VECTOR

MICROSAR OS

Technical Reference

Version 1.7.0

Authors	Anton Schmukel, Ivan Begert, Stefano Simoncelli, Torsten Schmidt, Da He, David Feuerstein, Michael Kock, Martin Schultheiß, Andreas Jehl
Status	Released

Document Information

History

Author	Date	Version	Remarks
Torsten Schmidt	2016-04-27	1.0.0	First release version
Torsten Schmidt	2016-05-18	1.0.1	References to hardware manuals added. Revision work
Torsten Schmidt	2016-06-03	1.0.2	Fix of ESCAN00089598
Torsten Schmidt	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.
Torsten Schmidt	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.
Torsten Schmidt	2016-07-19	1.1.2	Chapter “Memory Mapping Concept” changed. Hints for shorter compile times added. Nesting behavior of OS hooks described.
Ivan Begert	2016-08-11	1.1.3	HSI of ARM family added.
Torsten Schmidt	2016-08-12	1.1.4	Chapter “Memory Mapping Concept” extended. Chapter “Clear Pending Interrupt” extended. Chapter “RH850 Special Characteristics” extended.
Ivan Begert	2016-08-18	1.1.5	HSI of ARM Zynq UltraScale added.
Torsten Schmidt	2016-08-30	1.1.6	HSI of RH850 extended.
Torsten Schmidt	2016-08-31	1.1.7	ORTI Debugging added. Timing Hook Macros reworked. Chapter “Memory Mapping Concept” changed. Chapter “Category 1 Interrupts” extended.
Stefano Simoncelli Torsten Schmidt	2016-09-15	1.1.8	Chapter “Interrupt Source API” extended. HSI chapter for ARM extended
Torsten Schmidt	2016-09-22	1.2.0	VTT OS and Dual Target Concept added. Chapter ORTI Debugging extended.
Anton Schmukel Da He	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.
Torsten Schmidt	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.
David Feuerstein	2016-11-03	1.4.0	PPC HSI extended. Chapter ORTI Debugging extended.
Michael Kock	2016-11-25	1.5.0	Updated chapter Timing Hooks
Martin Schultheiß	2016-12-08	1.6.0	PPC HSI extended. Updated characteristics of VTT OS.
David Feuerstein Andreas Jehl Ivan Begert Stefano Simoncelli	2016-22-12	1.7.0	Updated precautions in PreStartTask. Support new Power PC Derivative: PC580003 Support IAR compiler for ARM ARM Cortex-A HSI added

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

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

Illustrations

Tables

1 Introduction

This document describes the usage and functions of “MICROSAR OS”, an operating system which implements the AUTOSAR BSW module “OS” as specified in [1].

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.

