS_APPMODE_ANY" shall only be referenced. It shall never be passed to StartOS(), as this will cause a Kernel Panic.

3.21 Counter Algorithm

3.21.1 Description

MICROSAR Classic OS offers two algorithms for working off the alarms and schedule tables of a counter. An appropriate choice of the algorithm improves performance (run-time).

These algorithms are available:

- HEAP: A sorted data structure. Recommended if on average only a small percentage (at most approx. 10%) of the active alarms and schedule tables are expiring during a counter cycle (Timer-ISR).
- LIST: An unsorted data structure. Recommended if on average at least approx. 10% of the active alarms and schedule tables are expiring during a counter cycle. The higher the percentage, the better the performance of LIST compared to HEAP.

Example: If one or more of ten active alarms (at least 10%) expire every counter cycle, LIST is better than HEAP.

3.21.2 Activation

Within OS configuration the counter attribute "OsCounterAlgorithm" can either be set to HEAP or to LIST.

LIST is not available for a High Resolution Timer.

3.22 Asynchronous operation of API functions to activate tasks and set events

The AUTOSAR standard [1] does not describe any asynchronous execution of API functions. MICROSAR Classic OS provides the possibility to use all non-getter API functions asynchronously, as described in chapter 3.9. Chapter 3.22 describes the API functions ActivateTaskAsyn() and SetEventAsyn() as described in the later AUTOSAR OS standard [12] in order to perform task activation and event setting asynchronously. Asynchronous operation could increase the performance by reduction of synchronization waiting times.

3.22.1 Description

MICROSAR Classic OS supports asynchronous activation of tasks and event setting. Asynchronous means that the core that has executed the call (the caller core) will not be blocked waiting for the called core until it returns with status reporting.

- Available in systems that support OS applications.
- Intended to be used for cross-core task activation or event setting.
- Not intended to be used for core local task activation or event setting
- Errors are not returned to the caller but reported via error hooks.

3.22.2 Activation

The API service functions themselves do not need any activation.

3.22.3 Usage

Activating a task or setting an event for a task on a foreign core when the caller does not intend to wait for the result. The functions do not return an error code; they only call ErrorHook() in cases of errors.

Note

- If an error occurs during the task activation or event setting on the destination core, the ErrorHook will be called delayed. The call will be performed on the caller core at the next cross core API call from the same caller core to the same destination core.
- At the time of the ErrorHook call, the caller may no longer be available (OS-Application may already be terminated, caller core may be shutting down). In this case, the ErrorHook call is dropped and no reporting is done.

3.23 Configuration of Task and ISR Periods

The OS supports the configuration containers OsTaskPeriod and OsIsrPeriod as described in the later AUTOSAR OS standard [12].

4 Integration

4.1 Safety Manual

The MICROSAR SafetyManual [10] should be evaluated at an early stage. It is provided by Vector in case of ASIL deliveries.

It describes restrictions regarding the MICROSAR Classic OS configuration for safe systems, which have to be considered during the development process.

4.2 Compiler Optimization Assumptions

MICROSAR Classic OS makes the following assumptions for compiler optimization:

- Inlining of functions is active
- Not used functions are removed
- If statements with a constant condition (due to configuration) are optimized

4.2.1 Compile Time

To shorten the compile time of the OS the following measures can be taken within the OS configuration:

Systems without active memory protection (SC1/SC2)	Set “OsGenerateMemMap” to “EMPTY”
Systems with memory protection (SC3/SC4)	Set “OsGenerateMemMap” to “COMPLETE” and "OsGenerateMemMapForThreads” to “FALSE”

4.3 Hardware Software Interfaces (HSI)

The Hardware Software Interface describes all hardware registers which are used by the OS. Such registers must not be altered by user software.

Expert Knowledge

User software is allowed to access timer hardware registers in case no OS counter has been configured to use the corresponding timer. To verify that this case applies, check that the */MICROSAR/Os/OsCounter/OsDriver/OsDriverHardwareTimerChannelRef* of no OS counter references the desired timer hardware.

Included within the HSI is the context of the OS. The context is the sum of all registers which are preserved upon a task switch and ISR execution.

The detailed description of the HSI can be found in [9].

4.4 Memory Mapping Concept

MICROSAR Classic OS uses the AUTOSAR MemMap mechanism to locate its own variables but also application variables.

Note

To use the OS memory mapping concept within the AUTOSAR MemMap mechanism the generated OS file “Os_MemMap.h” has to be included into “MemMap.h”.

It should be included after the inclusion of the MemMap headers of all other basic software components.

4.4.1 Provided MemMap Section Specifiers

MICROSAR Classic OS uses and specifies section specifiers as described in the AUTOSAR specification of memory mapping. All section specifiers have one of the following forms:

OS_START_SEC_<SectionType>[_<InitPolicy>][_<Alignment>]

OS_STOP_SEC_<SectionType>[_<InitPolicy>][_<Alignment>]

Note

Due to clarity and understanding this chapter does only refer to section specifiers that shall be handled by the application.

The OS internally used section specifiers are not listed here.

SectionType	InitPolicy	Alignment
<Callout>_CODE	-	-
NONAUTOSAR_CORE<Core Id>_CONST	-	UNSPECIFIED
NONAUTOSAR_CORE<Core Id>_VAR	NOINIT	UNSPECIFIED
<ApplicationName>_VAR	- NOINIT ZERO_INIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
<ApplicationName>_VAR_FAST	- NOINIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
<ApplicationName>_VAR_NOCACHE	- NOINIT	BOOLEAN 8BIT

	ZERO_INIT	16BIT 32BIT UNSPECIFIED
<ApplicationName>_CONST	-	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
<ApplicationName>CONST_FAST	-	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED

SectionType	InitPolicy	Alignment
<Task/IsrName>_VAR	- NOINIT ZERO_INIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
<Task/IsrName>_VAR_FAST	- NOINIT ZERO_INIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
<Task/IsrName>_CONST	-	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
<Task/IsrName>_CONST_FAST	-	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED

SectionType	InitPolicy	Alignment
GLOBALSHARED_VAR	- NOINIT ZERO_INIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED

GLOBALSHARED_VAR_FAST	- NOINIT ZERO_INIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
GLOBALSHARED_VAR_NOCACHE	- NOINIT ZERO_INIT	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
GLOBALSHARED_CONST	-	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
GLOBALSHARED_CONST_FAST	-	BOOLEAN 8BIT 16BIT 32BIT UNSPECIFIED
APPSHARED_0X<application bitmask>_VAR_NOCACHE	NOINIT	UNSPECIFIED
CORESHARED_0X<core bitmask>_VAR_NOCACHE	NOINIT	UNSPECIFIED

Table 4-1 Provided MemMap Section Specifiers

Note

The < application bitmask >: Is a bitmask that specifies all OS applications which are sharing the section.

Note

The < core bitmask >: Is a bitmask that specifies all cores which are sharing the section.

4.4.1.1 Usage of MemMap Macros

Example

```
#define OS_START_SEC_MyAppl_VAR_FAST_NOINIT_UNSPECIFIED
#include "MemMap.h"
uint16 MyApplicationVariable;
#define OS_STOP_SEC_MyAppl_VAR_FAST_NOINIT_UNSPECIFIED
#include "MemMap.h"
```

This code snippet puts the user variable into an OS application section.

4.4.1.2 Resulting sections

The usage of the above described macros will result in the following memory sections:

SectionType	Content / Description
OS_CODE	> OS Code
OS_INTVEC_CODE	> Interrupt vector table in case the system needs one generic vector table for all cores
OS_INTVEC_CORE<Core Id>_CODE	> Interrupt vector table of one specific core
OS_EXCVEC_CORE<Core Id>_CODE	> Exception vector table of one core

Table 4-2 MemMap Code Sections Descriptions

Caution

The user must ensure, that the whole MICROSAR Classic OS code is linked between the labels generated into the array OsCfg_OsCode_Section<x>. This array may be found in Os_Error_Lcfg.c.

The resulting sections for callouts are generated in dependency of the configuration attribute "/MICROSAR/Os/OsOS/OsGenerateMemMap".

OsGenerateMemMap	Section	Content
USERCODE_AND_STACKS_GROUPED_PER_CORE	OS_USER_CORE<Core Id>_CODE	> Code of all Tasks, ISRs and all other user callouts which are mapped on one core.
COMPLETE	OS_<Callout>_CODE	> Code of one Task or one ISR or one OS Hook or other callouts.

Table 4-3 MemMap Callout Code Sections Descriptions

Note
The MPU may be set up to grant execution from the whole address space.

SectionType	Content / Description
OS_CONST	> OS constant data
OS_CONST_FAST	> OS constant data for fast memory
OS_INTVEC_CONST	> Interrupt vector table in case the system needs one generic vector table for all cores
OS_CORE<Core Id>_CONST	> OS constant data related to one specific core
OS_CORE<Core Id>_CONST_FAST	> OS constant data related to one specific for fast memory
OS_INTVEC_CORE<Core Id>_CONST	> Interrupt vector table of one specific core
OS_EXCVEC_CORE<Core Id>_CONST	> Exception vector table of one core
OS_NONAUTOSAR_CORE<Core Id>_CONST	> OS constant data of a non-AUTOSAR core
OS_NONAUTOSAR_CORE<Core Id>_CONST_FAST	> OS constant data of a non-AUTOSAR core with shord addressing
OS_GLOBALSHARED_CONST	> Constants which shall be shared among core boundaries
OS_GLOBALSHARED_CONST_FAST	> Constants which shall be shared among core boundaries and which use short addressing accesses (e.g. by base address pointer)
OS_<Task/IsrName>_CONST	> Thread specific constants
OS_<Task/IsrName>_CONST_FAST	> Thread specific constants which use short addressing accesses (e.g. by base address pointer)
OS_<ApplicationName>_CONST	> Application specific constants
OS_<ApplicationName>_CONST_FAST	> Application specific constants which use short addressing accesses (e.g. by base address pointer)

Table 4-4 MemMap Const Sections Descriptions

Note
The MPU may be set up to grant read access to const sections from all runtime contexts (trusted and non-trusted)

Section	Content
OS_VAR_NOCACHE	OS global variables. All cores may have to access these variables.
OS_VAR_NOCACHE_NOINIT	
OS_VAR_FAST_NOCACHE	
OS_VAR_FAST_NOCACHE_NOINIT	
OS_CORE<Core Id>_VAR	OS core local variables. These variables are never accessed from foreign cores.
OS_CORE<Core Id>_VAR_FAST	
OS_CORE<Core Id>_VAR_NOINIT	
OS_CORE<Core Id>_VAR_FAST_NOINIT	
OS_CORE<Core Id>_VAR_NOCACHE	
OS_CORE<Core Id>_VAR_FAST_NOCACHE	
OS_CORE<Core Id>_VAR_NOCACHE_NOINIT	
OS_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT	
OS_PUBLIC_CORE<Core Id>_VAR_NOINIT	OS core local variables. These variables may also be accessed from foreign cores
OS_PUBLIC_CORE<Core Id>_VAR_FAST_NOINIT	
OS_APPSHARED_0X<application bitmask>_VAR_NOCACHE_NOINIT	OS optimized spinlock variables. Only OS applications specified by <application bitmask> have access to them.
OS_CORESHARED_0X<core bitmask>_VAR_NOCACHE_NOINIT	OS Standard/Optimized spinlock variables. IOC data structures. All cores wich are specified by <core bitmask> have access to them.
OS_NONAUTOSAR_CORE<Core Id>_VAR	User core local variables of non-AUTOSAR cores. Access to these from foreign cores may be allowed.
OS_NONAUTOSAR_CORE<Core Id>_VAR_FAST	
OS_NONAUTOSAR_CORE<Core Id>_VAR_NOINIT	
OS_NONAUTOSAR_CORE<Core Id>_VAR_FAST_NOINIT	

Section	Content
OS_GLOBALSHARED_VAR	User global shared variables. All cores have access to them.
OS_GLOBALSHARED_VAR_FAST	
OS_GLOBALSHARED_VAR_NOINIT	
OS_GLOBALSHARED_VAR_FAST_NOINIT	
OS_GLOBALSHARED_VAR_ZERO_INIT	
OS_GLOBALSHARED_VAR_NOCACHE	
OS_GLOBALSHARED_VAR_FAST_NOCACHE	
OS_GLOBALSHARED_VAR_NOCACHE_NOINIT	
OS_GLOBALSHARED_VAR_FAST_NOCACHE_NOINIT	
OS_GLOBALSHARED_VAR_NOCACHE_ZERO_INIT	
OS_<ApplicationName>_VAR	User application private variables. Only application members and other trusted software may have access to them.
OS_<ApplicationName>_VAR_FAST	
OS_<ApplicationName>_VAR_NOINIT	
OS_<ApplicationName>_VAR_FAST_NOINIT	
OS_<ApplicationName>_VAR_FAST_ZERO_INIT	
OS_<ApplicationName>_VAR_NOCACHE	
OS_<ApplicationName>_VAR_FAST_NOCACHE	
OS_<ApplicationName>_VAR_NOCACHE_NOINIT	
OS_<ApplicationName>_VAR_FAST_NOCACHE_NOINIT	
OS_<ApplicationName>_VAR_NOCACHE_ZERO_INIT	

Section	Content
OS_<Task/IsrName>_VAR	User thread private variables. Only the owning thread and other trusted software may have access to them
OS_<Task/IsrName>_VAR_FAST	
OS_<Task/IsrName>_VAR_NOINIT	
OS_<Task/IsrName>_VAR_FAST_NOINIT	
OS_<Task/IsrName>_VAR_ZERO_INIT	
OS_BARRIER_CORE<Core Id>_VAR_NOCACHE_NOINIT	OS synchronization barriers. Only the OS must have access to them. They will be accessed from all cores
OS_BARRIER_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT	
OS_CORESTATUS_CORE<Core Id>_VAR_ NOCACHE_NOINIT	Startup state of each physical core. Only the OS must have access to them. They will be written by the master core and the owning core itself, and read from all cores.
OS_CORESTATUS_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT	

Table 4-5 MemMap Variable Sections Descriptions

The resulting sections for stacks are generated in dependency of the configuration attribute "/MICROSAR/Os/OsOS/OsGenerateMemMap".

OsGenerateMemMap	Section	Content
USERCODE_AND_STACKS_GROUPED_PER_CORE	OS_STACK_CORE<Core Id>_VAR_NOINIT	Contains all stacks of one core. Only the current running software has access to the stack. Software which runs on a foreign core must not have access to it.
COMPLETE	OS_STACK_<StackName>_VAR_NOINIT	Contains one OS stack. Only the current running software has access to the stack. Software which runs on a foreign core must not have access to it.

Table 4-6 MemMap Variable Stack Sections Descriptions

Notes
Sections which contain the keyword "FAST" are intended to be linked into fast RAM.

Sections which contain the keyword "NOCACHE" must never be linked into cacheable memory.

Sections which contain the keyword "NOINIT" contain non-initialized variables.

Sections which contain the keyword "ZERO_INIT" contain zero initialized variables.

4.4.1.3 Access Rights to Variable Sections

The table shows the recommended access rights to the sections.

<table>
<tr><th>Section</th><th>Local core trusted</th><th>Local core non trusted</th><th>Foreign core trusted</th><th>Foreign core non trusted</th></tr>
<tr><td>OS_VAR_NOCACHE</td><td rowspan="4">RW</td><td rowspan="4">RO</td><td rowspan="4">RW</td><td rowspan="4">RO</td></tr>
<tr><td>OS_VAR_NOCACHE_NOINIT</td></tr>
<tr><td>OS_VAR_FAST_NOCACHE</td></tr>
<tr><td>OS_VAR_FAST_NOCACHE_NOINIT</td></tr>
<tr><td>OS_CORE<Core Id>_VAR</td><td rowspan="8">RW</td><td rowspan="8">RO</td><td rowspan="8">RO</td><td rowspan="8">RO</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_FAST</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_NOINIT</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_FAST_NOINIT</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_NOCACHE</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_FAST_NOCACHE</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_NOCACHE_NOINIT</td></tr>
<tr><td>OS_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT</td></tr>
<tr><td>OS_PUBLIC_CORE<Core Id>_VAR_NOINIT</td><td rowspan="2">RW</td><td rowspan="2">RO</td><td rowspan="2">RW</td><td rowspan="2">RO</td></tr>
<tr><td>OS_PUBLIC_CORE<Core Id>_VAR_FAST_NOINIT</td></tr>
<tr><td>OS_NONAUTOSAR_CORE<Core Id>_VAR</td><td rowspan="4">RW</td><td rowspan="4">RO</td><td rowspan="4">RW</td><td rowspan="4">RO</td></tr>
<tr><td>OS_NONAUTOSAR_CORE<Core Id>_VAR_FAST</td></tr>
<tr><td>OS_NONAUTOSAR_CORE<Core Id>_VAR_NOINIT</td></tr>
<tr><td>OS_NONAUTOSAR_CORE<Core Id>_VAR_FAST_NOINIT</td></tr>
<tr><td>OS_GLOBALSHARED_VAR</td><td rowspan="10">RW</td><td rowspan="10">RW</td><td rowspan="10">RW</td><td rowspan="10">RW</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_FAST</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_NOINIT</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_FAST_NOINIT</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_ZERO_INIT</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_NOCACHE</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_FAST_NOCACHE</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_NOCACHE_NOINIT</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_FAST_NOCACHE_NOINIT</td></tr>
<tr><td>OS_GLOBALSHARED_VAR_NOCACHE_ZERO_INIT</td></tr>
</table>

<table>
<tr><th>Section</th><th>Local core trusted</th><th>Local core non trusted</th><th>Foreign core trusted</th><th>Foreign core non trusted</th></tr>
<tr><td>OS_<ApplicationName>_VAR</td><td rowspan="10">RW</td><td rowspan="10">RW</td><td rowspan="10">RW</td><td rowspan="10">RO</td></tr>
<tr><td>OS_<ApplicationName>_VAR_FAST</td></tr>
<tr><td>OS_<ApplicationName>_VAR_NOINIT</td></tr>
<tr><td>OS_<ApplicationName>_VAR_FAST_NOINIT</td></tr>
<tr><td>OS_<ApplicationName>_VAR_FAST_ZERO_INIT</td></tr>
<tr><td>OS_<ApplicationName>_VAR_NOCACHE</td></tr>
<tr><td>OS_<ApplicationName>_VAR_FAST_NOCACHE</td></tr>
<tr><td>OS_<ApplicationName>_VAR_NOCACHE_NOINIT</td></tr>
<tr><td>OS_<ApplicationName>_VAR_FAST_NOCACHE_NOINIT</td></tr>
<tr><td>OS_<ApplicationName>_VAR_NOCACHE_ZERO_INIT</td></tr>
<tr><td>OS_<Task/IsrName>_VAR</td><td rowspan="5">RW</td><td rowspan="5">RW</td><td rowspan="5">RW</td><td rowspan="5">RO</td></tr>
<tr><td>OS_<Task/IsrName>_VAR_FAST</td></tr>
<tr><td>OS_<Task/IsrName>_VAR_NOINIT</td></tr>
<tr><td>OS_<Task/IsrName>_VAR_FAST_NOINIT</td></tr>
<tr><td>OS_<Task/IsrName>_VAR_ZERO_INIT</td></tr>
<tr><td>OS_BARRIER_CORE<Core Id>_VAR_NOCACHE_NOINIT</td><td rowspan="2">RW</td><td rowspan="2">RO</td><td rowspan="2">RW</td><td rowspan="2">RO</td></tr>
<tr><td>OS_BARRIER_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT</td></tr>
<tr><td>OS_CORESTATUS_CORE<Core Id>_VAR_ NOCACHE_NOINIT</td><td rowspan="2">RW</td><td rowspan="2">RO</td><td rowspan="2">RW</td><td rowspan="2">RO</td></tr>
<tr><td>OS_CORESTATUS_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT</td></tr>
</table>

Table 4-7 Recommended Section Access Rights

Note
The access to the stack section is handled completely by MICROSAR Classic OS

Note
The table is only valid for cores which have the same diagnostic coverage capabilities. Cores with a lower diagnostic coverage level must never interact with data from a core with a higher diagnostic coverage level.