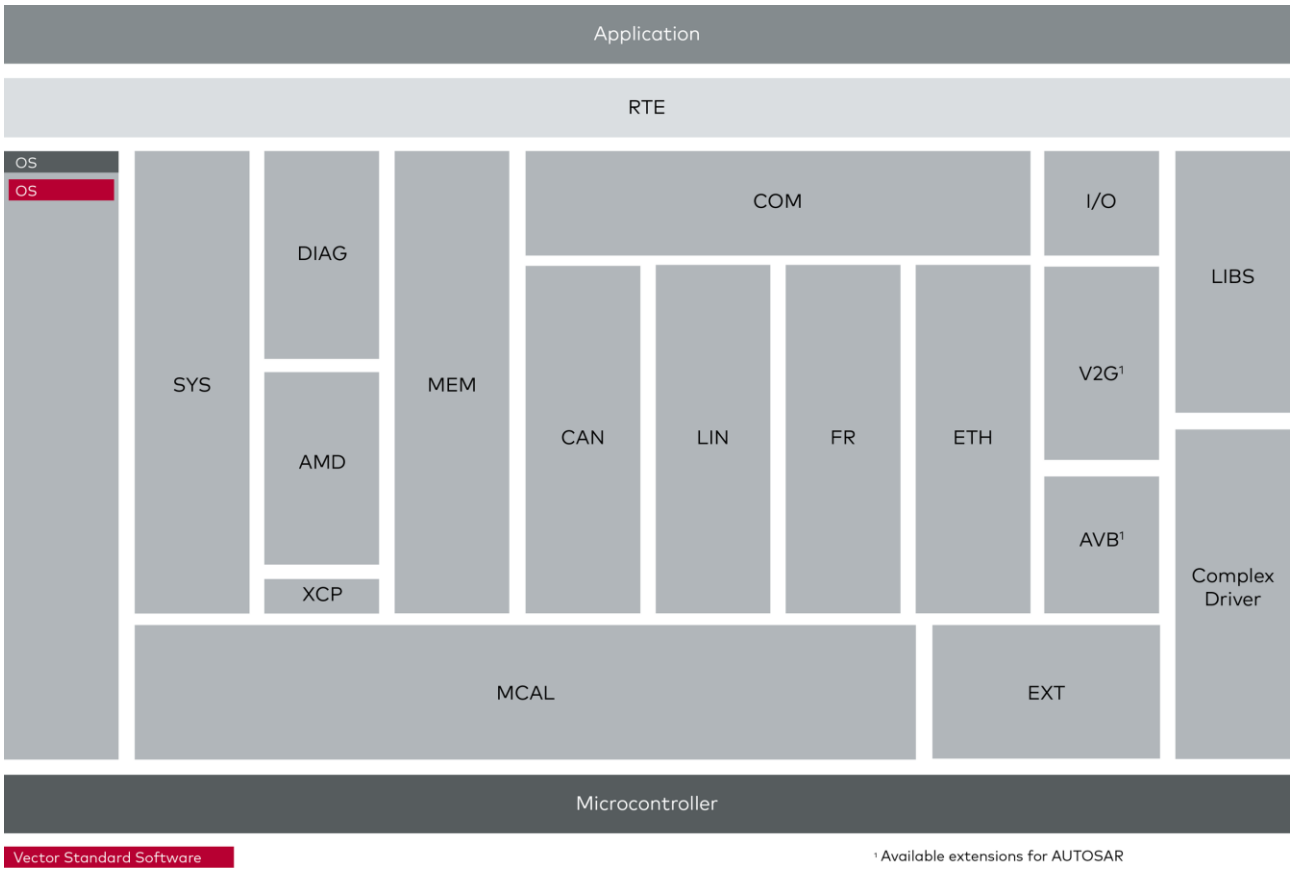

Figure 1-1 AUTOSAR Architecture Overview

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

1.2 Abstract

The MICROSAR 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 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 OS Characteristics

MICROSAR 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 Hardware Overview

The following table summarizes information about MICROSAR OS. It gives detailed information about the derivatives and compilers. As very important information the documentations of the hardware manufacturers are listed. MICROSAR OS is based upon these documents in the given version.

Table Rows

- **Compiler**: List of Compilers MICROSAR OS is working with.
- **Derivative**: This can be a single information or a list of derivatives, MICROSAR OS can be used on.
- **Hardware Manufacturer Document Name**: List of hardware documentation MICROSAR OS is based on.
- **Document Version**: To be able to reference to this hardware documentation its version is very important.

1.4.1 TriCore Aurix

Derivative	Hardware Manufacturer Document Name	Document Version
TC21x TC22x TC23x	User Manual: tc23x_tc22x_tc21x_um_v1.1.pdf	V1.1
	Errata Sheet: TC22x_TC21x_AB_Errata_Sheet_v1_2_03804A.pdf	V1.2
TC24x	Target Specification: tc24x_ts_v2.0_OPEN_MARKET.pdf	V2.0
TC26x	User Manual: tc26xB_um_v1.3._usermanual_rev1v3.pdf	V1.3
	Errata Sheet: TC26x_BB_Errata_Sheet_rev1v2_03989A_2016-04-18.pdf	V1.2
TC27x	User Manual: tc27xD_um_v2.2_UserManual_rev2v2_2014-12.pdf	V2.2
	Errata Sheet: TC27x_BC_Errata_Sheet_rev1v5_2015_09_16.pdf	V1.5
TC29x	User Manual: tc29xB_um_v1.3._TC29x_B-Step_User_Manual_rev_1v3_2014_12.pdf	V1.3
	Errata Sheet: TC29x_BA_Errata_Sheet_v1_0.pdf	V1.0

Table 1-2 Supported TriCore Aurix Hardware

Tasking	v4.2r2
HighTec (GNU)	V4.6.3.0

Table 1-3 Supported TriCore Aurix Compilers

1.4.2 Power PC

Derivative	Hardware Manufacturer Document Name	Document Version
MPC574xBD	Freescale Semiconductor MPC5746C Reference Manual	Rev. 2.1, 06/2015
MPC574xC1	Freescale Semiconductor MPC5746C Reference Manual	Rev. 2.1, 06/2015
MPC574xC2	NXP MPC5748G Reference Manual	Rev. 4, 07/2015
MPC574xG	NXP MPC5748G Reference Manual	Rev. 4, 07/2015
	NXP Safety Manual for MPC5748G	Rev. 2, 01/2016
MPC574xK	ST SPC574Kxx Reference Manual	Rev. 5, 08/2015
MPC574xM	Freescale Semiconductor MPC5746M Reference Manual	Rev. 5.1, 04/2014
MPC574xP	Freescale Semiconductor MPC5744P Reference Manual	Rev. 5.1, 02/2015
	NXP Safety Manual for MPC5744P	Rev. 3, 06/2014
MPC574xR	NXP MPC5746R Reference Manual	Rev. 6, 03/2016
MPC577xK	Freescale Semiconductor MPC5775K Reference Manual	Rev. 4, 12/2015
MPC577xM	NXP MPC5777M Reference Manual	Rev. 4, 04/2015
MPC577xN	Freescale Semiconductor MPC5774N Reference Manual	Rev. 2, 02/2014
PC580003	Freescale Semiconductor PC580003 Reference Manual	Rev. 2, 05/2014
SPC58ECxx	ST SPC584Cx/SPC58ECx Reference Manual	Rev. 1, 10/2015
SPC58EGxx	ST SPC58NE84x/SPC58xG84x Reference Manual	Rev. 2, 02/2016
SPC58NGxx	ST SPC58NE84x/SPC58xG84x Reference Manual	Rev. 2, 02/2016
SPC582Bxx	ST SPC582Bx Reference Manual	Rev. 1, 08/2015
SPC584Bxx	ST SPC584Cx/SPC58ECx Reference Manual	Rev. 1, 10/2015
SPC584Cxx	ST SPC584Cx/SPC58ECx Reference Manual	Rev. 1, 10/2015
SPC584Gxx	ST SPC58NE84x/SPC58xG84x Reference Manual	Rev. 2, 02/2016