4.4.1.4 Access Rights to Shared Data Sections

Section	Access Rights
OS_APPSHARED_0X<application bitmask>_VAR_NOCACHE_NOINIT	Only applications which are specified by the <application bitmask> shall have read / write access. The bitmasks of applications may be looked up in “Os_Types_Lcfg.h” > "ApplicationType".
OS_CORESHARED_0X<core bitmask>_VAR_NOCACHE_NOINIT	Only cores which are specified by the <core bitmask> shall have read / write access. The bitmasks of cores may be looked up in “Os_Hal_Lcfg.h” > "CoreIdType".

Table 4-8 Recommended Shared Data Section Access Rights

The shared data sections are used to achieve freedom from interference between cores with different diagnostic coverage capability.

The table below shows the recommended access rights to the sections.

Section	Local core trusted	Local core non trusted	Foreign core trusted	Foreign core non trusted
OS_APPSHARED_0X<application bitmask>_VAR_NOCACHE_NOINIT OS_CORESHARED_0X<core bitmask>_VAR_NOCACHE_NOINIT	RW	RO	RW	RO

Table 4-9 Recommended Shared Data Section Core Access Rights

Note
As shared data section names contain a bitmask, they can be long. Renaming of applications can shorten APPSHARED section names as applications are sorted alphabetically in ApplicationType. Minimum length of a bitmask is 8 characters.

4.4.2 Link Sections

Once variables have been put into OS sections (by usage of the section specifiers described in 4.4.1.1) the sections would have to be linked.

Therefore MICROSAR Classic OS generates linker command files which utilize the linkage of those sections.

Linker Command Filename	Content
Os_Link_<Core>.<FileSuffix>	All data and code sections which are bound to a core
Os_Link.<FileSuffix>	All data and code sections which are global
Os_Link_<Core>_Stacks.<FileSuffix>	all stacks of a core

Table 4-10 List of Generated Linker Command Files

Note
<Core> is the logical core ID

<FileSuffix> is the suffix for linker command files. It depends on the used compiler.

4.4.2.1 Pre-Process Linker Command Files

The generated linker command files uses C pre-processor statements. Some Linkers don't understand pre-processor statements. These Linkers require a pre-processing step on the linker command files.

Windriver DiabData	The pre-processor should be used on command line to pre-process the linker command files e.g.: `dcc.exe -P Os_Link.dld -o Os_Link_new.dld`

4.4.2.2 Simple Linker Defines

The following defines are used to select groups of OS sections from the linker command files.

Select OS code	OS_LINK_CODE
Select an interrupt vector table	OS_LINK_INTVEC_CODE
Select an exception vector table	OS_LINK_EXCVEC_CODE
Select user callouts (Tasks, ISRs, Hooks)	OS_LINK_CALLOUT_CODE
Select constants related to an interrupt vector table	OS_LINK_INTVEC_CONST
Select constants related to an exception vector table	OS_LINK_EXCVEC_CONST
Select OS stacks	OS_LINK_KERNEL_STACKS

Example

```
#define OS_LINK_INTVEC_CODE
#include Os_Link_Core0.lsl
```

Selects the interrupt vector table from the included linker command file for linking.

4.4.2.3 Hierachical Linker Defines

The linker command files are intended to be included into a main linker command file.
Single sections or group of sections can be selected for linkage by usage of C-like defines.
This mechanism is similar to the MemMap mechanism of AUTOSAR.
The linker defines of MICROSAR Classic OS uses a hierarchical syntax.
The more one walks down in the hierarchy the less sections are selected.

Note
Once one have made the decision for a specific hierarchical level one will have to stick to this level throughout the linker defines group. Otherwise there may be multiple section definitions.

4.4.2.4 Selecting OS constants

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1
OS_LINK_CONST_KERNEL	_NEAR _FAR

Table 4-11 OS constants linker define group

Example

```
#define OS_LINK_CONST_KERNEL
#include Os_Link_Core0.lsl
```

Selects all OS constants.

Example

```
#define OS_LINK_CONST_KERNEL_NEAR
#include Os_Link_Core0.lsl
```

Selects all near addressable OS constants only.

4.4.2.5 Selecting OS variables

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1	Optional Hierarchy level 2	Optional Hierarchy level 3
OS_LINK_VAR_KERNEL	_NEAR _FAR	_CACHE _NOCACHE	_INIT _NOINIT

Table 4-12 OS variables linker define group

Example

```
#define OS_LINK_VAR_KERNEL
#include Os_Link_Core0.lsl
```

Selects all OS variables.

Example

```
#define OS_LINK_VAR_KERNEL_NEAR_CACHE
#include Os_Link_Core0.lsl
```

Selects all OS variables which are near addressable and cacheable.

4.4.2.6 Selecting special OS Variables

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1
OS_LINK_KERNEL_BARRIERS	_NEAR _FAR
OS_LINK_KERNEL_CORESTATUS	
OS_LINK_KERNEL_TRACE	

Table 4-13 OS Barriers and Core status linker define group

Example

```
#define OS_LINK_KERNEL_BARRIERS
#include Os_Link_Core0.lsl
```

Selects all OS Barriers.

Example

```
#define OS_LINK_KERNEL_CORESTATUS
#include Os_Link_Core0.lsl
```

Selects all OS core state variables.

Prefix	Optional Hierarchy level 1	Owner Bitmask	Optional Hierarchy level 2
OS_LINK_VAR	_APPSHARED	_0X<application bitmask>	_NEAR _FAR
	_CORESHARED	_0X<core bitmask>	

Example

```
#define OS_LINK_VAR_APPSHARED
#include Os_Link.lsl
```

Selects all OS application shared variables

4.4.2.7 Selecting User Constant Sections

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1	Owner Name	Optional Hierarchy level 2
OS_LINK_CONST	_APP	<Owner Name>	_NEAR _FAR
	_TASK		
	_ISR		
	_GLOBALSHARED	---	

Table 4-14 User constants linker define group

Example

```
#define OS_LINK_CONST_APP_<ApplicationName>
#include Os_Link_Core0.lsl
```

Selects all constants which belong to the OS application <ApplicationName>

Example

```
#define OS_LINK_CONST_ISR_<ISRName>_FAR
#include Os_Link_Core0.lsl
```

Selects all constants which belong to the ISR <ISRName> which have far addressing

4.4.2.8 Selecting User Variable Sections

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1	Owner Name	Optional Hierarchy level 2	Optional Hierarchy level 3	Optional Hierarchy level 4
OS_LINK_VAR	_APP	<Owner Name>	_NEAR _FAR	_CACHE _NOCACHE	_INIT _NOINIT _ZEROINIT
	_TASK				
	_ISR				
	_GLOBALSHARED	---			

Table 4-15 User variables linker define group

Example

```
#define OS_LINK_VAR_APP_<ApplicationName>
#include Os_Link_Core0.lsl
```

Selects all variables which belong to the OS application <ApplicationName>

Example

```
#define OS_LINK_VAR_APP_<ApplicationName>_FAR_CACHE_INIT
#include Os_Link_Core0.lsl
```

Selects all variables which belong to the OS application <ApplicationName> which have far addressing, are cacheable and are initialized

4.4.3 Section Symbols

The linker command files described in 0 also generate section start and stop symbols which can be used to configure start and end addresses of MPU regions, peripheral regions or access check region objects.

The generated linker section start and stop symbols have the following syntax:

_OS_<SectionType>_START

_OS_<SectionType>_END

_OS_<SectionType>_LIMIT

The symbol _OS_<SectionType>_END points to the last accessible address of a region.

The symbol _OS_<SectionType>_LIMIT points to the first address after the region.

Example

Const data which belongs to section OS_MyAppl_CONST is included within the symbols

_OS_MyAppl_CONST_START

_OS_MyAppl_CONST_END

_OS_MyAppl_CONST_LIMIT

Data which belongs to section OS_MyAppl_VAR_FAST is included within the symbols

_OS_MyAppl_VAR_FAST_START

_OS_MyAppl_VAR_FAST_END

_OS_MyAppl_VAR_FAST_LIMIT

Data which belongs to section OS_MyTask_VAR_FAST is included within the symbols

_OS_MyTask_VAR_FAST_START

_OS_MyTask_VAR_FAST_END

_OS_MyTask_VAR_FAST_LIMIT

4.4.3.1 Aggregation of Data Sections

Additional start and stop linker symbols are generated which contain all data sections of applications, tasks and ISRs. These symbols can be used to configure start and end addresses of MPU regions, peripheral regions or access check region objects.

These start and stop linker symbols have the syntax:

_OS_<SectionOwner>_VAR_ALL_START

_OS_<SectionOwner>_VAR_ALL_END

_OS_<SectionOwner>_VAR_ALL_LIMIT

<SectionOwner> is name of applications, tasks and CAT2 ISRs used in configurator.

The symbol _OS_<SectionOwner>_VAR_ALL_END points to the last accessible address of a region.

The symbol _OS_<SectionOwner>_VAR_ALL__LIMIT points to the first address after the region.

Example

All data sections which belong to application "MyAppl" are included within the symbols

_OS_MyAppl_VAR_ALL_START

_OS_MyAppl_VAR_ALL_END

_OS_MyAppl_VAR_ALL_LIMIT

All data sections which belong to task "MyTask" are included within the symbols

_OS_MyTask_VAR_ALL_START

_OS_MyTask_VAR_ALL_END

_OS_MyTask_VAR_ALL_LIMIT

All data sections which belong to CAT2 ISR "MyISR" are included within the symbols

_OS_MyISR_VAR_ALL_START

_OS_MyISR_VAR_ALL_END

_OS_MyISR_VAR_ALL_LIMIT

4.5 Static Code Analysis

Note

When running tools for static code analysis (e.g. MISRA, MSSV), the pre-processor definition OS_STATIC_CODE_ANALYSIS has to be set during analysis. It switches off compiler specific keywords and inline assembler parts. Typically code analysis tools cannot deal with such code parts.

4.6 Configuration of X-Signals

This chapter describes how X-Signals are configured for cross core API calls. Note that X-Signals are used only to provide the Cross Core API functions as described in this document and in the Autosar OS Standard. They do not provide functionality to the application directly. See chapter 3.9 for a better understanding of X-Signals.

1. Add an “OsCoreXSignalChannel” to an “OsCore” object. This core will be the sender of the X-Signal.
2. Specify the queue size of the channel with the “OsCoreXSignalChannelSize” attribute.
3. Add an X-Signal receiver ISR. It must be of category 2.
4. Assign this ISR to be the X-Signal receiver “OsCore/OsCoreXSignalChannelReceiverIsr”.
5. Configure an appropriate interrupt priority for the receiver ISR (see the following chapter for VTT details and [9] for platform specific details). The configured priority must follow the rules listed in Table 3-3.
6. Choose an appropriate interrupt source for the receiver ISR (see the following chapter for VTT details and [9] for platform specific details).
7. Add the "OsIsrXSignalReceiver" to the receiver ISR and select the provided APIs (callable from the sender core) with the "OsIsrXSignalReceiverProvidedApis" attribute.

Expert Knowledge
MICROSAR Classic OS offers the possibility to configure different Receiver-ISRs for each Sender. This may be helpful for systems that provide cores with different diagnostical coverage levels (e.g. Lockstep- and Non-Lockstep-Cores).

Note
The DaVinci Configurator provides solving actions which support the correct configuration of X-Signals.

4.6.1 VTT OS

Logical Priority	A low number for OsIsrInterruptPriority attribute means a low logical priority
X-Signal ISR Interrupt Priority	Beside the rules listed in Table 3-3 the OsIsrInterruptPriority can be chosen freely.
X-Signal ISR Interrupt Source	Any interrupt source, which is not used by other modules, may be used for the X-Signal ISR.

4.7 OS generated objects

In dependency of its configuration MICROSAR Classic OS may add other OS configuration objects to it.

4.7.1 System Application

Type	OsApplication
Name	SystemApplication_<CoreName>
Condition	Is added when the OsCore <CoreName> is configured to be an AUTOSAR core.

Features	> A system application contains the OS objects
	  > IdleTask_<CoreName>
	  > TpCounter_<CoreName>
	  > XSignalIsr_<CoreName>
	  > CounterIsr_TpCounter_<CoreName>

User configured OS objects
The System Application shall not contain any user configured OS objects in case of SC2, SC3 and SC4 systems. As defined by AUTOSAR the user shall configure additional Applications for user defined OS objects.

4.7.2 Idle Task

Type	OsTask
Name	IdleTask_<CoreName>
Condition	Is added when the OsCore <CoreName> is configured to be an AUTOSAR core.
Features	> Has the lowest priority of all tasks assigned to the same core. > Is fully preemptive. > Is implemented by the OS

Idle Task Priority
The generator has a special treatment for the idle task. The idle task has the virtual priority 0xFFFFFFFF to differentiate it from regular tasks. It will be generated to have the lowest priority, even if there are tasks configured with priority 0.

User Code Execution
The idle task is implemented by the OS to simplify scheduling and idle treatment. The OS does not rely on execution of the idle task. Implement an additional task with priority 0, if user code execution during idle time is needed.

4.7.3 Timer ISR

Type	OsIsr
Name	CounterIsr_<CoreName>
Condition	Is added if a hardware OsCounter is configured to have a driver (attribute “OsCounterDriver”).
Features	> Is Implemented by the OS. > Handles the system timer counter, alarms and scheduletables which are assigned to the core.

4.7.4 System Timer Counter

Type	OsCounter
Name	SystemTimer
Condition	Is added optionally within the recommended configuration.
Features	> Is used for OSEK backward compatibility

4.7.5 Timing Protection Counter

Type	OsCounter
Name	TpCounter_<CoreName>
Condition	Is added when OsTask/IsrTimingProtection parameters are configured on the core.
Features	> Handles all times related to timing protection

4.7.6 Timing protection ISR

Type	OsIsr
Name	CounterIsr_TpCounter_<CoreName>
Condition	Is added when OsTask/IsrTimingProtection parameters are configured on the core.
Features	> Interrupt service routine of the timing protection feature

4.7.7 Resource Scheduler

Type	OsResource
Name	RES_SCHEDULER_<CoreName>
Condition	For each core the resource scheduler is added when OsUseResScheduler is set to TRUE.
Features	> Is automatically assigned to all tasks of core <CoreName>

4.7.8 X-Signal ISR

Type	OsIsr
Name	XSignalIsr_<CoreName>
Condition	Is added when an X-Signal channel is configured on the core.
Features	> Handles cross core requests.

4.7.9 IOC Spinlocks

Type	OsSpinlock
Name	IocSpinlock_<IOC Name>
Condition	Is added when an IOC is configured which requires cross core communication.
Features	> Each IOC has its own spinlock to reduce core wait times

4.8 VTT OS Specifics

4.8.1 Configuration

As described in [6] all VTT configuration parameters are derived from the hardware target. The only exceptions are the ISR objects for the VTT OS.

- ➔ ISRs from other Vector MICROSAR BSW vVIRTUALtarget driver modules (e.g. VTTCan) are inserted automatically by the respective BSW module.
- ➔ ISRs from other modules and user ISRs have to be added separately.
- ➔ Interrupt levels for all ISRs have to be configured manually. VTT OS knows interrupt levels from 1 to 200 (where 1 is the lowest priority and 200 the highest).

4.8.2 CANoe Interface

A VTT OS is simulated within the CANoe simulation software. There are a set of API functions which are capable to communicate with CANoe (e.g. sending a message on the CAN bus).

These API functions are prefixed with “CANoeAPI_”.

The available set of API functions can be looked up in the delivered header “CANoeApi.h”.

4.8.2.1 Idle Task behavior with VTT OS

Any idle task which runs within the VTT OS must call the function “CANoeAPI_ConsumeTicks” (see description in CANoeApi.h).

> **Caution**
> If the call of “CANoeAPI_ConsumeTicks” is missing within the idle task, the CANoe windows application won’t respond any longer!

There are two possible solutions which solves this problem:

1. The OS generated idle task (see 4.7.2) calls this function by default. The application has to ensure that this idle task is entered cyclically.
2. It may be that the OS idle task is never executed, because there is a higher priority application idle task. This application idle task must implement a cyclic call of “CANoeAPI_ConsumeTicks” instead of the OS idle task.

4.8.3 Timing Adjustment

> **Expert Knowledge**
> Within waiting loops, which are used, for example, for synchronous X-Signal requests (see 3.9.1), the function "CANoeAPI_ConsumeTicks" is called. The number of ticks that should be consumed can be configured via the preprocessor definition OS_VTT_TICKS_PER_NOP_INSTRUCTION, which is set to 100 by default.
>
> Please be aware that redefining this macro can have unexpected effects on the timing behavior.

4.8.4 Compilation

> **Caution**
> If the Microsoft compiler is used, the following option must be specified:
>
> `/volatile:ms`

4.9 User include files

Within some features of MICROSAR Classic OS it may be necessary to provide foreign data types to the OS.

This can be done by referencing user headers within the OS configuration.

The features “IOC” and “trusted functions stub generation” are relying on such include mechanisms.

	Configuration	Content
IOC	IOC include files are configured with the IOC attribute "OsIocIncludeHeader“. A list of include files may be specified here.	The headers have to provide > Definitions of foreign OS data types which are used within IOC communication.
Trusted Functions	Include files which are needed for trusted function feature are configured within the application attribute “OsAppCalloutStubsIncludeHeader”. A list of include files may be specified here.	The headers have to provide > The definitions of foreign OS data types which are used as trusted functions parameters or return values.

> **Caution**
> All user include files need to implement a double inclusion preventer!

4.10 Preprocessing of assembler language files

Dependent on the hardware platform, MICROSAR Classic OS may use preprocessing of assembly language files. However, some of the supported compiler tool chains do not allow to preprocess assembly language files with the normal C preprocessor. Therefore, the compiler or the assembler may state some error messages.

In such a case, another preprocessor may be used.

Example

The following compiler tool suite does not support preprocessing of assembly language files: TI compiler (Texas Instruments).

The following tool of the GNU compiler collection has shown to work correctly on the files delivered with MICROSAR Classic OS:

cpp (tdm-1) 4.9.2

It should be used in the following way:

```
cpp.exe -P -DOS_CFG_DERIVATIVEGROUP_<YourDerivativeGroup>
        -DOS_CFG_COMPILER_TEXASINSTRUMENTS
        -I$(PATH_OS_IMPLEMENTATION) -I$(PATH_OS_GENDATA)
        <YourAssemblyFile>.asm -o $(PATH_OUTPUT)
```

4.11 Stack Summary

The DaVinci configurator provides an overview of all internal calculated stacks in a separated table in /MICROSAR/Os/OsOS/OsStackSummary.

For example, this overview table can be used to determine which task uses which stack and how the size is configured.

Note
This stack summary is automatically created and updated during configuration by the OS generator. Manual configuration of stacks in this summary is not supported.

The size must be configured at the stack size parameter of the container which is referenced as user (e.g. OsTaskStackSize).

Basic Knowledge
For shared stacks the biggest configured stack size of all users is used to set up the stack size in the summary.

5 API Description

This chapter lists all API service functions which are provided by MICROSAR Classic OS.

5.1 Specified OS services

The OS provides the following services which are specified within the AUTOSAR OS specification.

5.1.1 StartCore

Prototype	
`void StartCore (CoreIdType CoreID, StatusType *Status)`	
Parameter	
CoreID [in]	The core to start.
Status [out]	Status code.
Return code	
void	> E_OK No Error. > E_OS_ID (EXTENDED status) Either the Core ID is invalid or the Core is no AUTOSAR core. > E_OS_ACCESS (EXTENDED status) The function was called after starting the OS. > E_OS_STATE (EXTENDED status) The Core is already activated.
Functional Description	
OS service StartCore().	
Particularities and Limitations	
> Pre-Condition: Supervisor mode. Pre-Condition: Given object pointer(s) are valid. Starts the core given by CoreID that is controlled by the AUTOSAR OS. This API is allowed to be used from AUTOSAR and non-AUTOSAR cores.	
Call context	
> Startup Code before StartOS() > This function is Synchronous > This function is Non-Reentrant	

Table 5-1 StartCore

5.1.2 StartNonAutosarCore

Prototype	
`void StartNonAutosarCore (CoreIdType CoreID, StatusType *Status)`	
Parameter	
CoreID	The core to start.
Status [out]	Status code.
Return code	
void	> E_OK No Error. > E_OS_ID (EXTENDED status:) Core ID is invalid. > E_OS_STATE (EXTENDED status:) The Core is already activated.
Functional Description	
OS service StartNonAutosarCore().	
Particularities and Limitations	
Pre-Condition: Supervisor mode. Starts the core given by CoreID that is not controlled by the AUTOSAR OS.	
Call context	
> - > This function is Synchronous > This function is Non-Reentrant	

Table 5-2 StartNonAutosarCore

5.1.3 GetCoreID

Prototype	
`CoreIdType` **`GetCoreID`** `(void)`	
Parameter	
void	none
Return code	
CoreIdType	Unique ID of the calling core.
Functional Description	
OS service GetCoreID().	
Particularities and Limitations	
Pre-Condition: None Returns the unique logical core identifier of the core on which the function is called. The mapping of physical cores to logical CoreIDs is implementation specific. This API is allowed to be used from AUTOSAR cores only. If the API is required on a non-AUTOSAR core, it is possible to configure the core as an AUTOSAR core but start it as a non-AUTOSAR core using the StartNonAutosarCore() API.	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-3 GetCoreID

5.1.4 GetNumberOfActivatedCores

Prototype	
`uint32 GetNumberOfActivatedCores (void)`	
Parameter	
void	none
Return code	
uint32	Number of cores activated by the StartCore() function.
Functional Description	
OS service GetNumberOfActivatedCores().	
Particularities and Limitations	
Pre-Condition: None The function returns the number of cores activated by the StartCore() function. AUTOSAR specifies this function to be usable from task and ISR call level. But this function does not explicitly perform any call context checks. There is no need to, because it is a primitive getter function.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-4 GetNumberOfActivatedCores

5.1.5 GetActiveApplicationMode

Prototype	
`AppModeType` **`GetActiveApplicationMode`** `(void)`	
Parameter	
void	none
Return code	
AppModeType	Current Application Mode
Functional Description	
OS service GetActiveApplicationMode().	
Particularities and Limitations	
Pre-Condition: None This service returns the current application mode.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-5 GetActiveApplicationMode

5.1.6 StartOS

Prototype	
`void StartOS (AppModeType Mode)`	
Parameter	
Mode [in]	The application mode in which the OS shall start.
Return code	
void	none
Functional Description	
OS service StartOS().	
Particularities and Limitations	
> Pre-Condition: Supervisor mode. Pre-Condition: Os_Init() has been called before. Starts the operating system in a given application mode.	
Call context	
> - > This function is Synchronous > This function is Non-Reentrant	

Table 5-6 StartOS

5.1.7 ShutdownOS

Prototype	
`void ShutdownOS (StatusType Error)`	
Parameter	
Error	Error code which shall be passed to the ShutdownHook()
Return code	
void	none
Functional Description	
OS service ShutdownOS().	
Particularities and Limitations	
Pre-Condition: None This function shall shutdown the core on which it was called. Only allowed in trusted applications. In case that ShutdownOS() is called from an invalid context, OS_STATUS_CALLEVEL is reported via the ProtectionHook.	
Call context	
> TASK|ISR2|ERRHOOK|STARTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-7 ShutdownOS

5.1.8 ShutdownAllCores

Prototype	
`void ShutdownAllCores (StatusType Error)`	
Parameter	
Error [in]	This is the error code which shall be passed to the ShutdownHook().
Return code	
void	none
Functional Description	
OS service ShutdownAllCores().	
Particularities and Limitations	
Pre-Condition: None Propagates a shutdown request to all started AUTOSAR cores and performs a shutdown itself afterwards. Only allowed in trusted applications.	
Call context	
> TASK|ISR2|ERRHOOK|STARTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-8 ShutdownAllCores

5.1.9 ControlIdle

Prototype	
`StatusType ControlIdle (CoreIdType CoreID, IdleModeType IdleMode)`	
Parameter	
CoreID [in]	Selects the core which idle mode is set
IdleMode [in]	The mode which shall be performed during idle time
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status): Invalid core and/or invalid IdleMode. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service ControlIdle().	
Particularities and Limitations	
Pre-Condition: None This API allows the caller to select the idle mode action which is performed during idle time of the OS (e.g. if no Task/ISR is active). The real idle modes are hardware dependent and not standardized. The default idle mode on each core is IDLE_NO_HALT.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Non-Reentrant	

Table 5-9 ControlIdle

5.1.10 GetSpinlock

Prototype	
`StatusType **GetSpinlock** (SpinlockIdType SpinlockId)`	
Parameter	
SpinlockId [in]	The spinlock which shall be locked.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid SpinlockID. > E_OS_INTERFERENCE_DEADLOCK (EXTENDED status:) Spinlock already occupied by a task/ISR of the same core. > E_OS_NESTING_DEADLOCK (EXTENDED status:) Invalid Spinlock allocation order. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient.
Functional Description	
OS service GetSpinlock().	
Particularities and Limitations	
Pre-Condition: None Allocates the requested spinlock for the caller. If it is already locked, the function performs active around until the spinlock becomes available again.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-10 GetSpinlock

5.1.11 ReleaseSpinlock

Prototype	
`StatusType` **`ReleaseSpinlock`** `(SpinlockIdType SpinlockId)`	
Parameter	
SpinlockId [in]	The spinlock which shall be released.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid SpinlockID. > E_OS_STATE (EXTENDED status:) The caller is not the owner of the given spinlock. > E_OS_NOFUNC (EXTENDED status:) The caller tries to release a spinlock while another spinlock has to be released before. > E_OS_RESOURCE (EXTENDED status:) Spinlock and Resource API not used in LIFO order. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient. This error may occur in combination with trusted functions.
Functional Description	
OS service ReleaseSpinlock().	
Particularities and Limitations	
Pre-Condition: None ReleaseSpinlock releases a spinlock variable that was occupied before. Before terminating a task/ISR all spinlock variables that have been occupied with GetSpinlock() shall be released. The error E_OS_CALLEVEL is already checked by E_OS_STATE. See Os_SpinlockRelease() for details.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-11 ReleaseSpinlock

5.1.12 TryToGetSpinlock

Prototype	
`StatusType **TryToGetSpinlock** (SpinlockIdType SpinlockId, TryToGetSpinlockType *Success)`	
Parameter	
SpinlockId [in]	The spinlock which shall be locked.
Success [out]	The result of the allocation attempt.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid SpinlockID. > E_OS_INTERFERENCE_DEADLOCK (EXTENDED status:) Spinlock already occupied by a task/ISR of the same core. > E_OS_NESTING_DEADLOCK (EXTENDED status:) Invalid Spinlock allocation order. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient.
Functional Description	
OS service TryToGetSpinlock().	
Particularities and Limitations	
Pre-Condition: None Allocates the requested spinlock for the caller. If it is already locked, the function returns.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-12 TryToGetSpinlock

5.1.13 DisableAllInterrupts

Prototype	
`void DisableAllInterrupts (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
OS service DisableAllInterrupts()..	
Particularities and Limitations	
Pre-Condition: Not already in DisableAllInterrupts() sequence. Disables category 1 and category 2 ISRs. If timing protection is configured, the timing protection interrupt is not affected.	
Call context	
> TASK||ISR2|ISR1|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-13 DisableAllInterrupts

5.1.14 EnableAllInterrupts

Prototype	
`void EnableAllInterrupts (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
OS service EnableAllInterrupts().	
Particularities and Limitations	
Pre-Condition: In DisableAllInterrupts() sequence. Restores the state saved by DisableAllInterrupts().	
Call context	
> TASK|ISR2|ISR1|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-14 EnableAllInterrupts

5.1.15 SuspendAllInterrupts

Prototype	
`void SuspendAllInterrupts (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
OS service SuspendAllInterrupts().	
Particularities and Limitations	
Pre-Condition: Not in DisableAllInterrupts() sequence. Saves the recognition status of all interrupts and disables all interrupts for which the hardware supports disabling. This API can be called nested.	
Call context	
> TASK|ISR2|ISR1|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-15 SuspendAllInterrupts

5.1.16 ResumeAllInterrupts

Prototype	
`void **ResumeAllInterrupts** (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
OS service ResumeAllInterrupts().	
Particularities and Limitations	
> Pre-Condition: In SuspendAllInterrupts() sequence.Pre-Condition: Correct nesting sequence of suspend interrupt API. Restores the recognition status of all interrupts saved by the SuspendAllInterrupts() service.	
Call context	
> TASK|ISR2|ISR1|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-16 ResumeAllInterrupts

5.1.17 SuspendOSInterrupts

Prototype	
`void SuspendOSInterrupts (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
OS service SuspendOSInterrupts().	
Particularities and Limitations	
Pre-Condition: Not in DisableAllInterrupts() sequence. Saves the recognition status of interrupts of category 2 and disables the recognition of these interrupts. This API can be called nested.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-17 SuspendOSInterrupts

5.1.18 ResumeOSInterrupts

Prototype	
`void ResumeOSInterrupts (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
OS service ResumeOSInterrupts().	
Particularities and Limitations	
> Pre-Condition: In SuspendOSInterrupts() sequence.Pre-Condition: Correct nesting sequence of suspend interrupt API. Restores the recognition status of interrupts saved by the SuspendOSInterrupts() service.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-18 ResumeOSInterrupts

5.1.19 ActivateTask

Prototype	
`StatusType` **`ActivateTask`** `(TaskType TaskID)`	
Parameter	
TaskID [in]	The task which shall be activated.
Return code	
StatusType	> E_OK No error. > E_OS_LIMIT Too many task activations. > E_OS_ID (EXTENDED status:) Invalid TaskID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given task's owner application is not accessible.
Functional Description	
OS service ActivateTask().	
Particularities and Limitations	
Pre-Condition: None The task TaskID is transferred from the SUSPENDED state into the READY state. The operating system ensures that the task code is being executed from the first statement.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-19 ActivateTask

5.1.20 TerminateTask

Prototype	
`StatusType` **`TerminateTask`** `(void)`	
Parameter	
void	none
Return code	
StatusType	> E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_RESOURCE (EXTENDED status:) Task still occupies resources. > E_OS_SPINLOCK (EXTENDED status:) Task still holds spinlocks. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service TerminateTask().	
Particularities and Limitations	
Pre-Condition: None This service causes the termination of the calling task. The calling task is transferred from the RUNNING state into the SUSPENDED state. This service only returns in case it detects an error.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-20 TerminateTask

5.1.21 ChainTask

Prototype	
`StatusType ChainTask (TaskType TaskID)`	
Parameter	
TaskID [in]	The task which shall be activated.
Return code	
StatusType	> E_OS_LIMIT Too many task activations. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_RESOURCE (EXTENDED status:) Task still occupies resources. > E_OS_SPINLOCK (EXTENDED status:) Task still holds spinlocks. > E_OS_ID (EXTENDED status:) Invalid TaskID. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given task's owner application is not accessible.
Functional Description	
OS service ChainTask().	
Particularities and Limitations	
Pre-Condition: None After termination of the calling task the given task is activated. This service only returns in case it detects an error.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-21 ChainTask

5.1.22 Schedule

Prototype	
`StatusType **Schedule** (void)`	
Parameter	
void	none
Return code	
StatusType	> E_OK No Error. > E_OS_CALLEVEL (EXTENDED status:) The service was called from any context which is not allowed. > E_OS_RESOURCE (EXTENDED status:) The service was called from a task which holds an OS resource. > E_OS_SPINLOCK (EXTENDED status:) The service was called from a task which holds a spinlock. > E_OS_DISABLEDINT (EXTENDED status:) The service was called with disabled interrupts.
Functional Description	
OS service Schedule().	
Particularities and Limitations	
Pre-Condition: Interrupts are enabled.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-22 Schedule

5.1.23 GetTaskID

Prototype	
`StatusType **GetTaskID** (TaskRefType TaskID)`	
Parameter	
TaskID [out]	The current task ID.
Return code	
StatusType	> E_OK No error. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service GetTaskID().	
Particularities and Limitations	
Pre-Condition: None Returns the ID of the task which is currently RUNNING on the local core.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-23 GetTaskID

5.1.24 GetTaskState

Prototype	
`FUNC(StatusType, OS_CODE) GetTaskState (TaskType TaskID, TaskStateRefType State)`	
Parameter	
TaskID [in]	The task to be queried.
State [out]	The task's state.
Return code	
StatusType	> E_OK No error. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_ID (EXTENDED status:) Invalid TaskID. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given task's owner application is not accessible.
Functional Description	
OS service GetTaskState().	
Particularities and Limitations	
Pre-Condition: The given task has to be assigned to the current core. Returns the current scheduling state of a task (RUNNING, READY, ...).	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-24 GetTaskState

5.1.25 GetISRID

Prototype	
`ISRType GetISRID (void)`	
Parameter	
void	none
Return code	
ISRType	> Identifier of running ISR INVALID_ISR If called from > invalid call-context, > from a task or > a hook which was called inside a task context.
Functional Description	
OS service GetISRID().	
Particularities and Limitations	
Pre-Condition: None Return the identifier of the currently executing ISR.	
Call context	
> TASK|ISR2|ERRHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-25 GetISRID

5.1.26 SetEvent

Prototype	
`StatusType SetEvent (TaskType TaskID, EventMaskType Mask)`	
Parameter	
TaskID [in]	The task which shall be modified.
Mask [in]	The events which shall be set.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status) Invalid TaskID. > E_OS_ACCESS (EXTENDED status). Task is no extended task > E_OS_ACCESS (Service Protection). Task's owner application is not accessible. Caller's access rights are not sufficient. > E_OS_STATE (EXTENDED status:) Events cannot be set as the referenced task is in the SUSPENDED state. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_SYS_DISABLED (EXTENDED status:) Events are not enabled in the configuration.
Functional Description	
OS service SetEvent().	
Particularities and Limitations	
Pre-Condition: None The events of the given task are set according to the given event mask.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-26 SetEvent

5.1.27 ClearEvent

Prototype	
`StatusType` **`ClearEvent`** `(EventMaskType Mask)`	
Parameter	
Mask [in]	The events which shall be set.
Return code	
StatusType	> E_OK No error. > E_OS_ACCESS (EXTENDED status:) Task is no extended task. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_SYS_DISABLED (EXTENDED status:) Events are not enabled in the configuration.
Functional Description	
OS service ClearEvent().	
Particularities and Limitations	
Pre-Condition: None The events of the calling task are cleared according to the given event mask.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-27 ClearEvent

5.1.28 GetEvent

Prototype	
`StatusType **GetEvent** (TaskType TaskID, EventMaskRefType Mask)`	
Parameter	
TaskID [in]	The task which shall be queried.
Mask [out]	Events which are set.
Return code	
StatusType	> E_OK No error. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_ID (EXTENDED status:) Invalid TaskID. > E_OS_ACCESS (EXTENDED status:) Task is no extended task. > E_OS_ACCESS (Service Protection:). Task's owner application is not accessible. Caller's access rights are not sufficient. > E_OS_STATE (EXTENDED status:) Referenced task is in SUSPENDED state. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_SYS_DISABLED (EXTENDED status:) Events are not enabled in the configuration.
Functional Description	
OS service GetEvent().	
Particularities and Limitations	
Pre-Condition: Task is assigned to the current core. This service returns the state of all event bits of the given task, not the events that the task is waiting for.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-28 GetEvent

5.1.29 WaitEvent

Prototype	
`StatusType **WaitEvent** (EventMaskType Mask)`	
Parameter	
Mask [in]	Mask of the events waited for.
Return code	
StatusType	> E_OK No error. > E_OS_ACCESS (EXTENDED status:) Task is no extended task. > E_OS_RESOURCE (EXTENDED status:) Task still occupies resources. > E_OS_SPINLOCK (EXTENDED status:) Task still holds spinlocks. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_SYS_DISABLED (EXTENDED status:) Events are not enabled in the configuration.
Functional Description	
OS service WaitEvent().	
Particularities and Limitations	
Pre-Condition: None The state of the current task is set to WAITING, unless at least one of the given events is set.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-29 WaitEvent

5.1.30 IncrementCounter

Prototype	
`StatusType IncrementCounter (CounterType CounterID)`	
Parameter	
CounterID [in]	The counter to be incremented.
Return code	
StatusType	> E_OK No Error. > E_OS_ID (EXTENDED status:) CounterID is not a valid software counter ID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_CORE (EXTENDED status:) The given object belongs to a foreign core. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given counter's owner application is not accessible. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service IncrementCounter().	
Particularities and Limitations	
Pre-Condition: None	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-30 IncrementCounter

5.1.31 GetCounterValue

Prototype	
`StatusType GetCounterValue (CounterType CounterID, TickRefType Value)`	
Parameter	
CounterID [in]	The counter to be read.
Value [out]	Contains the current tick value of the counter.
Return code	
StatusType	> E_OK No Error. > E_OS_ID (EXTENDED status:) Invalid CounterID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_ACCESS (Service Protection:) Counter's owner application is not accessible or caller's access rights are not sufficient. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service GetCounterValue().	
Particularities and Limitations	
Pre-Condition: None	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-31 GetCounterValue

5.1.32 GetElapsedValue

Prototype	
`FUNC(StatusType, OS_CODE) GetElapsedValue (CounterType CounterID, TickRefType Value, TickRefType ElapsedValue)`	
Parameter	
CounterID [in]	The counter to be read.
Value [in,out]	**in:** The previously read tick value of the counter. **out:** The current tick value of the counter.
ElapsedValue [out]	The difference to the previous read value.
Return code	
StatusType	> E_OK No Error. > E_OS_ID (EXTENDED status:) Invalid CounterID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_VALUE (EXTENDED status:) The given Value was not valid. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_ACCESS (Service Protection:) Counter's owner application is not accessible or caller's access rights are not sufficient. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service GetElapsedValue().	
Particularities and Limitations	
Pre-Condition: None	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-32 GetElapsedValue

5.1.33 GetAlarmBase

Prototype	
`FUNC(StatusType, OS_CODE) **GetAlarmBase** (AlarmType AlarmID, AlarmBaseRefType Info)`	
Parameter	
AlarmID [in]	Reference to the alarm element.
Info [out]	Contains information about the counter on successful return.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid AlarmID. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient.
Functional Description	
OS service GetAlarmBase().	
Particularities and Limitations	
Pre-Condition: Given object pointer(s) are valid. The system service GetAlarmBase reads the alarm base characteristics. The return value Info is a structure in which the information of data type AlarmBaseType is stored.	
Call context	
> TASK|ISR2|PRETHOOK|POSTTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-33 GetAlarmBase