Table 1-4 Supported Power PC Hardware

Windriver DiabData	5.9.4.x
Green Hills (GHS)	2014.1.6

Table 1-5 Supported Power PC compilers

1.4.3 ARM

Derivative	Hardware Manufacturer Document Name	Document Version
S6J32xx	Cypress S6J3200 Series Hardware Manual	Rev. 4.0, 09/2015
ZUxxx	XILINX Zynq UltraScale+ MPSoc Technical Reference Manual	v1.2, 06/2016
iMX6xx	i.MX 6Dual/6Quad Applications Processor Reference Manual	Rev. 3, 07/2015

Table 1-6 Supported ARM Hardware

Green Hills (GHS)	2013.5.4
IAR	V7.50.1

Table 1-7 Supported ARM compilers

1.4.4 RH850

Derivative Family	Hardware Manufacturer Document Name	Document Version
RH850 C1M RH850 C1H	RH850/C1x User's Manual: Hardware	Rev.1.00 Mar 2015
RH850 D1x	RH850/D1L/D1M Group User's Manual: Hardware	Rev.2.01 Aug 2016
RH850 E1x FCC2	RH850/E1x-FCC2 User's Manual: Hardware	Rev.0.50 Apr 2015
RH850 E1x FCC1	RH850/E1x-FCC1 User's Manual: Hardware	Rev.0.50 Jul 2014
RH850 E1L	RH850/E1L User's Manual: Hardware	Rev.1.10 Apr 2016
RH850 E1M	RH850/E1M-S User's Manual: Hardware	Rev.1.10 Apr 2016
RH850 F1H	RH850/F1H Group User's Manual: Hardware	Rev.1.12 May 2016
RH850 F1L	RH850/F1L Group User's Manual: Hardware	Rev.1.33 Apr 2016
RH850 F1K	RH850/F1K Group User's Manual: Hardware	Rev.1.00 Jun 2016
RH850 F1M	RH850/F1M Group User's Manual: Hardware	Rev.1.03 May 2016
RH850 P1HC	RH850/P1x-C Group User's Manual: Hardware	Rev.1.10 Jul 2016
RH850 P1MC	RH850/P1x-C Group User's Manual: Hardware	Rev.1.10 Jul 2016
RH850 P1M	RH850/P1x Group User's Manual: Hardware	Rev.1.00 Jul, 2015
RH850 R1L	RH850/R1x Group User's Manual: Hardware	Rev.1.31 Jun 2016
G3K Core	RH850G3K User's Manual: Software	Rev.1.20 Apr 2016
G3KH Core	RH850G3KH User's Manual: Software	Rev.1.10 Jul 2016
G3M Core	RH850G3M User's Manual: Software	Rev.1.30 Jun 2016
G3MH Core	RH850G3MH User's Manual: Software	Rev.1.00 Mar 2015

Table 1-8 Supported RH850 Hardware

Green Hills (GHS)	V6.1.4 2013.5.5 V6.1.6 2015.1.5

Table 1-9 Supported RH850 Compilers

1.4.5 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].

1.4.5.1 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
Stack Sharing	Not supported

Table 1-10 VTT OS characteristics

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

2 Functional Description

2.1 General

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

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

On the other hand MICROSAR OS extends the AUTOSAR OS standard with numerous functions. These extensions in function are described in detail in chapter 2.21.1.

2.2 MICROSAR 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 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 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 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 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 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 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.

2.2.5 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 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 OS In MICROSAR 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.

2.2.6 Interrupt API Deviation

Specified Behavior	The API functions SuspendOSInterrupts() and ResumeOSInterrupts() are allowed within a category 1 ISR
Deviation Description	In MICROSAR OS it is not allowed to use SuspendOSInterrupts() and ResumeOSInterrupts() within a category 1 ISR.
Deviation Reason	The function SuspendOSInterrupts() lowers the current interrupt level when used in a category 1 ISR. This may lead to data inconsistencies if another category 1 ISR occurs. Therefore those functions are not allowed.

2.2.7 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 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.

2.2.8 IOC

Specified Behavior	IocSend/IocWrite APIs have an IN parameter. The parameter will be passed by value for primitive data elements and by reference for all other types. The data type is configured in “OsIocDataTypeRef”.
Deviation Description	The configurator does not evaluate information in “OsIocDataTypeRef”. Instead it evaluates the parameter “OsIocDataType“. Primitive data types are passed by value. The configurator identifies the following strings as primitive data types: “uint8”, “uint16” and “uint32”. All other data types are passed by reference.
Deviation Reason	Usage of “OsIocDataType” reduces dependencies and complexity of the OS configurator.

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 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 OS is not possible to distinguish between stack violation and other memory violation

2.2.10 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().