5.1.34 GetAlarm

<table>
<tr><th colspan="2">Prototype</th></tr>
<tr><td colspan="2"><code>FUNC(StatusType, OS_CODE) GetAlarm (AlarmType AlarmID, TickRefType Tick)</code></td></tr>
<tr><th colspan="2">Parameter</th></tr>
<tr><td>AlarmID [in]</td><td>Reference to the alarm element.</td></tr>
<tr><td>Tick [out]</td><td>Relative value in ticks before the alarm expires.</td></tr>
<tr><th colspan="2">Return code</th></tr>
<tr><td>StatusType</td><td>> E_OK No error. E_OS_NOFUNC Alarm is not in use.
> E_OS_ID (EXTENDED status:) Invalid AlarmID.
> E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL.
> E_OS_CALLEVEL (EXTENDED status:) Called from invalid context.
> E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
> E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given task's owner application is not accessible.</td></tr>
<tr><th colspan="2">Functional Description</th></tr>
<tr><td colspan="2">OS service GetAlarm().</td></tr>
<tr><th colspan="2">Particularities and Limitations</th></tr>
<tr><td colspan="2">The given alarm is assigned to the local core.
It is up to the application to decide whether for example a CancelAlarm may still be useful. If AlarmID is not in use, Tick is not defined. Allowed on task level, ISR, and in several hook routines.</td></tr>
<tr><th colspan="2">Call context</th></tr>
<tr><td colspan="2">> TASK|ISR2|PRETHOOK|POSTTHOOK
> This function is Synchronous
> This function is Reentrant</td></tr>
</table>

Table 5-34 GetAlarm

5.1.35 SetRelAlarm

Prototype	
`StatusType SetRelAlarm (AlarmType AlarmID, TickType Increment, TickType Cycle)`	
Parameter	
AlarmID [in]	Reference to the alarm element.
Increment [in]	Relative value in ticks.
Cycle [in]	Cycle value in case of cyclic alarm. In case of single alarms, cycle shall be zero.
Return code	
StatusType	> E_OK No error. > E_OS_STATE Alarm is already in use. > E_OS_ID (EXTENDED status:) Invalid AlarmID. > E_OS_VALUE (EXTENDED status:) Returned if: Value of increment is zero. Value of Increment outside of the admissible limits (lower than zero or greater than maxallowedvalue). Value of Cycle unequal to 0 and outside of the admissible counter limits (less than mincycle or greater than maxallowedvalue). > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or Given alarm's owner application is not accessible.
Functional Description	
OS service SetRelAlarm().	
Particularities and Limitations	
Pre-Condition: None The system service occupies the alarm AlarmID element. After increment ticks have elapsed, the task assigned to the alarm AlarmID is activated or the assigned event (only for extended tasks) is set or the alarm-callback routine is called.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-35 SetRelAlarm

5.1.36 SetAbsAlarm

Prototype	
`StatusType **SetAbsAlarm** (AlarmType AlarmID, TickType Start, TickType Cycle)`	
Parameter	
AlarmID [in]	Reference to the alarm element.
Start [in]	Absolute value in ticks.
Cycle [in]	Cycle value in case of cyclic alarm. In case of single alarms, cycle shall be zero.
Return code	
StatusType	> E_OK No error. > E_OS_STATE Alarm is already in use. > E_OS_ID (EXTENDED status:) Invalid AlarmID. > E_OS_VALUE (EXTENDED status:) Returned if: Value of Start outside of the admissible counter limit (less than zero or greater than maxallowedvalue). Value of Cycle unequal to 0 and outside of the admissible counter limits (less than mincycle or greater than maxallowedvalue). > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given alarm's owner application is not accessible.
Functional Description	
OS service SetAbsAlarm().	
Particularities and Limitations	
Pre-Condition: None The system service occupies the alarm AlarmID element. When start ticks are reached, the task assigned to the alarm AlarmID is activated or the assigned event (only for extended tasks) is set or the alarm-callback routine is called.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-36 SetAbsAlarm

5.1.37 CancelAlarm

Prototype	
`StatusType **CancelAlarm** (AlarmType AlarmID)`	
Parameter	
AlarmID [in]	Reference to the alarm element.
Return code	
StatusType	> E_OK No error. > E_OS_NOFUNC Alarm is not in use. > E_OS_ID (EXTENDED status:) Invalid AlarmID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given alarm's owner application is not accessible.
Functional Description	
OS service CancelAlarm().	
Particularities and Limitations	
Pre-Condition: None The system service cancels the alarm AlarmID.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-37 CancelAlarm

5.1.38 GetResource

Prototype	
`StatusType **GetResource** (ResourceType ResID)`	
Parameter	
ResID [in]	The resource which shall be occupied.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid ResID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_CORE (EXTENDED status:) The given object belongs to a foreign core. > E_OS_ACCESS (EXTENDED status:) Statically assigned priority of the caller is higher than the calculated ceiling priority. Attempt to get a resource which is already occupied. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service GetResource().	
Particularities and Limitations	
Pre-Condition: None This API serves to enter critical sections in the code. A critical section shall always be left using ReleaseResource().	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-38 GetResource

5.1.39 ReleaseResource

Prototype	
`StatusType **ReleaseResource** (ResourceType ResID)`	
Parameter	
ResID [in]	The resource which shall be released.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid ResID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_CORE (EXTENDED status:) The given object belongs to a foreign core. > E_OS_NOFUNC (EXTENDED status:) Attempt to release a resource which has not been occupied by the caller before. Attempt to release a nested resource in wrong order. Spinlock and Resource API not used in LIFO order. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service ReleaseResource().	
Particularities and Limitations	
This API is the counterpart of GetResource() and serves to leave critical sections in the code.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-39 ReleaseResource

5.1.40 StartScheduleTableRel

Prototype	
`StatusType **StartScheduleTableRel** (ScheduleTableType ScheduleTableID, TickType Offset)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table to be started.
Offset [in]	The relative offset when the schedule table shall be started.
Return code	
StatusType	> E_OK No error. > E_OS_STATE (EXTENDED status:) Schedule table has already been started. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_VALUE (EXTENDED status:) Offset is bigger than (OsCounterMaxAllowedValue - InitialOffset) or is equal to zero > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service StartScheduleTableRel().	
Particularities and Limitations	
Pre-Condition: None The schedule table is started at a relative offset to the current time.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-40 StartScheduleTableRel

5.1.41 StartScheduleTableAbs

Prototype	
`StatusType` **`StartScheduleTableAbs`** `(ScheduleTableType ScheduleTableID, TickType Start)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table to be started
Start [in]	The absolute time when the schedule table shall be started
Return code	
StatusType	> E_OK No error. > E_OS_STATE (EXTENDED status:) Schedule table has already been started. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_VALUE (EXTENDED status:) Offset is bigger than OsCounterMaxAllowedValue. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service StartScheduleTableAbs().	
Particularities and Limitations	
Pre-Condition: None The schedule table is started at an absolute time.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-41 StartScheduleTableAbs

5.1.42 StopScheduleTable

Prototype	
`StatusType **StopScheduleTable** (ScheduleTableType ScheduleTableID)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table to be stopped.
Return code	
StatusType	> E_OK No error. > E_OS_NOFUNC (EXTENDED status:) Schedule table has already been stopped. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or Given schedule table's owner application is not accessible.
Functional Description	
OS service StopScheduleTable().	
Particularities and Limitations	
Pre-Condition: None The schedule table is stopped immediately.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-42 StopScheduleTable

5.1.43 NextScheduleTable

Prototype	
`StatusType` **`NextScheduleTable`** `(ScheduleTableType ScheduleTableID_From, ScheduleTableType ScheduleTableID_To)`	
Parameter	
ScheduleTableID_From [in]	The ID of the schedule table which is requested to stop at its end
ScheduleTableID_To [in]	The ID of the schedule table which starts after the other one has stopped
Return code	
StatusType	> E_OK No error. > E_OS_NOFUNC (EXTENDED status:) Schedule table ScheduleTableID_From has not been started. > E_OS_STATE (EXTENDED status:) Schedule table ScheduleTableID_To has already been requested to start at the end of another schedule table. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID_From/To. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service NextScheduleTable().	
Particularities and Limitations	
Pre-Condition: None Requests the switch of schedule table processing from one schedule table to another after the first one has reached its end.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-43 NextScheduleTable

5.1.44 GetScheduleTableStatus

Prototype	
`FUNC(StatusType, OS_CODE) GetScheduleTableStatus ( ScheduleTableType ScheduleTableID, ScheduleTableStatusRefType ScheduleStatus)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table for which the status shall be requested.
ScheduleStatus [out]	Reference to ScheduleTableStatusType.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_PARAM_POINTER (EXTENDED status:) ScheduleStatus is a pointer to null. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service GetScheduleTableStatus().	
Particularities and Limitations	
Pre-Condition: None This service queries the state of a schedule table (also with respect to synchronization).	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-44 GetScheduleTableStatus

5.1.45 StartScheduleTableSynchron

Prototype	
`StatusType **StartScheduleTableSynchron** (ScheduleTableType ScheduleTableID)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table which shall start synchronously
Return code	
StatusType	> E_OK No error. > E_OS_STATE (EXTENDED status:) Schedule table ScheduleTableID has already been started. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID. > E_OS_CORE (EXTENDED status:) The given object belongs to a foreign core. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service StartScheduleTableSynchron().	
Particularities and Limitations	
Pre-Condition: None This service starts an explicitly synchronized schedule table synchronously. As a result the schedule table enters the state SCHEDULETABLE_WAITING and waits for a synchronization count to be provided.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-45 StartScheduleTableSynchron

5.1.46 SyncScheduleTable

Prototype	
`StatusType` **`SyncScheduleTable`** `(ScheduleTableType ScheduleTableID, TickType Value)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table to the synchronized
Value [in]	The current value of the synchronization counter
Return code	
StatusType	> E_OK No error. > E_OS_STATE (EXTENDED status:) The state of the schedule table ScheduleTableId is equal to SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID. > E_OS_CORE (EXTENDED status:) The given object belongs to a foreign core. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_VALUE (EXTENDED status:) The Value is out of range > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service SyncScheduleTable().	
Particularities and Limitations	
Pre-Condition: None This service provides the schedule table with a synchronization count and starts the synchronization.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-46 SyncScheduleTable

5.1.47 SetScheduleTableAsync

Prototype	
`StatusType` **`SetScheduleTableAsync`** `(ScheduleTableType ScheduleTableID)`	
Parameter	
ScheduleTableID [in]	The ID of the schedule table which shall no longer be synchronized.
Return code	
StatusType	> E_OK No error. > E_OS_STATE (EXTENDED status:) Current state of ScheduleTableID is SCHEDULETABLE_STOPPED, SCHEDULETABLE_NEXT or SCHEDULETABLE_WAITING. > E_OS_ID (EXTENDED status:) Invalid ScheduleTableID or OsScheduleTblSyncStrategy of ScheduleTableID is not equal to EXPLICIT > E_OS_CORE (EXTENDED status:) The given object belongs to a foreign core. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given schedule table's owner application is not accessible.
Functional Description	
OS service SetScheduleTableAsync().	
Particularities and Limitations	
Pre-Condition: None This service stops the synchronization of a schedule table.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-47 SetScheduleTableAsync

5.1.48 GetApplicationID

Prototype	
`ApplicationType` **`GetApplicationID`** `(void)`	
Parameter	
void	none
Return code	
ApplicationType	Identifier of the OS-Application.
Functional Description	
OS service GetApplicationID().	
Particularities and Limitations	
Pre-Condition: None This service determines the OS-Application where the caller (Task/ISR/Hook) originally belongs to (was configured to). All system objects (e.g. system hooks, idle task, ...) belong to kernel applications. Kernel applications are regular applications and have valid identifiers. Therefore INVALID_OSAPPLICATION is never returned because there is always a valid application active.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-48 GetApplicationID

5.1.49 GetCurrentApplicationID

Prototype	
`ApplicationType` **`GetCurrentApplicationID`** `(void)`	
Parameter	
void	none
Return code	
ApplicationType	Identifier of the OS-Application.
Functional Description	
OS service GetCurrentApplicationID().	
Particularities and Limitations	
Pre-Condition: None This service determines the OS-Application where the caller (Task/ISR/Hook) of the service is currently executing. Note that, if the caller is not within a CallTrustedFunction() call, the value is equal to the result of GetApplicationID().	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-49 GetCurrentApplicationID

5.1.50 GetApplicationState

Prototype	
`StatusType **GetApplicationState** (ApplicationType Application, ApplicationStateRefType Value)`	
Parameter	
Application [in]	The OS-Application from which the state is requested.
Value [out]	The current state of the application.
Return code	
StatusType	> E_OK No error. > E_OS_ID (EXTENDED status:) Invalid Application. > E_OS_PARAM_POINTER (EXTENDED status:) Given pointer is NULL. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service GetApplicationState().	
Particularities and Limitations	
Pre-Condition: None This service returns the current state of an OS-Application.	
Call context	
> TASK|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-50 GetApplicationState

5.1.51 CheckObjectAccess

Prototype	
`ObjectAccessType **CheckObjectAccess** (ApplicationType ApplID, ObjectTypeType ObjectType, Os_ObjectIdType ObjectID)`	
Parameter	
ApplID [in]	OS-Application identifier.
ObjectType [in]	Type of the following parameter.
ObjectID [in]	The object to be examined.
Return code	
ObjectAccessType	> ACCESS if the ApplID has access to the object. NO_ACCESS If: > - ApplID doesn't have access to the object. > - ApplID is invalid. > - ObjectID is invalid.
Functional Description	
OS service CheckObjectAccess().	
Particularities and Limitations	
Pre-Condition: None This service determines if the OS-Application, given by ApplID, is allowed to use the IDs of a Task, Resource, Counter, Alarm or Schedule Table in API calls.	
Call context	
> TASK|ISR2|ERRHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-51 CheckObjectAccess

5.1.52 CheckObjectOwnership

Prototype	
`ApplicationType` **`CheckObjectOwnership`** `(ObjectTypeType ObjectType, Os_ObjectIdType ObjectID)`	
Parameter	
ObjectType [in]	Type of the following parameter.
ObjectID [in]	The object to be examined.
Return code	
ApplicationType	Identifier of the owner OS-Application. INVALID_OSAPPLICATION if the object does not exist.
Functional Description	
OS service CheckObjectOwnership().	
Particularities and Limitations	
Pre-Condition: None This service determines to which OS-Application a given Task, ISR, Counter, Alarm or Schedule Table belongs.	
Call context	
> TASK|ISR2|ERRHOOK|PROTHOOK > This function is Synchronous > This function is Reentrant	

Table 5-52 CheckObjectOwnership

5.1.53 AllowAccess

Prototype	
`StatusType AllowAccess (void)`	
Parameter	
void	none
Return code	
StatusType	> E_OK No error. > E_OS_STATE The application is not in the restarting state. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
Functional Description	
OS service AllowAccess().	
Particularities and Limitations	
Pre-Condition: None This service sets the state of the current OS-Application from APPLICATION_RESTARTING to APPLICATION_ACCESSIBLE.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-53 AllowAccess

5.1.54 TerminateApplication

<table>
<tr><th colspan="2">Prototype</th></tr>
<tr><td colspan="2"><code>StatusType TerminateApplication (ApplicationType Application, RestartType RestartOption)</code></td></tr>
<tr><th colspan="2">Parameter</th></tr>
<tr><td>Application [in]</td><td>The identifier of the OS-Application to be terminated. If the caller belongs to Application the call results in a self-termination.</td></tr>
<tr><td>RestartOption [in]</td><td>Either RESTART for doing a restart of the OS-Application or NO_RESTART if OS-Application shall not be restarted.</td></tr>
<tr><th colspan="2">Return code</th></tr>
<tr><td>StatusType</td><td>> E_OK No errors.
> E_OS_STATE The state of Application does not allow terminating it if: The application is already terminated. Or the application is restarting AND the caller does not belong to the application. Or the application is restarting AND the caller does belong to the application AND the RestartOption is RESTART.
> E_OS_ID (EXTENDED status:) Application was not valid.
> E_OS_VALUE (EXTENDED status:) RestartOption was neither RESTART nor NO_RESTART.
> E_OS_CALLEVEL (EXTENDED status:) Called from invalid context.
> E_OS_ACCESS (EXTENDED status:) The caller belongs to a non-trusted OS-Application AND the caller does not belong to given Application TerminateApplication() shall return.
> E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.</td></tr>
<tr><th colspan="2">Functional Description</th></tr>
<tr><td colspan="2">OS service TerminateApplication().</td></tr>
<tr><th colspan="2">Particularities and Limitations</th></tr>
<tr><td colspan="2">Pre-Condition: None
This service terminates the OS-Application to which the calling Task/ISR/application specific error hook belongs.</td></tr>
<tr><th colspan="2">Call context</th></tr>
<tr><td colspan="2">> TASK|ISR2|ERRHOOK
> This function is Synchronous
> This function is Reentrant</td></tr>
</table>

Table 5-54 TerminateApplication

5.1.55 CallTrustedFunction

Prototype	
`StatusType` **`CallTrustedFunction`** `(TrustedFunctionIndexType FunctionIndex, TrustedFunctionParameterRefType FunctionParams)`	
Parameter	
FunctionIndex [in]	Index of the function to be called.
FunctionParams [in]	Pointer to the parameters for the function. If no parameters are provided, a NULL pointer has to be passed.
Return code	
StatusType	> E_OK No error. > E_OS_SERVICEID No function defined for this index. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_ACCESS (EXTENDED status:) The given object belongs to a foreign core. > E_OS_ACCESS (Service Protection:) The owner application is not accessible.
Functional Description	
OS service CallTrustedFunction().	
Particularities and Limitations	
Pre-Condition: None Each trusted OS-Application may export services which are callable from other OS-Applications.	
Call context	
> TASK|ISR2 > This function is Synchronous > This function is Reentrant	

Table 5-55 CallTrustedFunction

5.1.56 Check Task Memory Access

Prototype	
`FUNC(AccessType, OS_CODE) CheckTaskMemoryAccess( TaskType TaskID, MemoryStartAddressType Address, MemorySizeType Size )`	
Parameter	
`TaskID`	ID of task
`Address`	Start address of checked address range
`Size`	Size of checked address range
Return code	
`AccessType`	Returns the access rights of the Task to the given address range OS_MEM_ACCESS_TYPE_NON No access, invalid TaskID or address range overflow.
Functional Description	
The service distinguishes the memory access rights of a given Task.	
Particularities and Limitations	

> The access checks are based upon the "OsAccessCheckRegion" configuration objects.
> The return value of this functions is typically used with the AUTOSAR OS specified macros
  > OSMEMORY_IS_READABLE
  > OSMEMORY_IS_WRITEABLE
  > OSMEMORY_IS_EXECUTABLE
  > OSMEMORY_IS_STACKSPACE	

Table 5-56 API Service CheckTaskMemoryAccess

5.1.57 Check ISR Memory Access

Prototype	
`FUNC(AccessType, OS_CODE) CheckISRMemoryAccess( ISRType ISRID, MemoryStartAddressType Address, MemorySizeType Size )`	
Parameter	
`ISRID`	ID of category 2 ISR
`Address`	Start address of checked address range
`Size`	Size of checked address range
Return code	
`AccessType`	Returns the access rights of the ISR to the given address range OS_MEM_ACCESS_TYPE_NON No access, invalid TaskID or address range overflow.
Functional Description	
The service distinguishes the memory access rights of a given category 2 ISR	
Particularities and Limitations	

> The access checks are based upon the “OsAccessCheckRegion” configuration objects.
> The return value of this functions is typically used with the AUTOSAR OS specified macros
  > OSMEMORY_IS_READABLE
  > OSMEMORY_IS_WRITEABLE
  > OSMEMORY_IS_EXECUTABLE
  > OSMEMORY_IS_STACKSPACE	