2.3 Stack Concept

MICROSAR 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
Kernel stack	1 per core	> OS memory exception handling > Os_PanicHook()
Protection stack	0..1 per core	> ProtectionHook() > OS API calls > Os_PanicHook()
Error stack	0..1 per core	> ErrorHooks (global and OS-application specific) > OS API calls > Category 1 ISRs > Os_PanicHook()
Shutdown stack	0..1 per core	> ShutdownHooks (global and OS-application specific) > OS API calls > Os_PanicHook()
Startup stack	0..1 per core	> StartupHooks (global and OS-application specific) > OS API calls > Category 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 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 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 1 ISRs > Os_PanicHook()
Task stacks	1..n	> Tasks > OS API calls > OS ISR wrapper > Trusted functions > Alarm callback functions > Pre / PostTaskHook() > Category 1 ISRs > Os_PanicHook()
IOC receiver pull callback stack	0..1 per core	> IOC receiver pull callback functions

Table 2-1 MICROSAR 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.

These basic tasks must behave cooperative to each other (must not preempt each other).

2.3.1.2 Activation

The attribute “OsTaskStackSharing” of a basic task has to be set to TRUE.

All basic tasks on the same task priority and with activated task stack sharing are cooperative and share one task stack.

The size of the shared task stack is the maximum of all configured task stack sizes of tasks with the same priority.

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

2.3.1.3 Usage

The feature is automatically used by the OS. Tasks which are cooperative to each other are sharing the same stack.

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 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 supervision by memory protection unit (MPU) (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 last 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!

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 Supervision by MPU

2.3.5.1 Description

During the whole runtime of the OS the current active stack is supervised by the MPU of the microcontroller. Therefore the OS reserves one MPU region which is reprogrammed by the OS with each stack switch.

Stack overflows cannot happen since the MPU 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.

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 initializes one MPU region for stack supervision.

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

- When configuring additional MPU regions included memory region must never overlap with any OS stack sections.
- By using an OS generated linker command file (see 4.3.2) 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 (in dependency of the stack growth direction; see Figure 2-1). 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.
- The linkage of the safety gap is mandatory. Otherwise a stack violation of the stack with the lowest address cannot be detected.

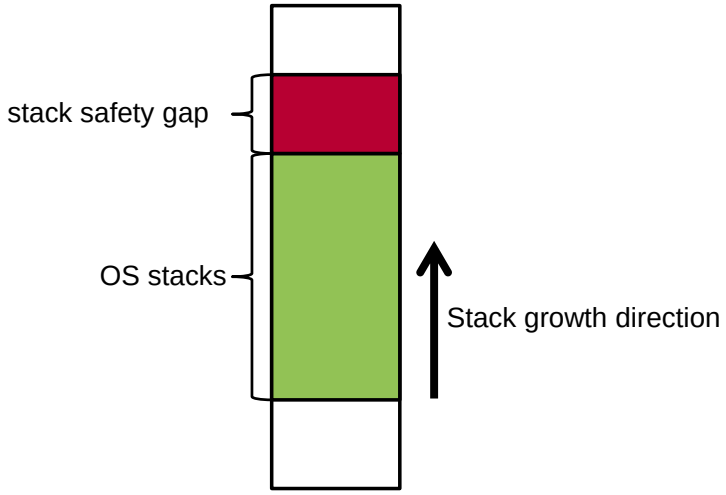

Figure 2-1 Stack Safety Gap

Caution
Don’t configure MPU regions which grant access to any OS stacks

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.

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 5.6).

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. The implementation of those functions and their effects on the application depends on the timing protection configuration.

2.4.1.1 Interrupt Handling in SC1 / SC3

Without configured timing protection the interrupt APIs are implemented as follows:

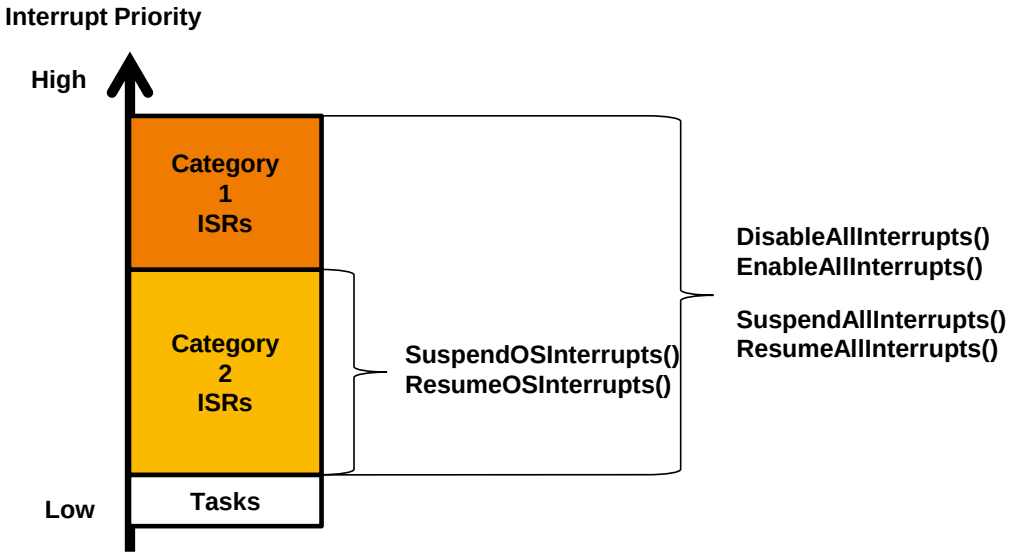

Figure 2-2 Interrupt APIs in SC1 / SC3

DisableAllInterrupts() EnableAllInterrupts()	The functions disable all interrupts by manipulate a global interrupt mask / disable flag
SuspendAllInterrupts() ResumeAllInterrupts()	
SuspendOSInterrupts() ResumeOSInterrupts()	The functions disable category 2 interrupts by manipulate the interrupt priority / level

Note
The lowest priority of all category 1 ISRs is called OS system level.