Table 5-57 API Service CheckISRMemoryAccess

5.1.58 OSErrorGetServiceId

Prototype	
`OSServiceIdType` **`OSErrorGetServiceId`** `(void)`	
Parameter	
void	none
Return code	
OSServiceIdType	none
Functional Description	
OS service OSErrorGetServiceId().	
Particularities and Limitations	
Pre-Condition: None Provides the service identifier where the error has been risen.	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-58 OSErrorGetServiceId

5.1.59 OSError_Os_DisableInterruptSource_ISRID

Prototype	
`ISRType OSError_Os_DisableInterruptSource_ISRID (void)`	
Parameter	
void	none
Return code	
ISRType	Requested parameter value.
Functional Description	
Returns parameter ISRID of a faulty Os_DisableInterruptSource call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-59 OSError_Os_DisableInterruptSource_ISRID

5.1.60 OSError_Os_EnableInterruptSource_ISRID

Prototype	
`ISRType OSError_Os_EnableInterruptSource_ISRID (void)`	
Parameter	
void	none
Return code	
ISRType	Requested parameter value.
Functional Description	
Returns parameter ISRID of a faulty Os_EnableInterruptSource call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-60 OSError_Os_EnableInterruptSource_ISRID

5.1.61 OSError_Os_EnableInterruptSource_ClearPending

Prototype	
boolean **OSError_Os_EnableInterruptSource_ClearPending** (void)	
Parameter	
void	none
Return code	
boolean	Requested parameter value.
Functional Description	
Returns parameter ClearPending of a faulty Os_EnableInterruptSource call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-61 OSError_Os_EnableInterruptSource_ClearPending

5.1.62 OSError_Os_ClearPendingInterrupt_ISRID

Prototype	
ISRType **OSError_Os_ClearPendingInterrupt_ISRID** (void)	
Parameter	
void	none
Return code	
ISRType	Requested parameter value.
Functional Description	
Returns parameter ISRID of a faulty Os_ClearPendingInterrupt call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-62 OSError_Os_ClearPendingInterrupt_ISRID

5.1.63 OSError_Os_IsInterruptSourceEnabled_ISRID

Prototype	
ISRType **OSError_Os_IsInterruptSourceEnabled_ISRID** (void)	
Parameter	
void	none
Return code	
ISRType	Requested parameter value.
Functional Description	
Returns parameter ISRID of a faulty Os_IsInterruptSourceEnabled call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-63 OSError_Os_IsInterruptSourceEnabled_ISRID

5.1.64 OSError_Os_IsInterruptSourceEnabled_IsEnabled

Prototype	
boolean * **OSError_Os_IsInterruptSourceEnabled_IsEnabled** (void)	
Parameter	
void	none
Return code	
boolean *	Requested parameter value.
Functional Description	
Returns parameter IsEnabled of a faulty Os_IsInterruptSourceEnabled call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-64 OSError_Os_IsInterruptSourceEnabled_IsEnabled

5.1.65 OSError_Os_IsInterruptPending_ISRID

Prototype	
`ISRType` **`OSError_Os_IsInterruptPending_ISRID`** `(void)`	
Parameter	
void	none
Return code	
ISRType	Requested parameter value.
Functional Description	
Returns parameter ISRID of a faulty Os_IsInterruptPending call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-65 OSError_Os_IsInterruptPending_ISRID

5.1.66 OSError_Os_IsInterruptPending_IsPending

Prototype	
`boolean *` **`OSError_Os_IsInterruptPending_IsPending`** `(void)`	
Parameter	
void	none
Return code	
boolean *	Requested parameter value.
Functional Description	
Returns parameter IsPending of a faulty Os_IsInterruptPending_IsPending call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-66 OSError_Os_IsInterruptPending_IsPending

5.1.67 OSError_CallTrustedFunction_FunctionIndex

Prototype	
`TrustedFunctionIndexType **OSError_CallTrustedFunction_FunctionIndex** (void)`	
Parameter	
void	none
Return code	
TrustedFunctionIndexType	Requested parameter value.
Functional Description	
Returns parameter FunctionIndex of a faulty CallTrustedFunction call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-67 OSError_CallTrustedFunction_FunctionIndex

5.1.68 OSError_CallTrustedFunction_FunctionParams

Prototype	
`TrustedFunctionParameterRefType **OSError_CallTrustedFunction_FunctionParams** (void)`	
Parameter	
void	none
Return code	
TrustedFunctionParameterRefType	Requested parameter value.
Functional Description	
Returns parameter FunctionParams of a faulty CallTrustedFunction call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-68 OSError_CallTrustedFunction_FunctionParams

5.1.69 OSError_CallFastTrustedFunction_FunctionIndex

Prototype	
`Os_FastTrustedFunctionIndexType` **`OSError_CallFastTrustedFunction_FunctionIndex`** `(void)`	
Parameter	
void	none
Return code	
Os_FastTrustedFunctionIndexType	Requested parameter value.
Functional Description	
Returns parameter FunctionIndex of a faulty CallFastTrustedFunction call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-69 OSError_CallFastTrustedFunction_FunctionIndex

5.1.70 OSError_CallFastTrustedFunction_FunctionParams

Prototype	
`Os_FastTrustedFunctionParameterRefType` **`OSError_CallFastTrustedFunction_FunctionParams`** `(void)`	
Parameter	
void	none
Return code	
Os_FastTrustedFunctionParameterRefType	Requested parameter value.
Functional Description	
Returns parameter FunctionParams of a faulty CallFastTrustedFunction call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-70 OSError_CallFastTrustedFunction_FunctionParams

5.1.71 OSError_CallNonTrustedFunction_FunctionIndex

Prototype	
`Os_NonTrustedFunctionIndexType` **`OSError_CallNonTrustedFunction_FunctionIndex`** `(void)`	
Parameter	
void	none
Return code	
Os_NonTrustedFunctionIndexType	Requested parameter value.
Functional Description	
Returns parameter FunctionIndex of a faulty CallNonTrustedFunction call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-71 OSError_CallNonTrustedFunction_FunctionIndex

5.1.72 OSError_CallNonTrustedFunction_FunctionParams

Prototype	
`Os_NonTrustedFunctionParameterRefType` **`OSError_CallNonTrustedFunction_FunctionParams`** `(void)`	
Parameter	
void	none
Return code	
Os_NonTrustedFunctionParameterRefType	Requested parameter value.
Functional Description	
Returns parameter FunctionParams of a faulty CallNonTrustedFunction call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-71 OSError_CallNonTrustedFunction_FunctionParams

5.1.73 OSError_StartScheduleTableRel_ScheduleTableID

Prototype	
`ScheduleTableType` **`OSError_StartScheduleTableRel_ScheduleTableID`** `(void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty StartScheduleTableRel call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-72 OSError_StartScheduleTableRel_ScheduleTableID

5.1.74 OSError_StartScheduleTableRel_Offset

Prototype	
`TickType` **`OSError_StartScheduleTableRel_Offset`** `(void)`	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter Offset of a faulty StartScheduleTableRel call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-73 OSError_StartScheduleTableRel_Offset

5.1.75 OSError_StartScheduleTableAbs_ScheduleTableID

Prototype	
ScheduleTableType **OSError_StartScheduleTableAbs_ScheduleTableID** (void)	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty StartScheduleTableAbs call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-74 OSError_StartScheduleTableAbs_ScheduleTableID

5.1.76 OSError_StartScheduleTableAbs_Start

Prototype	
TickType **OSError_StartScheduleTableAbs_Start** (void)	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter Start of a faulty StartScheduleTableAbs call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-75 OSError_StartScheduleTableAbs_Start

5.1.77 OSError_StopScheduleTable_ScheduleTableID

Prototype	
`ScheduleTableType OSError_StopScheduleTable_ScheduleTableID (void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty StopScheduleTable call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-76 OSError_StopScheduleTable_ScheduleTableID

5.1.78 OSError_NextScheduleTable_ScheduleTableID_From

Prototype	
`ScheduleTableType OSError_NextScheduleTable_ScheduleTableID_From (void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID_From of a faulty NextScheduleTable call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-77 OSError_NextScheduleTable_ScheduleTableID_From

5.1.79 OSError_NextScheduleTable_ScheduleTableID_To

Prototype	
ScheduleTableType **OSError_NextScheduleTable_ScheduleTableID_To** (void)	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID_To of a faulty NextScheduleTable call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-79 OSError_NextScheduleTable_ScheduleTableID_To

5.1.80 OSError_StartScheduleTableSynchron_ScheduleTableID

Prototype	
ScheduleTableType **OSError_StartScheduleTableSynchron_ScheduleTableID** (void)	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty StartScheduleTableSynchron call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-80 OSError_StartScheduleTableSynchron_ScheduleTableID

5.1.81 OSError_SyncScheduleTable_ScheduleTableID

Prototype	
`ScheduleTableType` **`OSError_SyncScheduleTable_ScheduleTableID`** `(void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty SyncScheduleTable call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-81 OSError_SyncScheduleTable_ScheduleTableID

5.1.82 OSError_SyncScheduleTable_Value

Prototype	
`TickType` **`OSError_SyncScheduleTable_Value`** `(void)`	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter Value of a faulty SyncScheduleTable call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-82 OSError_SyncScheduleTable_Value

5.1.83 OSError_SetScheduleTableAsync_ScheduleTableID

Prototype	
`ScheduleTableType` **`OSError_SetScheduleTableAsync_ScheduleTableID`** `(void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty SetScheduleTableAsync call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-78 OSError_SetScheduleTableAsync_ScheduleTableID

5.1.84 OSError_GetScheduleTableStatus_ScheduleTableID

Prototype	
`ScheduleTableType` **`OSError_GetScheduleTableStatus_ScheduleTableID`** `(void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleTableID of a faulty GetScheduleTableStatus call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-79 OSError_GetScheduleTableStatus_ScheduleTableID

5.1.85 OSError_GetScheduleTableStatus_ScheduleStatus

Prototype	
`ScheduleTableStatusRefType` **`OSError_GetScheduleTableStatus_ScheduleStatus`** `(void)`	
Parameter	
void	none
Return code	
ScheduleTableType	Requested parameter value.
Functional Description	
Returns parameter ScheduleStatus of a faulty GetScheduleTableStatus call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-80 OSError_GetScheduleTableStatus_ScheduleStatus

5.1.86 OSError_IncrementCounter_CounterID

Prototype	
`CounterType` **`OSError_IncrementCounter_CounterID`** `(void)`	
Parameter	
void	none
Return code	
CounterType	Requested parameter value.
Functional Description	
Returns parameter CounterID of a faulty IncrementCounter call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-81 OSError_IncrementCounter_CounterID

5.1.87 OSError_GetCounterValue_CounterID

Prototype	
`CounterType` **`OSError_GetCounterValue_CounterID`** `(void)`	
Parameter	
void	none
Return code	
CounterType	Requested parameter value.
Functional Description	
Returns parameter CounterID of a faulty GetCounterValue call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-82 OSError_GetCounterValue_CounterID

5.1.88 OSError_GetCounterValue_Value

Prototype	
`TickRefType` **`OSError_GetCounterValue_Value`** `(void)`	
Parameter	
void	none
Return code	
TickRefType	Requested parameter value.
Functional Description	
Returns parameter Value of a faulty GetCounterValue call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-83 OSError_GetCounterValue_Value

5.1.89 OSError_GetElapsedValue_CounterID

Prototype	
`CounterType` **`OSError_GetElapsedValue_CounterID`** `(void)`	
Parameter	
void	none
Return code	
CounterType	Requested parameter value.
Functional Description	
Returns parameter CounterID of a faulty GetElapsedValue call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-84 OSError_GetElapsedValue_CounterID

5.1.90 OSError_GetElapsedValue_Value

Prototype	
`TickRefType` **`OSError_GetElapsedValue_Value`** `(void)`	
Parameter	
void	none
Return code	
TickRefType	Requested parameter value.
Functional Description	
Returns parameter Value of a faulty GetElapsedValue call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-90 OSError_GetElapsedValue_Value

5.1.91 OSError_GetElapsedValue_ElapsedValue

Prototype	
`TickRefType` **`OSError_GetElapsedValue_ElapsedValue`** `(void)`	
Parameter	
void	none
Return code	
TickRefType	Requested parameter value.
Functional Description	
Returns parameter ElapsedValue of a faulty GetElapsedValue call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-91 OSError_GetElapsedValue_ElapsedValue

5.1.92 OSError_TerminateApplication_Application

Prototype	
`ApplicationType OSError_TerminateApplication_Application (void)`	
Parameter	
void	none
Return code	
ApplicationType	Requested parameter value.
Functional Description	
Returns parameter Application of a faulty TerminateApplication call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-85 OSError_TerminateApplication_Application

5.1.93 OSError_TerminateApplication_RestartOption

Prototype	
`RestartType OSError_TerminateApplication_RestartOption (void)`	
Parameter	
void	none
Return code	
RestartType	Requested parameter value.
Functional Description	
Returns parameter RestartOption of a faulty TerminateApplication call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-86 OSError_TerminateApplication_RestartOption

5.1.94 OSError_GetApplicationState_Application

Prototype	
`ApplicationType` **`OSError_GetApplicationState_Application`** `(void)`	
Parameter	
void	none
Return code	
ApplicationType	Requested parameter value.
Functional Description	
Returns parameter Application of a faulty GetApplicationState call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-87 OSError_GetApplicationState_Application

5.1.95 OSError_GetApplicationState_Value

Prototype	
`ApplicationStateRefType` **`OSError_GetApplicationState_Value`** `(void)`	
Parameter	
void	none
Return code	
ApplicationStateRefType	Requested parameter value.
Functional Description	
Returns parameter Value of a faulty GetApplicationState call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-88 OSError_GetApplicationState_Value

5.1.96 OSError_GetSpinlock_SpinlockId

Prototype	
SpinlockIdType **OSError_GetSpinlock_SpinlockId** (void)	
Parameter	
void	none
Return code	
SpinlockIdType	Requested parameter value.
Functional Description	
Returns parameter SpinlockId of a faulty GetSpinlock call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-89 OSError_GetSpinlock_SpinlockId

5.1.97 OSError_ReleaseSpinlock_SpinlockId

Prototype	
SpinlockIdType **OSError_ReleaseSpinlock_SpinlockId** (void)	
Parameter	
void	none
Return code	
SpinlockIdType	Requested parameter value.
Functional Description	
Returns parameter SpinlockId of a faulty ReleaseSpinlock call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-90 OSError_ReleaseSpinlock_SpinlockId

5.1.98 OSError_TryToGetSpinlock_SpinlockId

Prototype	
`SpinlockIdType OSError_TryToGetSpinlock_SpinlockId (void)`	
Parameter	
void	none
Return code	
SpinlockIdType	Requested parameter value.
Functional Description	
Returns parameter SpinlockId of a faulty TryToGetSpinlock call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-91 OSError_TryToGetSpinlock_SpinlockId

5.1.99 OSError_TryToGetSpinlock_Success

Prototype	
`TryToGetSpinlockType const * OSError_TryToGetSpinlock_Success (void)`	
Parameter	
void	none
Return code	
TryToGetSpinlockType const *	Requested parameter value.
Functional Description	
Returns parameter Success of a faulty TryToGetSpinlock call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-92 OSError_TryToGetSpinlock_Success

5.1.100 OSError_ControlIdle_CoreID

Prototype	
`CoreIdType` **`OSError_ControlIdle_CoreID`** `(void)`	
Parameter	
void	none
Return code	
CoreIdType	Requested parameter value.
Functional Description	
Returns parameter CoreID of a faulty ControlIdle call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-100 OSError_ControlIdle_CoreID

5.1.101 OSError_Os_GetExceptionContext_Context

Prototype	
`Os_ExceptionContextRefType` **`OSError_Os_GetExceptionContext_Context`** `(void)`	
Parameter	
void	none
Return code	
Os_ExceptionContextRefType	Requested parameter value.
Functional Description	
Returns parameter Context of a faulty Os_GetExceptionContext call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-101 OSError_Os_GetExceptionContext_Context

5.1.102 OSError_Os_SetExceptionContext_Context

Prototype	
`Os_ExceptionContextRefType` **`OSError_Os_SetExceptionContext_Context`** `(void)`	
Parameter	
void	none
Return code	
Os_ExceptionContextRefType	Requested parameter value.
Functional Description	
Returns parameter Context of a faulty Os_SetExceptionContext call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-93 OSError_Os_SetExceptionContext_Context

5.1.103 OSError_ControlIdle_IdleMode

Prototype	
`IdleModeType` **`OSError_ControlIdle_IdleMode`** `(void)`	
Parameter	
void	none
Return code	
IdleModeType	Requested parameter value.
Functional Description	
Returns parameter IdleMode of a faulty ControlIdle call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-94 OSError_ControlIdle_IdleMode

5.1.104 OSError_IocSend_IN

Prototype	
`void const * OSError_IocSend_IN (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter IN of a faulty IocSend call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-95 OSError_IocSend_IN

5.1.105 OSError_IocWrite_IN

Prototype	
`void const * OSError_IocWrite_IN (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter IN of a faulty IocWrite call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-96 OSError_IocWrite_IN

5.1.106 OSError_IocSendGroup_IN

Prototype	
`void const * OSError_IocSendGroup_IN (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter IN of a faulty IocSendGroup call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-97 OSError_IocSendGroup_IN

5.1.107 OSError_IocWriteGroup_IN

Prototype	
`void const * OSError_IocWriteGroup_IN (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter IN of a faulty IocWriteGroup call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-98 OSError_IocWriteGroup_IN

5.1.108 OSError_IocReceive_OUT

Prototype	
`void const * OSError_IocReceive_OUT (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter OUT of a faulty IocReceive call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-99 OSError_IocReceive_OUT

5.1.109 OSError_IocRead_OUT

Prototype	
`void const * OSError_IocRead_OUT (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter OUT of a faulty IocRead call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-100 OSError_IocRead_OUT

5.1.110 OSError_IocReceiveGroup_OUT

Prototype	
`void const * OSError_IocReceiveGroup_OUT (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter OUT of a faulty IocReceiveGroup call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-101 OSError_IocReceiveGroup_OUT

5.1.111 OSError_IocReadGroup_OUT

Prototype	
`void const * OSError_IocReadGroup_OUT (void)`	
Parameter	
void	none
Return code	
void const *	Requested parameter value.
Functional Description	
Returns parameter OUT of a faulty IocReadGroup call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-102 OSError_IocReadGroup_OUT

5.1.112 OSError_StartOS_Mode

Prototype	
`AppModeType **OSError_StartOS_Mode** (void)`	
Parameter	
void	none
Return code	
AppModeType	Requested parameter value.
Functional Description	
Returns parameter Mode of a faulty StartOS call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-103 OSError_StartOS_Mode

5.1.113 OSError_ActivateTask_TaskID

Prototype	
`TaskType **OSError_ActivateTask_TaskID** (void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty ActivateTask call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-104 OSError_ActivateTask_TaskID

5.1.114 OSError_ChainTask_TaskID

Prototype	
`TaskType` **`OSError_ChainTask_TaskID`** `(void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty ChainTask call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-105 OSError_ChainTask_TaskID

5.1.115 OSError_GetTaskID_TaskID

Prototype	
`TaskRefType` **`OSError_GetTaskID_TaskID`** `(void)`	
Parameter	
void	none
Return code	
TaskRefType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty GetTaskID call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-106 OSError_GetTaskID_TaskID

5.1.116 OSError_GetTaskState_TaskID

Prototype	
`TaskType` **`OSError_GetTaskState_TaskID`** `(void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty GetTaskState call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-107 OSError_GetTaskState_TaskID

5.1.117 OSError_GetTaskState_State

Prototype	
`TaskStateRefType` **`OSError_GetTaskState_State`** `(void)`	
Parameter	
void	none
Return code	
TaskStateRefType	Requested parameter value.
Functional Description	
Returns parameter State of a faulty GetTaskState call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-108 OSError_GetTaskState_State

5.1.118 OSError_SetEvent_TaskID

Prototype	
`TaskType **OSError_SetEvent_TaskID** (void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty SetEvent call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-109 OSError_SetEvent_TaskID

5.1.119 OSError_SetEvent_Mask

Prototype	
`EventMaskType **OSError_SetEvent_Mask** (void)`	
Parameter	
void	none
Return code	
EventMaskType	Requested parameter value.
Functional Description	
Returns parameter Mask of a faulty SetEvent call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-110 OSError_SetEvent_Mask

5.1.120 OSError_ClearEvent_Mask

Prototype	
`EventMaskType` **`OSError_ClearEvent_Mask`** `(void)`	
Parameter	
void	none
Return code	
EventMaskType	Requested parameter value.
Functional Description	
Returns parameter Mask of a faulty ClearEvent call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-111 OSError_ClearEvent_Mask

5.1.121 OSError_GetEvent_TaskID

Prototype	
`TaskType` **`OSError_GetEvent_TaskID`** `(void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty GetEvent call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-112 OSError_GetEvent_TaskID

5.1.122 OSError_GetEvent_Mask

Prototype	
`EventMaskRefType` **`OSError_GetEvent_Mask`** `(void)`	
Parameter	
void	none
Return code	
EventMaskRefType	Requested parameter value.
Functional Description	
Returns parameter Mask of a faulty GetEvent call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-113 OSError_GetEvent_Mask

5.1.123 OSError_WaitEvent_Mask

Prototype	
`EventMaskType` **`OSError_WaitEvent_Mask`** `(void)`	
Parameter	
void	none
Return code	
EventMaskType	Requested parameter value.
Functional Description	
Returns parameter Mask of a faulty WaitEvent call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-114 OSError_WaitEvent_Mask

5.1.124 OSError_GetAlarmBase_AlarmID

Prototype	
`AlarmType` **`OSError_GetAlarmBase_AlarmID`** `(void)`	
Parameter	
void	none
Return code	
AlarmType	Requested parameter value.
Functional Description	
Returns parameter AlarmID of a faulty GetAlarmBase call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-115 OSError_GetAlarmBase_AlarmID

5.1.125 OSError_GetAlarmBase_Info

Prototype	
`AlarmBaseRefType` **`OSError_GetAlarmBase_Info`** `(void)`	
Parameter	
void	none
Return code	
AlarmBaseRefType	Requested parameter value.
Functional Description	
Returns parameter Info of a faulty GetAlarmBase call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-116 OSError_GetAlarmBase_Info

5.1.126 OSError_GetAlarm_AlarmID

Prototype	
`AlarmType` **`OSError_GetAlarm_AlarmID`** `(void)`	
Parameter	
void	none
Return code	
AlarmType	Requested parameter value.
Functional Description	
Returns parameter AlarmID of a faulty GetAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-117 OSError_GetAlarm_AlarmID

5.1.127 OSError_GetAlarm_Tick

Prototype	
`TickRefType` **`OSError_GetAlarm_Tick`** `(void)`	
Parameter	
void	none
Return code	
TickRefType	Requested parameter value.
Functional Description	
Returns parameter Tick of a faulty GetAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-118 OSError_GetAlarm_Tick

5.1.128 OSError_SetRelAlarm_AlarmID

Prototype	
`AlarmType` **`OSError_SetRelAlarm_AlarmID`** `(void)`	
Parameter	
void	none
Return code	
AlarmType	Requested parameter value.
Functional Description	
Returns parameter AlarmID of a faulty SetRelAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-119 OSError_SetRelAlarm_AlarmID

5.1.129 OSError_SetRelAlarm_increment

Prototype	
`TickType` **`OSError_SetRelAlarm_increment`** `(void)`	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter increment of a faulty SetRelAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-120 OSError_SetRelAlarm_increment

5.1.130 OSError_SetRelAlarm_cycle

Prototype	
`TickType OSError_SetRelAlarm_cycle (void)`	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter cycle of a faulty SetRelAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-121 OSError_SetRelAlarm_cycle

5.1.131 OSError_SetAbsAlarm_AlarmID

Prototype	
`AlarmType OSError_SetAbsAlarm_AlarmID (void)`	
Parameter	
void	none
Return code	
AlarmType	Requested parameter value.
Functional Description	
Returns parameter AlarmID of a faulty SetAbsAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-122 OSError_SetAbsAlarm_AlarmID

5.1.132 OSError_SetAbsAlarm_start

Prototype	
`TickType` **`OSError_SetAbsAlarm_start`** `(void)`	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter start of a faulty SetAbsAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-123 OSError_SetAbsAlarm_start

5.1.133 OSError_SetAbsAlarm_cycle

Prototype	
`TickType` **`OSError_SetAbsAlarm_cycle`** `(void)`	
Parameter	
void	none
Return code	
TickType	Requested parameter value.
Functional Description	
Returns parameter cycle of a faulty SetAbsAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-124 OSError_SetAbsAlarm_cycle

5.1.134 OSError_CancelAlarm_AlarmID

Prototype	
`AlarmType OSError_CancelAlarm_AlarmID (void)`	
Parameter	
void	none
Return code	
AlarmType	Requested parameter value.
Functional Description	
Returns parameter AlarmID of a faulty CancelAlarm call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-125 OSError_CancelAlarm_AlarmID

5.1.135 OSError_GetResource_ResID

Prototype	
`ResourceType OSError_GetResource_ResID (void)`	
Parameter	
void	none
Return code	
ResourceType	Requested parameter value.
Functional Description	
Returns parameter ResID of a faulty GetResource call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-126 OSError_GetResource_ResID

5.1.136 OSError_ReleaseResource_ResID

Prototype	
`ResourceType **OSError_ReleaseResource_ResID** (void)`	
Parameter	
void	none
Return code	
ResourceType	Requested parameter value.
Functional Description	
Returns parameter ResID of a faulty ReleaseResource call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-127 OSError_ReleaseResource_ResID

5.1.137 OSError_Os_GetUnhandledIrq_InterruptSource

Prototype	
`Os_InterruptSourceIdRefType **OSError_Os_GetUnhandledIrq_InterruptSource** (void)`	
Parameter	
void	none
Return code	
Os_InterruptSourceIdRefType	Requested parameter value.
Functional Description	
Returns parameter InterruptSource of a faulty Os_GetUnhandledIrq call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-128 OSError_Os_GetUnhandledIrq_InterruptSource

5.1.138 OSError_Os_GetUnhandledExc_ExceptionSource

Prototype	
`Os_ExceptionSourceIdRefType` **`OSError_Os_GetUnhandledExc_ExceptionSource`** `(void)`	
Parameter	
void	none
Return code	
Os_ExceptionSourceIdRefType	Requested parameter value.
Functional Description	
Returns parameter ExceptionSource of a faulty Os_GetUnhandledExc call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-129 OSError_Os_GetUnhandledExc_ExceptionSource

5.1.139 OSError_BarrierSynchronize_BarrierID

Prototype	
`Os_BarrierIdType` **`OSError_BarrierSynchronize_BarrierID`** `(void)`	
Parameter	
`void`	none
Return code	
`Os_BarrierIdType`	Requested parameter value
Functional Description	
Returns parameter BarrierID of a faulty Os_BarrierSynchronize call.	
Particularities and Limitations	
> Pre-Condition: None	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-130 OSError_BarrierSynchronize_BarrierID

5.1.140 OSError_ActivateTaskAsyn_TaskID

Prototype	
`TaskType OSError_ActivateTaskAsyn_TaskID (void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty ActivateTaskAsyn call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-131 OSError_ActivateTaskAsyn_TaskID

5.1.141 OSError_SetEventAsyn_TaskID

Prototype	
`TaskType OSError_SetEventAsyn_TaskID (void)`	
Parameter	
void	none
Return code	
TaskType	Requested parameter value.
Functional Description	
Returns parameter TaskID of a faulty SetEventAsyn call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-132 OSError_SetEventAsyn_TaskID

5.1.142 OSError_SetEventAsyn_Mask

Prototype	
`EventMaskType **OSError_SetEventAsyn_Mask** (void)`	
Parameter	
void	none
Return code	
EventMaskType	Requested parameter value.
Functional Description	
Returns parameter Mask of a faulty SetEventAsyn call.	
Particularities and Limitations	
Pre-Condition: None --no details--	
Call context	
> ERRHOOK > This function is Synchronous > This function is Reentrant	

Table 5-133 OSError_SetEventAsyn_Mask

5.2 Additional OS services

The OS provides the following additional services which are not part of the AUTOSAR OS specification.

5.2.1 Os_GetVersionInfo

Prototype	
`void **Os_GetVersionInfo** (Std_VersionInfoType *versioninfo)`	
Parameter	
`versioninfo [out]`	Version information (decimal coded).
Return code	
`void`	none
Functional Description	
AUTOSAR Get Version Information API.	
Particularities and Limitations	
Given object pointer(s) are valid. Returns the Published information of MICROSAR Classic OS.	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-131 Os_GetVersionInfo

5.2.2 Peripheral Access API

The API consists of read, write and bit manipulating functions for 8, 16 and 32 bit accesses.

5.2.2.1 Read Functions

Prototype	
`FUNC(uint8, OS_CODE) Os_ReadPeripheral8( Os_PeripheralIdType PeripheralID, P2CONST(uint8, AUTOMATIC, OS_APPL_DATA) Address )`	
`FUNC(uint16, OS_CODE) Os_ReadPeripheral16( Os_PeripheralIdType PeripheralID, P2CONST(uint16, AUTOMATIC, OS_APPL_DATA) Address )`	
`FUNC(uint32, OS_CODE) Os_ReadPeripheral32( Os_PeripheralIdType PeripheralID, P2CONST(uint32, AUTOMATIC, OS_APPL_DATA) Address )`	
Parameter	
`PeripheralID`	The ID of a configured peripheral region. The symbolic name may be passed here.
`Address`	The address of the peripheral register which shall be read.
Return code	
`uint8`	The content of the peripheral register which has been passed in the Address parameter.
`uint16`	
`uint32`	
Functional Description	
The function distinguishes the address range of the passed peripheral region. It checks whether the parameter "Address" is within this range. Then it checks whether the calling OS application has access rights to the passed peripheral region. If all checks did pass the API returns the content of the passed address	
Particularities and Limitations	
> If one of the performed checks within the API is not passed the OS treats it as a memory protection violation. The ProtectionHook() is called. > The data alignment of the "Address" parameter is not checked by the service function. Misaligned accesses may lead to exceptions.	

Table 5-132 Read Peripheral API

Note
The former names of the API functions osReadPeripheral8(), osReadPeripheral16() and osReadPeripheral32() may also be used (the OS is backward compatible).

5.2.2.2 Write Functions

Prototype	

`FUNC(void, OS_CODE) Os_WritePeripheral8(`
`  Os_PeripheralIdType PeripheralID,`
`  P2VAR(uint8, AUTOMATIC, OS_APPL_DATA) Address,`
`  uint8 Value`
`)`	

`FUNC(void, OS_CODE) Os_WritePeripheral16(`
`  Os_PeripheralIdType PeripheralID,`
`  P2VAR(uint16, AUTOMATIC, OS_APPL_DATA) Address,`
`  uint16 Value`
`)`	

`FUNC(void, OS_CODE) Os_WritePeripheral32(`
`  Os_PeripheralIdType PeripheralID,`
`  P2VAR(uint32, AUTOMATIC, OS_APPL_DATA) Address,`
`  uint32 Value`
`)`	

Parameter	
`PeripheralID`	The ID of a configured peripheral region. The symbolic name may be passed here.
`Address`	The address of the peripheral register which shall be written.
`Value uint8`	Value which shall be written to the peripheral register.
`Value uint16`	
`Value uint32`	
Return code	
`void`	none
Functional Description	
The function distinguishes the address range of the passed peripheral region. It checks whether the parameter “Address” is within this range. Then it checks whether the calling OS application has access rights to the passed peripheral region. If all checks did pass the OS writes the Value into the peripheral register.	
Particularities and Limitations	
> If one of the performed checks within the API is not passed the OS treats it as a memory protection violation. The ProtectionHook() is called. > The data alignment of the “Address” parameter is not checked by the service function. Misaligned accesses may lead to exceptions.	

Table 5-133 Write Peripheral APIs

Note
The former names of the API functions osWritePeripheral8(), osWritePeripheral16() and osWritePeripheral32() may also be used (the OS is backward compatible).

5.2.2.3 Bitmask Functions

Prototype	
`FUNC(void, OS_CODE) Os_ModifyPeripheral8( Os_PeripheralIdType PeripheralID, P2VAR(uint8, AUTOMATIC, OS_APPL_DATA) Address, uint8 ClearMask, uint8 SetMask )`	
`FUNC(void, OS_CODE) Os_ModifyPeripheral16( Os_PeripheralIdType PeripheralID, P2VAR(uint16, AUTOMATIC, OS_APPL_DATA) Address, uint16 ClearMask, uint16 SetMask )`	
`FUNC(void, OS_CODE) Os_ModifyPeripheral32( Os_PeripheralIdType PeripheralID, P2VAR(uint32, AUTOMATIC, OS_APPL_DATA) Address, uint32 ClearMask, uint32 SetMask )`	
Parameter	
`PeripheralID`	The ID of a configured peripheral region. The symbolic name may be passed here.
`Address`	The address of the peripheral register which shall be modified.

`ClearMask uint8`
`ClearMask uint16`
`ClearMask uint32`
	The mask for the AND operation.

`SetMask uint8`
`SetMask uint16`
`SetMask uint32`
	The mask for the OR operation.

Return code	
`void`	none
Functional Description	
The function distinguishes the address range of the passed peripheral region. It checks whether the parameter “Address” is within this range. Then it checks whether the calling OS application has access rights to the passed peripheral region. If all checks did pass the OS performs the following operation: `Address = (Address & ClearMask) | SetMask;`	
Particularities and Limitations	
> If one of the performed checks within the API is not passed the OS treats it as a memory protection violation. The ProtectionHook() is called.	

> The data alignment of the “Address” parameter is not checked by the service function. Misaligned accesses may lead to exceptions.

Table 5-134 Bitmask Peripheral API

Note
The former names of the API functions osModifyPeripheral8(), osModifyPeripheral16() and osModifyPeripheral32() may also be used (the OS is backward compatible).

5.2.3 Pre-Start Task

Prototype
`FUNC(void, OS_CODE) Os_EnterPreStartTask(void)`
Parameter
none
Return code
none
Functional Description
The function schedules and dispatches to the pre-start task. The core is initialized that non-trusted function calls can be used safely within this task.
Particularities and Limitations
> Has to be called on a core which is started as an AUTOSAR core. > The core which calls this function must have a configured pre-start task. > Must only be called once. > Must be called prior to `StartOS()` but after `Os_Init()`

Table 5-135 API Service Os_EnterPreStartTask

5.2.4 Non-Trusted Functions (NTF)

Prototype	
`FUNC(StatusType, OS_CODE) Os_CallNonTrustedFunction( Os_NonTrustedFunctionIndexType FunctionIndex, Os_NonTrustedFunctionParameterRefType FunctionParams )`	
Parameter	
FunctionIndex	The Index of the non-trusted function.
FunctionParams	Pointer to parameters which are passed to the non-trusted function.
Return code	
E_OK	No error.
E_OS_SERVICEID	No function defined for this index.
E_OS_CALLEVEL	Called from invalid context. (EXTENDED status)
E_OS_ACCESS	The given object belongs to a foreign core. (EXTENDED status)
E_OS_ACCESS	Owner OS application is not accessible. (Service Protection)
E_OS_SYS_NO_NTFSTACK	No further NTF-Stacks available. (EXTENDED status)
Functional Description	
Performs a call to the non-trusted function passed in „FunctionIndex“.	
Particularities and Limitations	
> The non-trusted function will not be able to return any values. It has no access rights to the data structure of the caller referenced by the “FunctionParams” parameter. > This API service may be called with disabled interrupts.	

Table 5-136 Call Non-Trusted Function API

5.2.5 Fast Trusted Functions

Prototype	
`FUNC(StatusType, OS_CODE) Os_CallFastTrustedFunction ( Os_FastTrustedFunctionIndexType FunctionIndex, Os_FastTrustedFunctionParameterRefType FunctionParams )`	
Parameter	
FunctionIndex	Index of the function to be called.
FunctionParams	Pointer to the parameters for the function. If no parameters are provided a NULL pointer has to be passed.
Return code	
E_OK	No error.
E_OS_SERVICEID	No function defined for this index.
Functional Description	
Performs a call to the fast trusted function passed in „FunctionIndex“.	
Particularities and Limitations	
> May be called with interrupts disabled	

5.2.6 Interrupt Source API

5.2.6.1 Disable Interrupt Source

Prototype	
`FUNC(StatusType, OS_CODE) Os_DisableInterruptSource( ISRType ISRID )`	
Parameter	
`ISRID`	The ID of a category 2 ISR.
Return code	
`E_OK`	No error.
`E_OS_ID`	ISRID is not a valid category 2 ISR identifier (EXTENDED status)
`E_OS_CALLEVEL`	Wrong call context of the API function (EXTENDED status)
`E_OS_ACCESS`	The calling application is not the owner of the ISR passed in ISRID (Service Protection)
Functional Description	
MICROSAR Classic OS disables the interrupt source by modifying the interrupt controller registers.	
Particularities and Limitations	
> May be called for category 2 ISRs only.	

Table 5-137 API Service Os_DisableInterruptSource

Caution

Depending on target platform (e.g. ARM platforms), the ISR may still become active although Os_DisableInterruptSource has returned E_OK.

This may be caused by hardware racing conditions e.g. when the interrupt is requested immediately before the effect of Os_DisableInterruptSource becomes active.

5.2.6.2 Enable Interrupt Source

Prototype	
`FUNC(StatusType, OS_CODE) Os_EnableInterruptSource( ISRType ISRID, boolean ClearPending )`	
Parameter	
ISRID	The ID of a category 2 ISR.
ClearPending	Defines whether the pending flag shall be cleared (TRUE) or not (FALSE).
Return code	
E_OK	No error.
E_OS_ID	ISRID is not a valid category 2 ISR identifier ID (EXTENDED status)
E_OS_CALLEVEL	Wrong call context of the API function (EXTENDED status)
E_OS_VALUE	The parameter “ClearPending” is not a boolean value (EXTENDED status)
E_OS_ACCESS	The calling application is not the owner of the ISR passed in ISRID (Service Protection)
E_OS_SYS_UNIMPLEMENTED_FUNCTIONALITY	Hardware does not support to clear pending interrupts (EXTENDED status)
Functional Description	
MICROSAR Classic OS enables the interrupt source by modifying the interrupt controller registers. Additionally it may clear the interrupt pending flag	
Particularities and Limitations	
> May be called for category 2 ISRs only	

Table 5-138 API Service Os_EnableInterruptSource

5.2.6.3 Clear Pending Interrupt

Prototype	
`FUNC(StatusType, OS_CODE) Os_ClearPendingInterrupt( ISRType ISRID )`	
Parameter	
`ISRID`	The ID of a category 2 ISR.
Return code	
`E_OK`	No errors
`E_OS_ID`	ISRID is not a valid category 2 ISR identifier (EXTENDED status)
`E_OS_CALLEVEL`	Wrong call context of the API function (EXTENDED status)
`E_OS_ACCESS`	The calling application is not the owner of the ISR passed in ISRID (Service Protection)
`E_OS_SYS_UNIMPLEMENTED_FUNCTIONALITY`	Hardware does not support to clear pending interrupts (EXTENDED status)
Functional Description	
MICROSAR Classic OS clears the interrupt pending flag by modifying the interrupt controller registers.	
Particularities and Limitations	
> May be called for category 2 ISRs only	

Table 5-139 API Service Os_ClearPendingInterrupt

Note
In order to minimize the risk of spurious interrupts, Os_ClearPendingInterrupt shall be called only after the ISR (IsrId) has been disabled and before it is enabled again.