2.4.1.2 Interrupt Handling in SC2 / SC4

With configured timing protection the interrupt APIs are implemented as follows:

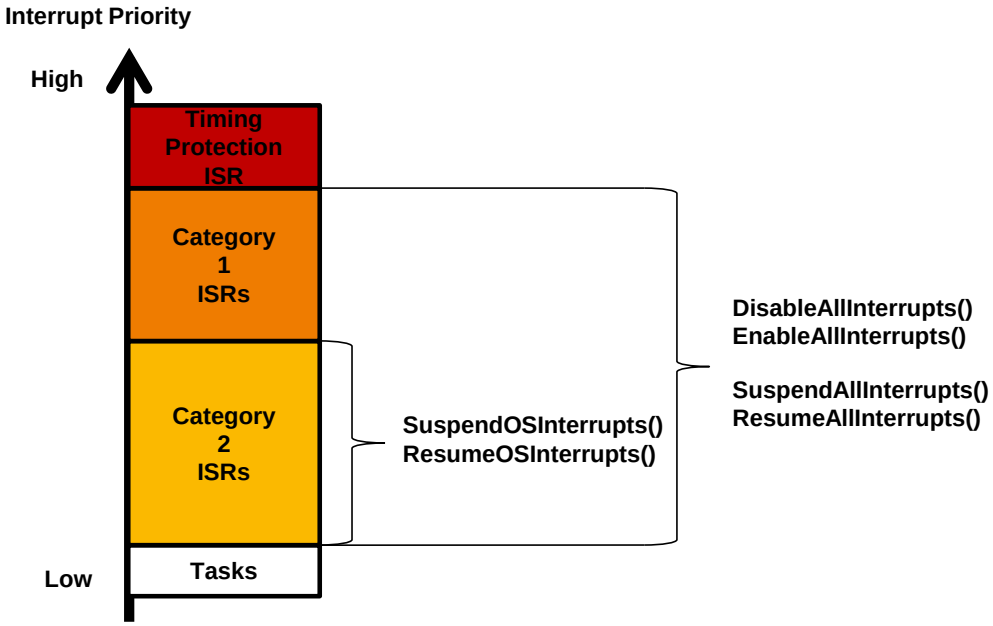

Figure 2-3 Interrupt API in SC2 / SC4

DisableAllInterrupts() **EnableAllInterrupts()**	The functions disable all interrupts except the timing protection interrupt ▶ By manipulate a global interrupt mask / disable flag (if the timing protection interrupt is not maskable) ▶ By manipulate the interrupt priority / level (if the timing protection interrupt is maskable) See chapter 2.19 for details
SuspendAllInterrupts() **ResumeAllInterrupts()**	
SuspendOSInterrupts() **ResumeOSInterrupts()**	The functions disable category 2 interrupts by manipulate the interrupt priority / level

2.4.2 Interrupt Vector Table

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

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

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

2.4.3 Nesting of Category 2 Interrupts

2.4.3.1 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)

2.4.3.2 Activation

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

2.4.4 Category 1 Interrupts

2.4.4.1 Implementation of Category 1 ISRs

MICROSAR 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 OS abstracts the needed compiler keywords.