Note
The API service tries to clear the pending flag only. The interrupt cause has to be reset by the application software. Otherwise the flag may be set again immediately after it has been cleared by the API. This may be the case e.g. with level triggered ISRs.

5.2.6.4 Check Interrupt Source Enabled

Prototype	
`FUNC(StatusType, OS_CODE) Os_IsInterruptSourceEnabled( ISRType ISRID, P2VAR(boolean, AUTOMATIC, OS_VAR_NOINIT) IsEnabled )`	
Parameter	
`ISRID`	The ID of a category 2 ISR.
`IsEnabled`	Defines wether the source of the ISR is enabled (TRUE) or not (FALSE)
Return code	
`E_OK`	No errors
`E_OS_ID`	ISRID is not a valid category 2 ISR identifier (EXTENDED status)
`E_OS_CALLEVEL`	Wrong call context of the API function (EXTENDED status)
`E_OS_ACCESS`	The calling application is not the owner of the ISR passed in ISRID (Service Protection)
`E_OS_PARAM_POINTER`	Given pointer parameter (isEnabled) is NULL (EXTENDED status)
Functional Description	
MICROSAR Classic OS checks if the interrupt source is enabled reading the interrupt controller registers and update the boolean addressed by IsEnabled accordingly	
Particularities and Limitations	
> May be called for category 2 ISRs only	

Table 5-140 API Service Os_IsInterruptSourceEnabled

5.2.6.5 Check Interrupt Pending

Prototype	
`FUNC(StatusType, OS_CODE) Os_IsInterruptPending( ISRType ISRID, P2VAR(boolean, AUTOMATIC, OS_VAR_NOINIT) IsPending )`	
Parameter	
`ISRID`	The ID of a category 2 ISR.
`IsPending`	Defines wether the ISR has been already requesterd (TRUE) or not (FALSE)
Return code	
`E_OK`	No errors
`E_OS_ID`	ISRID is not a valid category 2 ISR identifier (EXTENDED status)
`E_OS_CALLEVEL`	Wrong call context of the API function (EXTENDED status)
`E_OS_ACCESS`	The calling application is not the owner of the ISR passed in ISRID (Service Protection)
`E_OS_PARAM_POINTER`	Given pointer parameter (isPending) is NULL (EXTENDED status)
`E_OS_SYS_UNIMPLEMENTED_FUNCTIONALITY`	Hardware does not support to check if there are pending interrupts
Functional Description	
MICROSAR Classic OS checks if the ISR has been already requested, reading the interrupt controller registers and update the boolean addressed by IsPending accordingly	
Particularities and Limitations	
> May be called for category 2 ISRs only	

Table 5-141 API Service Os_IsInterruptPending

5.2.6.6 Initial Enable Interrupt Sources

Prototype	
`FUNC(StatusType, OS_CODE) Os_InitialEnableInterruptSources( boolean ClearPending )`	
Parameter	
`ClearPending`	Defines whether the pending flag shall be cleared (TRUE) or not (FALSE).
Return code	
`E_OK`	No error.
`E_OS_CALLEVEL`	Wrong call context of the API function (EXTENDED status)
`E_OS_VALUE`	The parameter “ClearPending” is not a boolean value (EXTENDED status)
`E_OS_SYS_UNIMPLEMENTED_FUNCTIONALITY`	Hardware does not support to clear pending interrupts (EXTENDED status)
Functional Description	
MICROSAR Classic OS enables the interrupt sources of all category 2 ISRs by modifying the interrupt controller registers. Additionally it may clear the interrupt pending flags.	
Particularities and Limitations	
> API function can only be called in the context of a trusted and privileged task.	

Table 5-142 API Service Os_InitialEnableInterruptSources

5.2.7 Detailed Error API

5.2.7.1 Get detailed Error

Prototype	
`FUNC(StatusType, OS_CODE) Os_GetDetailedError( Os_ErrorInformationRefType ErrorRef )`	
Parameter	
`ErrorRef`	Output parameter of type Os_ErrorInformationRefType
Return code	
`E_OK`	No error.
`E_OS_CALLEVEL`	Called from invalid context. (EXTENDED status)
`E_OS_PARAM_POINTER`	Given parameter pointer is NULL. (EXTENDED status)
Functional Description	
Returns error information of the last error occurred on the local core.	
Particularities and Limitations	
> The ErrorRef output parameter is a struct which holds the 8 bit AUTOSAR error code, the detailed error code and the service ID of the causing API service.	

Table 5-143 API Service Os_GetDetailedError

5.2.7.2 Unhandled Interrupt Requests

Prototype	

`FUNC(StatusType, OS_CODE) Os_GetUnhandledIrq(`
`  Os_InterruptSourceIdRefType InterruptSource`
`)`	

Parameter	
InterruptSource	Output parameter of type Os_InterruptSourceIdRefType
Return code	
E_OK	No error.
E_OS_CORE	Called from a non-AUTOSAR core (EXTENDED status)
E_OS_PARAM_POINTER	Null pointer passed as argument (EXTENDED status)
E_OS_STATE	No unhandled interrupt reported since start up (EXTENDED status)
Functional Description	
In case of an unhandled interrupt request the triggering interrupt source can be distinguished with this service.	
Particularities and Limitations	
> The return value of this function may be interpreted differently for different controller families. Please refer to [9] for additional details.	

Table 5-144 API Service Os_GetUnhandledIrq

5.2.7.3 Unhandled Exception Requests

Prototype	
`FUNC(StatusType, OS_CODE) Os_GetUnhandledExc( Os_ExceptionSourceIdRefType ExceptionSource )`	
Parameter	
`ExceptionSource`	Output parameter of type Os_ExceptionSourceIdRefType
Return code	
`E_OK`	No error.
`E_OS_CORE`	Called from a non-AUTOSAR core (EXTENDED status)
`E_OS_PARAM_POINTER`	Null pointer passed as argument (EXTENDED status)
`E_OS_STATE`	No unhandled exception reported since start up. (EXTENDED status)
Functional Description	
In case of an unhandled exception request the triggering exception source can be distinguished with this service.	
Particularities and Limitations	
> The return value of this function may be interpreted differently for different controller families. Please refer to [9] for additional details.	

Table 5-145 API Service Os_GetUnhandledExc

5.2.7.4 Get Exception Address

Prototype	
`FUNC(Os_AddressOfConstType, OS_CODE) Os_GetExceptionAddress( void )`	
Parameter	
`void`	none
Return code	
`Os_AddressOfConstType`	Address of the instruction that raised the latest exception.
Functional Description	
Gets the address of the instruction that raised the latest exception. The returned address is only valid if at least one exception has occurred.	
Particularities and Limitations	
> This function will never fail. On platforms that cannot provide the exception address, the return value will always be invalid.	

Table 5-153 API Service Os_GetExceptionAddress

5.2.8 Stack Usage API

All Service API functions which calculate stack usage are working in the same way.

- The service performs error checks:
  - stack usage measurement (see 2.3.6) enabled
  - validity of passed parameters
  - existence of OS Hook routine (if hook stacks are queried)
  - cross core checks (when stack sizes are queried of stacks which are located on a foreign core)
  - if one of these checks fails, the OS initiates error handling (ErrorHook() is called)
- Calculates the maximum stack usage of the queried stack since call of StartOS()
  - For non-trusted function stacks, the highest consumption of all stacks (from stack pool) for the function is calculated
- Returns the stack usage in bytes or zero in case of any error
- Stack Usage API services may be called from any context
- Stack Usage API services may be used cross core

Stack usage service API Prototypes	Parameter
`FUNC(uint32, OS_CODE) Os_GetTaskStackUsage (TaskType TaskID)`	Task ID
`FUNC(uint32, OS_CODE) Os_GetISRStackUsage (ISRType IsrID)`	ISR ID
`FUNC(uint32, OS_CODE) Os_GetKernelStackUsage (CoreIdType CoreID)`	Core ID
`FUNC(uint32, OS_CODE) Os_GetStartupHookStackUsage(CoreIdType CoreID)`	Core ID
`FUNC(uint32, OS_CODE) Os_GetErrorHookStackUsage (CoreIdType CoreID)`	Core ID
`FUNC(uint32, OS_CODE) Os_GetShutdownHookStackUsage(CoreIdType CoreID)`	Core ID
`FUNC(uint32, OS_CODE) Os_GetProtectionHookStackUsage(CoreIdType CoreID)`	Core ID
`FUNC(uint32, OS_CODE) Os_GetInitHookStackUsage(CoreIdType CoreID);`	Core ID
`FUNC(uint32, OS_CODE) Os_GetNonTrustedFunctionStackUsage (Os_NonTrustedFunctionIndexType FunctionIndex);`	Non-trusted function ID

Table 5-146 Overview: Stack Usage Functions

Caution
Any stack usage function must not be used cross core with interrupts disabled.

5.2.9 RTE Interrupt API

MICROSAR Classic OS provides optimized interrupt en-/disable functions for exclusive usage by the RTE module of Vector.

API Name	Alias (for backward compatibility)	Comment
Os_DisableLevelAM()	osDisableLevelAM()	non nestable service to disable all category 2 interrupts callable from any mode
Os_DisableLevelKM()	osDisableLevelKM()	non nestable service to disable all category 2 interrupts callable from kernel mode
Os_DisableLevelUM()	osDisableLevelUM()	non nestable service to disable all category 2 interrupts callable from user mode
Os_EnableLevelAM()	osEnableLevelAM()	non nestable service to enable all category 2 interrupts callable from any mode
Os_EnableLevelKM()	osEnableLevelKM()	non nestable service to enable all category 2 interrupts callable from kernel mode
Os_EnableLevelUM()	osEnableLevelUM()	non nestable service to enable all category 2 interrupts callable from user mode
Os_DisableGlobalAM()	osDisableGlobalAM()	non nestable service to disable all interrupts callable from any mode
Os_DisableGlobalKM()	osDisableGlobalKM()	non nestable service to disable all interrupts callable from kernel mode
Os_DisableGlobalUM()	osDisableGlobalUM()	non nestable service to disable all interrupts callable from user mode
Os_EnableGlobalAM()	osEnableGlobalAM()	non nestable service to enable all interrupts callable from any mode
Os_EnableGlobalKM()	osEnableGlobalKM()	non nestable service to enable all interrupts callable from kernel mode
Os_EnableGlobalUM()	osEnableGlobalUM()	non nestable service to enable all interrupts callable from user mode

Caution
RTE interrupt handling functions should not be used by the application and are listed here to avoid naming collisions.

Caution
When nesting other OS interrupt locking/unlocking APIs and RTE interrupt APIs erroneous behavior is possible. One error that may be reported by the OS in this case is “OS_STATUS_DISABLEDINT”.

5.2.10 Time Conversion Macros

Based on counter configuration attributes conversion macros are generated which are capable to convert from time into counter ticks and vice versa.

There are a set of conversion macros for each configured OS counter

Caution
The conversion macros embody multiplication operations which may lead to a data type overflow. The macros are not capable to detect these overflows

Caution
Although the results of the macros are mathematically rounded the result will still be an integer (e.g. results smaller than 0.5 are used as 0).

5.2.10.1 Convert from Time into Counter Ticks

OS_NS2TICKS_<Counter Name>(x)	x is given in nanoseconds
OS_US2TICKS_<Counter Name>(x)	x is given in microseconds
OS_MS2TICKS_<Counter Name>(x)	x is given in milliseconds
OS_SEC2TICKS_<Counter Name>(x)	x is given in seconds

Table 5-147 Conversion Macros from Time to Counter Ticks

5.2.10.2 Convert from Counter Ticks into Time

OS_TICKS2NS_<Counter Name>(x)	The result is in nanoseconds
OS_TICKS2US_<Counter Name>(x)	The result is in microseconds
OS_TICKS2MS_<Counter Name>(x)	The result is in milliseconds
OS_TICKS2SEC_<Counter Name>(x)	The result is in seconds

Table 5-148 Conversion Macros from Counter Ticks to Time

5.2.11 OS Initialization

Prototype
`FUNC(void, OS_CODE) Os_InitInterruptOnly(void)`
Parameter
none
Return code
none
Functional Description
The function is only available if INTERRUPT_ONLY use case is selected (OsUseCase). The function performs the basic initialization to handle category 1 interrupts. This includes: > Core Interrupt Controller initialization > System Interrupt Controller initialization
Particularities and Limitations

> The function is only usable in single core OS configurations (only one AUTOSAR core).
> The function does not enable global interrupt handling by means of:
    > Altering global interrupt flags in core registers.
    > Altering interrupt level registers in the interrupt controller.
> The functions does not enable the interrupt source.
> After call of `Os_InitInterruptOnly()` the following AUTOSAR interrupt API may be used.
    > DisableAllInterrupts
    > EnableAllInterrupts
    > Os_EnableInterruptsPreStart

Table 5-149 API Service Os_InitInterruptOnly

Prototype
`FUNC(void, OS_CODE) Os_Init(void)`
Parameter
none
Return code
none
Functional Description
The function performs all the basic OS initialization which includes > Variable initialization > Interrupt controller initialization > System MPU initialization in SC3 and SC4 systems (if supported by platform) > Synchronization barriers in multi core systems
Particularities and Limitations

> A function call to this service must be available on all available cores (even for cores which are intended to be a non-AUTOSAR core)
> After call of `Os_Init()` the AUTOSAR interrupt API may be used.
> After Call of `Os_Init()` the API GetCoreID may be used.
> Pre-Condition:
    > Os_Init may only be called if the interrupts are globally disabled.
    > Either disable the interrupts by using the global flag or, in case of Cortex M platform, disable the interrupts by setting the highest possible interrupt level (BASEPRI register).

Table 5-150 API Service Os_Init

Prototype
`FUNC(void, OS_CODE) Os_InitMemory(void)`
Parameter
none
Return code
none
Functional Description
> This is an API function which is provided within all BSWs of Vector. It initializes variables of the BSW. Within the OS module this function is currently empty
Particularities and Limitations
> This service must be called on all available cores (even for cores which are intended to be a non-AUTOSAR core)

Table 5-151 API Service Os_InitMemory

5.2.12 Timing Hooks

Implementation of all timing hooks must conform to the following guidelines:

> They are expected to be implemented as a macro.

> Reentrancy is possible on multicore systems with different caller core IDs.

> Calls of any operating system API functions are prohibited within the hooks.

Note
All hooks are called from within an OS API service. Interrupts are disabled

5.2.12.1 Timing Hooks for Activation and Termination

5.2.12.1.1 Task Activation

Macro	
`#define OS_VTH_ACTIVATION(TaskId, DestCoreId, CallerCoreId)`	
Parameter	
`TaskId`	Identifier of the task which is activated
`DestCoreId`	Identifier of the core on which the task is activated
`CallerCoreId`	Identifier of the core which performs the activation (has called ActivateTask(), has called ChainTask() or has performed an alarm/schedule table action to activate a task)
Return code	
none	
Functional Description	
This hook is called on the caller core when that core has successfully performed the activation of TaskId on the destination core. On single core systems both core IDs are identical.	
Particularities and Limitations	
> Due to internal implementation `DestCoreId` and `CallerCoreId` are always the same.	

5.2.12.1.2 Task Activation Exeeding Limit

Macro	
`#define OS_VTH_ACTIVATION_LIMIT(TaskId, DestCoreId, CallerCoreId)`	
Parameter	
`TaskId`	Identifier of the task which is activated
`DestCoreId`	Identifier of the core on which the task is activated
`CallerCoreId`	Identifier of the core which performs the activation (has called ActivateTask(), has called ChainTask() or has performed an alarm/schedule table action to activate a task)
Return code	
none	
Functional Description	
This hook is called on the caller core when that core has failed the activation of TaskId on the destination core because number of activations exceeds the limit.	
Particularities and Limitations	
> Due to internal implementation `DestCoreId` and `CallerCoreId` are always the same.	

5.2.12.1.3 Set Event

Macro	
`#define OS_VTH_SETEVENT(TaskId, EventMask, StateChanged, DestCoreId, CallerCoreId)`	
Parameter	
`TaskId`	Identifier of the task which receives this event
`EventMask`	A bit mask with the events which shall be set
`StateChanged`	TRUE: The task state has changed from WAITING to READY FALSE: The task state hasn't changed
`DestCoreId`	Identifier of the core on which the task receives the event
`CallerCoreId`	Identifier of the core which performs the event setting (has called SetEvent() or performed an alarm/schedule table action to set an event)
Return code	
none	
Functional Description	
This hook is called on the caller core when that core has successfully performed the event setting on the destination core.	
Particularities and Limitations	
> Due to internal implementation `DestCoreId` and `CallerCoreId` are always the same.	

5.2.12.1.4 Wait Event Not Waiting

Macro	
`#define OS_VTH_WAITEVENT_NOWAIT(TaskId, EventMask, DestCoreId, CallerCoreId)`	
Parameter	
`TaskId`	Identifier of the task which is waiting for the event
`EventMask`	A bit mask with the events for which the task is waiting
`DestCoreId`	Identifier of the core on which the task is waiting for the event
`CallerCoreId`	Identifier of the core which performs the wait event (has called WaitEvent())
Return code	
none	
Functional Description	
This hook is called on the caller core when that core has successfully performed the wait event call on the destination core and the events waiting are already set and calling task stays in state RUNNING.	
Particularities and Limitations	
> Due to internal implementation `DestCoreId` and `CallerCoreId` are always the same.	

5.2.12.1.5 Timing Hook for Context Switch

Macro	
`#define OS_VTH_SCHEDULE(FromThreadId, FromThreadReason, ToThreadId, ToThreadReason, CallerCoreId)`	
Parameter	
`FromThreadId`	Identifier of the thread (task, ISR) which has run on the caller core before the switch took place
`FromThreadReason`	The reason, why thread "FromThreadId" is no longer running: OS_VTHP_TASK_TERMINATION > The thread is a task, which has just been terminated. OS_VTHP_ISR_END > The thread is an ISR, which has reached its end. OS_VTHP_TASK_WAITEVENT > The thread is a task, which waits for an event. OS_VTHP_TASK_WAITSEMA > The thread is a task, which waits for the release of a semaphore. OS_VTHP_THREAD_PREEMPT > The thread is interrupted by another one, which has higher priority.
`ToThreadId`	The identifier of the thread, which runs from now on
`ToThreadReason`	The reason, why thread "ToThreadId" becomes running: OS_VTHP_TASK_ACTIVATION > The thread is a task, which was activated. OS_VTHP_ISR_START > The thread is an ISR, which now starts execution. OS_VTHP_TASK_SETEVENT > The thread is a task, which has just received an event it was waiting for. It resumes execution right behind the call of WaitEvent(). OS_VTHP_TASK_GOTSEMA > The thread is a task, which has just got the semaphore it was waiting for. OS_VTHP_THREAD_RESUME: > The thread is a task or ISR, which was preempted before and becomes running again as all higher priority tasks and ISRs do not run anymore. OS_VTHP_THREAD_CLEANUP: > The thread is an ISR which has been forcibly terminated. The implementation of the ISR will not be entered but just some OS internal cleanup code which is needed to switch to the next thread.
`CallerCoreId`	Identifier of the core which performs the thread switch
Return code	
none	

Functional Description
This hook is called on a core when it performs a thread switch (from one task or ISR to another task or ISR).
Particularities and Limitations
> None

5.2.12.1.6 Forcible Termination

Macro	
`#define OS_VTH_FORCED_TERMINATION(ThreadId, CallerCoreId)`	
Parameter	
`ThreadId`	Identifier of the thread (task or ISR) which has been forcibly terminated
`CallerCoreId`	Identifier of the core which performs forcible termination
Return code	
none	
Functional Description	
This hook is called in case a thread (task or ISR) has been forcibly terminated. The thread may not have finished its computations as some error detection mechanism has decided before to forcibly terminate the thread.	
Particularities and Limitations	
> none	

5.2.12.2 Timing Hooks for Locking Purposes

5.2.12.2.1 Get Resource

Macro	
`#define OS_VTH_GOT_RES(ResId, CallerCoreId)`	
Parameter	
`ResId`	Identifier of the resource which has been taken
`CallerCoreId`	Identifier of the core where GetResorce() was called
Return code	
none	
Functional Description	
The OS calls this hook on a successful call of the API function GetResource(). The priority of the calling task or ISR has been increased so that other tasks and ISRs on the same core may need to wait until they can be executed.	