Implement a category 1 ISR

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

2.4.4.2 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 */
}
```

2.4.4.3 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 the interrupts by directly manipulating the global interrupt flag and / or current interrupt priority to allow the category 1 ISRs

2.4.4.4 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 2.3 for an overview).

Note
Although the interrupt priorities are initialized by MICROSAR OS there is no API to enable or acknowledge category 1 ISRs. The interrupt control registers have to be accessed directly.

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 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.

2.4.5 Initialization of Interrupt Sources

Through the OS configuration MICROSAR 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 OS generates data structures for initializing the interrupt controller. It initializes the interrupt priority and its core assignment.

2.4.6 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 5.5.1 for details).

In case of an unhandled interrupt request MICROSAR OS calls the ProtectionHook() with the parameter E_OS_SYS_PROTECTION_IRQ.

2.5 Exception Concept

2.5.1 Exception Vector Table

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

In a multi core multiple vector tables may be generated.

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

Note
In a SC3 and SC4 system MICROSAR 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 5.5.3 for details).

In case of an unhandled exception request MICROSAR OS calls the ProtectionHook() with the parameter E_OS_PROTECTION_EXCEPTION.

2.6 Timer Concept

2.6.1 Description

MICROSAR 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 OS provides conversion macros (which are based on the hardware counter configuration) to convert from hardware ticks to time and vice versa (see for 5.8 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 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.

2.8 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 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.

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 2-3 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.

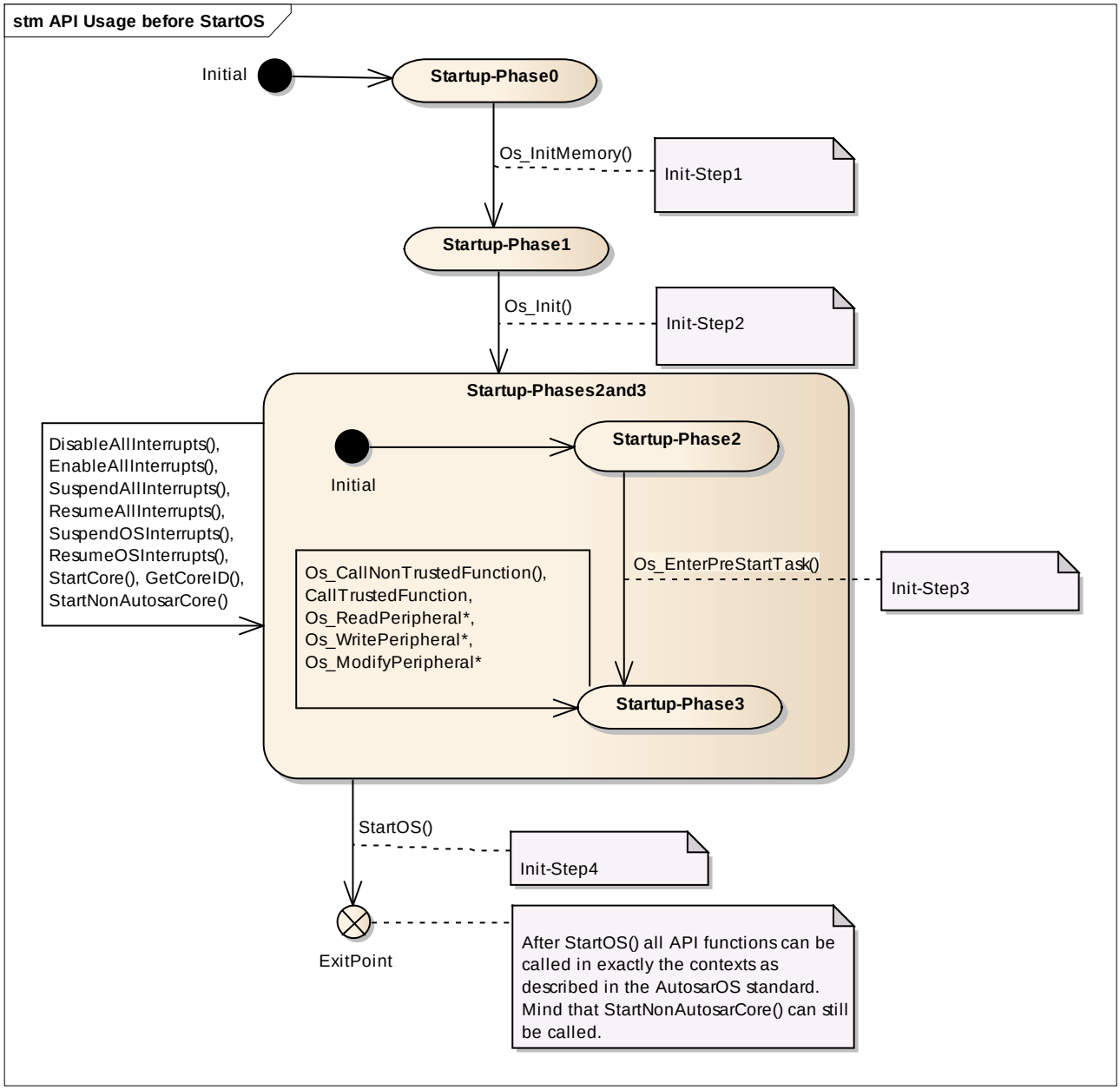

Figure 2-4 API functions during startup

2.11 Single Core Startup

This chapter shows some examples how MICROSAR 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 OS.
3. Optional: calls to StartNonAutosarCore() to start additional cores which are independent of MICROSAR OS.
4. Optional: call StartOS() to start MICROSAR 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 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 the internal OS 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 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 if its internal data (e.g. memory access violation during ProtectionHook()) ▶ Will result in call of the Os_PanicHook() to inform the application. ▶ Afterwards the OS disables all interrupts and enters an infinite loop.

Table 2-4 Types of OS Errors

2.14 Error Reporting

MICROSAR 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 OS introduces new service IDs for own services.

Reference

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

2.14.2 Extension of Error Codes

MICROSAR 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

Table 2-5 Extension of Error Codes

Reference

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

2.14.3 Detailed Error Codes

MICROSAR 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.

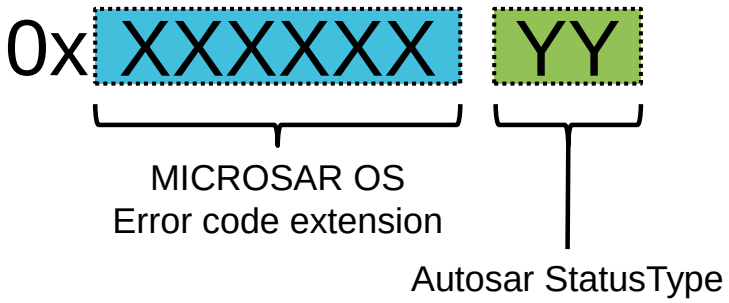

Figure 2-5 MICROSAR OS Detailed Error Code

Within OS Hook routines the error code can be obtained by calling Os_GetDetailedError() (see 5.5.1 for details).

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

Reference
The detailed error codes are listed in the file `OsInt.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 OS implements independent schedulers and dispatchers for each core.

2.15.2 Multi Core Data Concepts

The multi core data concept of MICROSAR 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 OS offers the X-Signal concept (see 3.7 for details).

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 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 Startup of a Multi Core System

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

MICROSAR 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.

2.15.6 Spinlocks

Synchronization of cores is done by

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

2.15.7 Cache

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

2.15.8 Shutdown

2.15.8.1 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

2.15.8.2 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].

2.15.8.3 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 OS offers two software utilities to support OS debugging.

ORTI	MICROSAR 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
TimingHooks	MICROSAR OS provides macros which may be used for debugging purposes (also suitable for third party tools). See chapter 3.8 for details.