Particularities and Limitations
> none

5.2.12.2.2 Release Resource

Macro	
`#define OS_VTH_REL_RES(ResId, CallerCoreId)`	
Parameter	
`ResId`	Identifier of the resource which has been released
`CallerCoreId`	Identifier of the core where ReleaseResorce() was called
Return code	
None	
Functional Description	
The OS calls this hook on a successful call of the API function ReleaseResource(). The priority of the calling task or ISR has been decreased so that other tasks and ISRs on the same core may become running as a result.	
Particularities and Limitations	
> none	

5.2.12.2.3 Request Spinlock

Macro	
`#define OS_VTH_REQ_SPINLOCK(SpinlockId, CallerCoreId)`	
Parameter	
`SpinlockId`	Identifier of the spinlock which has been requested
`CallerCoreId`	Identifier of the core where GetSpinlock() was called
Return code	
none	
Functional Description	
The OS calls this hook on any attempt to get a spinlock. The calling task or ISR may end up in entering a busy waiting loop. In such case other tasks or ISRs of lower priority have to wait until this task or ISR has taken and released the spinlock.	
Particularities and Limitations	
> The hook is not called for optimized spinlocks > The hook is called only on multicore operating system implementations	

5.2.12.2.4 Request Internal Spinlock

Macro	
`#define OS_VTH_REQ_ISPINLOCK(SpinlockId, CallerCoreId)`	
Parameter	
`SpinlockId`	Identifier of the spinlock which has been requested
`CallerCoreId`	Identifier of the core where the internal spinlock was requested
Return code	
none	
Functional Description	
The OS calls this hook on any attempt to get a spinlock for the OS itself. The OS may end up in entering a busy waiting loop. In such case other program parts on this core have to wait until the OS has taken and released the spinlock.	
Particularities and Limitations	
> Only called for Spinlocks which used internally by the OS	

5.2.12.2.5 Get Spinlock

Macro	
`#define OS_VTH_GOT_SPINLOCK(SpinlockId, CallerCoreId)`	
Parameter	
`SpinlockId`	Identifier of the spinlock which has been taken
`CallerCoreId`	Identifier of the core where GetSpinlock() or TryToGetSpinlock() were called
Return code	
none	
Functional Description	
The OS calls this hook whenever a spinlock has successfully been taken. If a previosly attempt of getting the spinlock was not successful immediately (entered busy waiting loop), this hook means that the core leaves the busy waiting loop. From now on no other thread may get the spinlock until the current task or ISR has released it.	
Particularities and Limitations	
> The hook is not called for optimized spinlocks > The hook is called only on multicore operating system implementations	

5.2.12.2.6 Get Internal Spinlock

Macro	
`#define OS_VTH_GOT_ISPINLOCK(SpinlockId, CallerCoreId)`	
Parameter	
`SpinlockId`	Identifier of the spinlock which has been taken
`CallerCoreId`	Identifier of the core where the internal spinlock has been taken
Return code	
None	
Functional Description	
The OS calls this hook whenever a spinlock has successfully been taken by the OS itself. If a previosly attempt of getting the spinlock was not successful immediately (entered busy waiting loop), this hook means that the core leaves the busy waiting loop. From now on no other thread may get the spinlock until the OS has released it.	
Particularities and Limitations	
> Only called for Spinlocks which used internally by the OS	

5.2.12.2.7 Release Spinlock

Macro	
`#define OS_VTH_REL_SPINLOCK(SpinlockId, CallerCoreId)`	
Parameter	
`SpinlockId`	Identifier of the spinlock which has been released
`CallerCoreId`	Identifier of the core where ReleaseSpinlock() was called
Return code	
none	
Functional Description	
The OS calls this hook on a release of a spinlock. Other tasks and ISR may take the spinlock now.	
Particularities and Limitations	
> The hook is not called for optimized spinlocks > The hook is called only on multicore operating system implementations	

5.2.12.2.8 Release Internal Spinlock

Macro	
`#define OS_VTH_REL_ISPINLOCK(SpinlockId, CallerCoreId)`	
Parameter	
`SpinlockId`	Identifier of the spinlock which has been released
`CallerCoreId`	Identifier of the core where the internal spinlock has been released
Return code	
none	
Functional Description	
The OS calls this hook on a release of a spinlock. Other tasks and ISR may take the spinlock now.	
Particularities and Limitations	
> Only called for Spinlocks which used internally by the OS	

5.2.12.2.9 Disable Interrupts

Macro	
`#define OS_VTH_DISABLEDINT(IntLockId, CallerCoreId)`	
Parameter	
`IntLockId`	OS_VTHP_CAT2INTERRUPTS: Interrupts have been disabled by means of the current interrupt level. That interrupt level has been changed in order to disable all category 2 interrupts, which also prevents task switch and alarm/schedule table management. OS_VTHP_ALLINTERRUPTS: Interrupts have been disabled by means of the global interrupt enable/disable flag. Additionally to the effects described above, also category 1 interrupts are disabled.
`CallerCoreId`	Identifier of the core where interrupts are disabled
Return code	
none	
Functional Description	
The OS calls this hook if the application has called an API function to disable interrupts. The parameter IntLockId describes whether category 1 interrupts may still occur. Mind that the two types of interrupt locking (as described by the IntLockId) are independent from each other so that the hook may be called twice before the hook OS_VTH_ENABLEDINT is called, dependent on the application.	
Particularities and Limitations	
> The hook is not called for operating system internal interrupt locks	

5.2.12.2.10 Enable Interrupts

Macro	
`#define OS_VTH_ENABLEDINT(IntLockId, CallerCoreId)`	
Parameter	
`IntLockId`	OS_VTHP_CAT2INTERRUPTS > Interrupts had been disabled by means of the current interrupt level until this hook was called. The OS releases this lock right after the hook has returned. OS_VTHP_ALLINTERRUPTS > Interrupts had been disabled by means of the global interrupt enable/disable flag before this hook was called. The OS releases this lock right after the hook has returned.
`CallerCoreId`	Identifier of the core where interrupts are disabled
Return code	
None	
Functional Description	
The OS calls this hook if the application has called an API function to enable interrupts. Mind that the two types of interrupt locking (as described by the IntLockId) are independent from each other so that interrupts may still be disabled by means of the other locking type after this hook has returned.	
Particularities and Limitations	
> The hook is not called for operating system internal interrupt locks	

5.2.13 PanicHook

Prototype	
`FUNC(void, OS_PANICHOOK_CODE) Os_PanicHook(Os_PanicStatusType Status)`	
Parameter	
`Status`	The reason why the panic hook was called.
Return code	
none	
Functional Description	
Called upon kernel panic mode.	
Particularities and Limitations	
> Trusted access rights > Interrupts are disabled > No OS API service calls are allowed	

Reference
All status codes are listed in the OS header file `Os_Types.h` and may be looked up in the enum data type `Os_PanicStatusType`.

Caution
The provided status code is only for debugging purpose. As the OS may be in an inconsistent state, the status code is not reliable.

5.2.14 Barriers

Prototype	

`FUNC(StatusType, OS_CODE) Os_BarrierSynchronize(`
`  Os_BarrierIdType BarrierID`
`)`	

Parameter	
BarrierID	The barrier to which the task shall be synchronized.
Return code	
E_OK	No error
E_OS_ID	Invalid BarrierID (EXTENDED status)
E_OS_CALLEVEL	Called from invalid context (EXTENDED status)
E_OS_SYS_NO_BARRIER_PARTICIPANT	> The given barrier is not configured for the local core (EXTENDED status) > Task is not configured to participate in the barrier (EXTENDED status)
Functional Description	
Synchronize the calling task at the barrier given in "BarrierID". The calling task blocks until all other participating tasks have called this API method with the same "BarrierID".	
Particularities and Limitations	
> none	
Call context	
> Task	

Table 5-152 Barriers

5.2.15 Exception Context Manipulation

5.2.15.1 Os_GetExceptionContext

Prototype	

`FUNC(StatusType, OS_CODE) Os_GetExceptionContext(`
`  Os_ExceptionContextRefType Context`
`)`	

Parameter	
Context	Current exception context.
Return code	
E_OK	No error
E_OS_PARAM_POINTER	given pointer is a NULL_PTR (EXTENDED status)
E_OS_CALLEVEL	Called from invalid context (EXTENDED status)
E_OS_SYS_UNIMPLEMENTED_FUNCTIONALITY	Context manipulation is not supported on this hardware (EXTENDED status)
Functional Description	
Getter function for the exception context. Returns the context structure of the thread interrupted by an exception.	
Particularities and Limitations	
> none	
Call context	
> ProtectionHook	

Table 5-153 Os_GetExceptionContext

5.2.15.2 Os_SetExceptionContext

Prototype	
`FUNC(StatusType, OS_CODE) Os_SetExceptionContext(` `Os_ExceptionContextRefType Context` `)`	
Parameter	
Context	Context to set.
Return code	
E_OK	No error
E_OS_PARAM_POINTER	given pointer is a NULL_PTR (EXTENDED status)
E_OS_CALLEVEL	Called from invalid context (EXTENDED status)
E_OS_SYS_UNIMPLEMENTED_FUNCTIONALITY	Context manipulation is not supported on this hardware (EXTENDED status)
Functional Description	
Setter function for the exception context. Writes the given context into the exception context structure.	
Particularities and Limitations	
> none	
Call context	
> ProtectionHook	

Table 5-154 Os_SetExceptionContext

5.2.16 Os_GetCoreStartState

Prototype	
`FUNC(void, OS_CODE) Os_GetCoreStartState( CoreIdType CoreID, Os_CoreStartStateType *CoreState, StatusType *Status );`	
Parameter	
CoreID [in]	The core which shall be queried.
CoreState [out]	Core state.
Status [out]	Status code.
Return code	
Status	> E_OK No Error. > E_OS_PARAM_POINTER Given pointer is NULL (EXTENDED status) > E_OS_ID Invalid CoreID (EXTENDED status)
CoreState	> OS_CORESTARTSTATE_START_UNREQUESTED The start of the core has not been requested. > OS_CORESTARTSTATE_START_REQUESTED_ASR The start of the AUTOSAR core has been requested. > OS_CORESTARTSTATE_START_REQUESTED_NONASR The start of the non-AUTOSAR core has been requested. > OS_CORESTARTSTATE_STARTED_ASR The AUTOSAR core has been started. > OS_CORESTARTSTATE_STARTED_NONASR The non-AUTOSAR master core has been started.
Functional Description	
This service returns the current start state of a given core. This service supports AUTOSAR as well as non-AUTOSAR cores. This API is allowed to be used from AUTOSAR cores.	
Particularities and Limitations	
> Has to be called on a core which is started as an AUTOSAR core. > Can be called before StartOs() or StartCore(). > Must be called after Os_Init(). The state OS_CORESTARTSTATE_STARTED_NONASR can only be returned if the queried non-AUTOSAR core has called Os_Init(). For all other non-AUTOSAR cores only the start request can be evaluated.	
Call context	
> - ANY > - This function is Synchronous > - This function is Reentrant	

Table 5-155 Os_GetCoreStartState

5.2.17 OS_EnableInterruptsPreStart

Prototype

`FUNC(void, OS_CODE) Os_EnableInterruptsPreStart(`
`  void`
`);`

Parameter
none
Return code
none
Functional Description
This service enables interrupt handling on the calling core before StartOS, for example, by setting the global interrupt flag or by adjusting the interrupt level.
Particularities and Limitations
> Pre-Condition: Os_Init() or Os_InitInterruptOnly() has been called on the current core. > Pre-Condition: StartOS() has not been called on the current core.
Call context
> - This service is only available prior to StartOS() but after Os_Init() or Os_InitInterruptOnly() > - This service is only available on AUTOSAR cores > - This function is Synchronous > - This function is not Reentrant

Table 5-156 Os_EnableInterruptsPreStart

5.2.18 ActivateTaskAsyn

Prototype	
`void ActivateTaskAsyn (TaskType TaskID)`	
Parameter	
TaskID [in]	The task which shall be activated.
Return code	
void	> E_OK No error. > E_OS_LIMIT Too many task activations. > E_OS_ID (EXTENDED status:) Invalid TaskID or Asynchronous Local Task activation. > E_OS_CALLEVEL (EXTENDED status:) Called from invalid context. > E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence. > E_OS_ACCESS (Service Protection:) Caller's access rights are not sufficient or given task's owner application is not accessible.
Functional Description	
OS service ActivateTaskAsyn().	
Particularities and Limitations	
Pre-Condition: None The task TaskID is transferred from the SUSPENDED state into the READY state. The operating system ensures that the task code is being executed from the first statement.	
Call context	
> TASK|ISR2 > This function is Asynchronous > This function is Reentrant	

Table 5-160 ActivateTaskAsyn

5.2.19 SetEventAsyn

<table>
<tr><th colspan="2">Prototype</th></tr>
<tr><td colspan="2"><code>void SetEventAsyn (TaskType TaskID, EventMaskType Mask)</code></td></tr>
<tr><th colspan="2">Parameter</th></tr>
<tr><td>TaskID [in]</td><td>The task which shall be modified.</td></tr>
<tr><td>Mask [in]</td><td>The events which shall be set.</td></tr>
<tr><th colspan="2">Return code</th></tr>
<tr><td>void</td><td>> E_OK No error.
> E_OS_ID (EXTENDED status) Invalid TaskID or Asynchronous Local Task Event setting.
> E_OS_ACCESS (EXTENDED status). Task is no extended task
> E_OS_ACCESS (Service Protection). Task's owner application is not accessible. Caller's access rights are not sufficient.
> E_OS_STATE (EXTENDED status:) Events cannot be set as the referenced task is in the SUSPENDED state.
> E_OS_CALLEVEL (EXTENDED status:) Called from invalid context.
> E_OS_DISABLEDINT (EXTENDED status:) Caller is in interrupt API sequence.
> E_OS_SYS_DISABLED (EXTENDED status:) Events are not enabled in the configuration.</td></tr>
<tr><th colspan="2">Functional Description</th></tr>
<tr><td colspan="2">OS service SetEventAsyn().</td></tr>
<tr><th colspan="2">Particularities and Limitations</th></tr>
<tr><td colspan="2">Pre-Condition: None
The events of the given task are set according to the given event mask.</td></tr>
<tr><th colspan="2">Call context</th></tr>
<tr><td colspan="2">> TASK|ISR2
> This function is Asynchronous.
> This function is Reentrant</td></tr>
</table>

Table 5-161 SetEventAsyn

The following table gives an overview about the valid context for MICROSAR Classic OS additional API service calls.

Calling Context / API Service	Task	Category 1 ISR	Category 2 ISR	Error Hook	PreTask Hook	PostTask Hook	Startup Hook	Shutdown Hook	Alarm Callback	Protection Hook	Before Start of OS	Pre-Start Task	IOC callbacks
Peripheral Access APIs	X		X	X			X	X				X	
Generated IOC APIs	X		X										
Os_EnterPreStartTask											X		
Os_CallNonTrustedFunction	X		X									X	
Os_DisableInterruptSource	X		X										
Os_EnableInterruptSource	X		X										
Os_ClearPendingInterrupt	X		X										
Os_GetDetailedError				X									
Os_GetUnhandledIrq	X		X	X	X	X	X	X	X	X			
Os_GetUnhandledExc	X		X	X	X	X	X	X	X	X			
Stack Usage APIs	X		X	X	X	X	X	X	X	X			
Time Conversion Macros	X		X	X	X	X	X	X	X	X			
Os_Init											X		
CheckISRMemoryAccess	X		X	X						X			
CheckTaskMemoryAccess	X		X	X						X			
CallTrustedFunction	X		X									X	
Os_CallFastTrustedFunction	X		X									X	
Os_BarrierSynchronize	X												
Os_GetExceptionContext										X			
Os_SetExceptionContext										X			
Os_EnableInterruptsPreStart											X		

Table 5-157 Calling Context Overview

6 Configuration

MICROSAR Classic OS is configured with Vectors “DaVinci Configurator”.

The descriptions of all OS configuration attributes are described with tool tips within the configuration tool.

They can easily be look up during configuration of the OS component.

Note
The configuration with OIL (OSEK implementation language) is not supported.

7 Cybersecurity

This chapter describes relevant information for a secure integration and configuration, so-called Cybersecurity Manual Items (CMI), of this component to fulfill identified Technical Cybersecurity Requirements (TCR). Additionally, functional security dependencies to other components are described.

The following TCRs are allocated to this component:

- TCR-MSRC-ProtectMemory

8 Glossary

Term	Description
Application	Any software parts that uses the OS. This may include other software modules or customer software (don’t confuse this with the OS-application object).
Category 1 Lock Level	The priority of the highest category 1 ISR
Category 2 Lock Level	The priority of the highest category 2 ISR
Kernel Panic	An inconsistent state of the OS results in kernel panic mode. The OS does not know how to proceed correctly. It goes into freeze as fast as possible (interrupts are disabled, the panic hook is called and afterwards an endless loop is entered).
Memory Management Unit (MMU)	A programmable hardware component responsible for monitoring memory accesses made by the CPU and/or peripheral devices. In comparison to an MPU, the number of MMU regions is not limited by hardware, as they are stored as tables in memory.
Memory Protection Unit (MPU)	A programmable hardware component responsible for monitoring memory accesses made by CPU and/or peripheral devices and triggering an exception upon detection of illegal accesses.
Non-privileged mode	An operation mode of the hardware, where the hardware prevents the execution of specific instructions and/or accesses to specific registers. This mode shall prevent the code from infuencing hardware safety mechhanisms. The non-privileged mode is also known as user-mode. The OS performs the code of non-trusted os-applications as well as the code of trusted os-application with protection in this mode.
Non-trusted function (NTF)	A non-trusted function is a functional service provided by a non-trusted OS application. It runs in the non-privileged mode of the processor with restricted memory rights.
OS-application	An OS object of type application.
Pre-start task	An OS task which may run before StartOS has been called. Within the pre-start task the usage of non-trusted functions is allowed.
Privileged mode	An operation mode of the hardware which allows the execution of instructions and accesses to specific registers, unallowed to access in non-privileged mode. This mode shall allow the code to influence hardware safety mechanisms. The privileged mode is also known as supervisor-mode. The OS performs its own code as well as the code of trusted os-applications (without protection) in this mode.
Supervisor mode	See privileged mode.
Thread	Umbrella Term for OS Task, OS hooks and OS ISR objects
TP Lock Level	The priority the timing protection interrupt
User mode	See non-privileged mode.
X-Signal	MICROSAR Classic OS mechanism which realizes cross core service APIs.

9 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com