2.16.2 Activation

ORTI and TimingHooks may be switched on within the OsDebug container.

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 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 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 OS.

Caution

Due to performance reasons the following OS services are not traced by ORTI service tracing:

- GetSpinlock (for optimized spinlocks)
- TryToGetSpinlock (for optimized spinlocks)
- ReleaseSpinlock (for optimized spinlocks)
- IOC
- Os_GetVersionInfo

2.17 Memory Protection

MICROSAR OS uses memory protection facilities of a processor to achieve freedom from interference between OS applications and cores. For this purpose it may use the system MPU and the core MPUs.

2.17.1 Usage of the System MPU

In multi core systems whereas the cores have different levels of diagnostic coverage it may be necessary to use a system MPU to achieve freedom of interference between cores.

A system MPU allows configuring access rights for cores to access specific memory ranges.

The system MPU is only initialized once during startup of the OS. It is never reprogrammed during runtime.

With a system MPU other potential bus masters (DMA, FlexRay) can be isolated to achieve freedom from interference.

This is done with the following steps:

Step	Toolchain phase
Set up a SC3 system	Configuration of OS
Configure memory regions	
Assign the memory region to the system MPU	

2.17.2 Usage of the Core MPUs

The core MPUs 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/executable) have to be granted explicitly to software parts.

This is done with the following steps:

Step	Toolchain phase
Set up a SC3 system	Configuration of OS
Configure memory regions	
Assign the memory region to a core MPU	
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 4.3.1.1)	Compilation
Use OS generated linker command files to locate the sections into memory (see 4.3.2)	Linkage

2.17.3 Configuration Aspects

A memory region is typically configured by

- Specify a start and end address by number, or by linker labels (see 4.3.3 for OS generated section labels)
- Specify access rights to this region (a pre-defined set of access rights is referable)
- Specify the validity of the region by ID (e.g. PID / ASID / Protection Set)
- Specify to which memory protection unit the region belongs (e.g. Core MPU / System MPU)
- Specify the owner of the region

The owner of the memory region distinguishes the runtime behavior of the hardware MPU regions (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 MPU.

Note
The number of available hardware MPU regions is limited by hardware!

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

2.17.3.1 Static MPU Regions

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

2.17.3.2 Dynamic MPU Regions

If an owner is specified for a memory region MICROSAR OS initializes a hardware MPU region to be dynamically reprogrammed during OS runtime. Whenever the owner of the memory is active during runtime a specific hardware MPU region 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.

2.17.3.3 Freedom from Interference

MICROSAR 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.

Memory Region Owner	Access Granted To	Access Denied For
OS application	Runtime objects of this OS application > Tasks > ISRs > IOC callbacks > Non-trusted functions > Application specific hooks	> Other non-trusted OS applications and its applications objects
Task	> The owning task	> Other non-trusted OS applications and its applications objects > Other runtime objects of the belonging OS application
ISR	> The owning ISR	

2.17.4 Stack Monitoring

MICROSAR OS uses one memory region of the MPU to supervise the current stack. This is the default handling in SC3 and SC4 systems. See 2.3.5 for details.

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

2.17.5 Protection Violation Handling

Upon any memory protection violation the OS

- Switches to the kernel stack
- Informs the application by execution of the ProtectionHook()

2.17.6 Optimized / Fast Core MPU Handling

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

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

MICROSAR 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.

2.17.7 Recommended Configuration

MICROSAR 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 2-6 Recommended Configuration MPU Access Rights

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

MICROSAR OS implements these two functions (see 5.9 for Details)

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 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 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 on the highest possible priority of the microcontroller.	The exception source has no interrupt level.
DisableAllInterrupts() **EnableAllInterrupts()** **SuspendAllInterrupts()** **ResumeAllInterrupts()**	The functions disable all ISRs (with exception of timing protection interrupt) by manipulate the interrupt priority / level	The functions disable all ISRs (with exception of timing protection interrupt) by manipulate a global interrupt mask / disable flag
SuspendOSInterrupts() **ResumeOSInterrupts()**	The functions disable category 2 interrupts by manipulate the interrupt priority / 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 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.

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 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 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	Not Used	Not 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 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 commination is realized with multiple sender queues. The receiver application does an even multiplexing on all sender queues when calling the receive function (see figure).

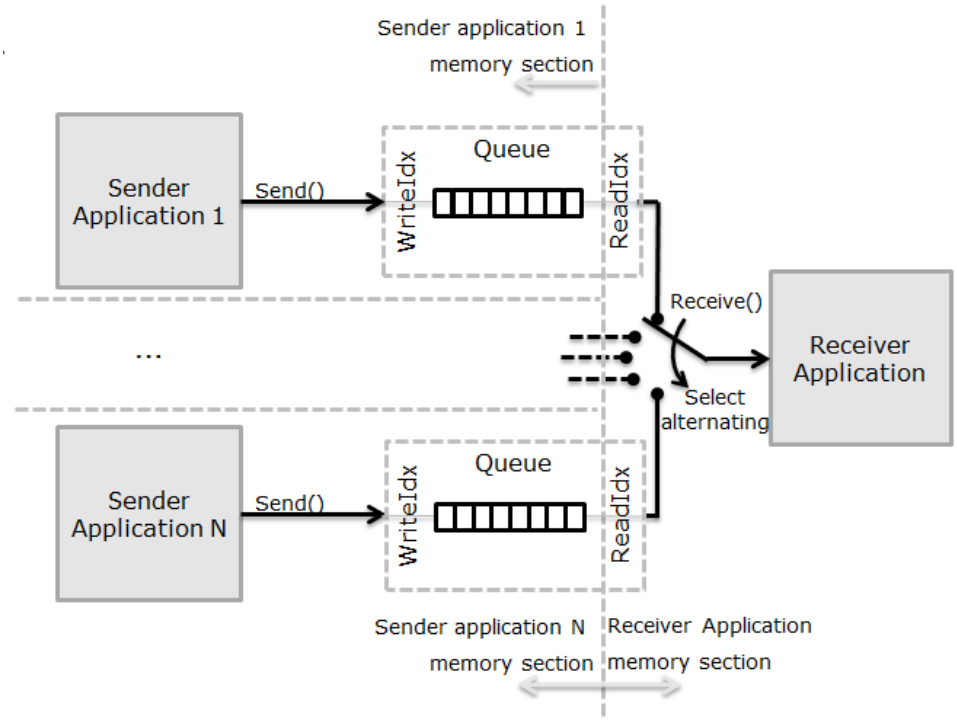

Figure 2-6 N:1 Multiple Sender Queues

2.20.5.2 IOC Spinlocks

Note
During generation of OS data structures, if MICROSAR 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.

2.21 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 OS allows gradually restricting of access rights of trusted OS applications.

Trusted OS applications may be restricted by memory access or by processor 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 supervisor 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.

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 OS Applications in User Mode

2.21.2.1 Description

Such OS applications can have read/write access to nearly the whole memory (except stack regions), but they are running in user mode. This is also applied to all runtime objects (Tasks / ISRs / Trusted functions) assigned to this OS application.

Note

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

2.21.2.2 Activation

Set “OsApplicationIsPrivileged” to FALSE.

2.21.2.3 Dependencies

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

2.21.3 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.

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

Hook Name	Processor Mode	Access Rights	Interrupt State
StartupHook	Supervisor	Trusted	Disabled up to system level
ErrorHook			
ShutdownHook			Globally disabled
ProtectionHook			
StartupHook_<OS application name>	Depending on the configuration of the owning OS application		Disabled up to system level
ErrorHook_<OS application name>			
ShutdownHook_<OS application name>			Globally disabled
Os_PanicHook	Supervisor	Trusted	Globally disabled
PreTaskHook	Supervisor	Trusted	Globally disabled
PostTaskHook	Supervisor	Trusted	Globally disabled
AlarmCallbacks	Supervisor	Trusted	Globally disabled
IOC receiver pull callbacks	Depending on the configuration of the owning OS application		Disabled up to system level (execution is done within ISR context)

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.

3 Vector Specific OS Features

This chapter describes functions which are available only in MICROSAR 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 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
Linking	OS and optimized spinlock variables are placed into different dedicated memory sections (see 4.3.1).	

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 Peripheral Access API

3.2.1 Description

MICROSAR 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.

3.2.2 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

3.2.3 Usage

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

Reference
The API service functions themselves are described in chapter 5.1.

3.2.4 Dependencies

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

3.2.5 Alternatives

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

3.2.6 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.

3.3 Trusted Function Call Stubs

3.3.1 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 OS is able to generate these stub functions.

3.3.2 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”.

3.3.3 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.

3.3.4 Dependencies

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

3.4 Non-Trusted Functions (NTF)

3.4.1 Description

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

- 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.

3.4.2 Activation

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

Special care needs to be taken when configuring stacks for non-trusted functions. For correct stack handling the following attributes need to be configured for each non-trusted function:

Non-Trusted Function Attribute	Description
OsNonTrustedFunctionStackSize	Specifies the stack size for one instance of the NTF.
OsNonTrustedFunctionStacks	Specifies the number of stacks which are generated for the NTF. Note: This attribute limits the number of parallel calls to the NTF. Once the maximum number is reached any call to the NTF issues an error.
OsNonTrustedFunctionCaller	Specifies an OS application which may call the NTF.
OsNonTrustedFunctionAppStacks	Specifies the number of parallel NTF calls from the calling OS application. Note: It must not exceed the attribute OsNonTrustedFunctionStacks.
OsNonTrustedFunctionAppRef	Reference to the calling OS application.

3.4.3 Usage

Similar to the CallTrustedFunction() API of the AUTOSAR OS standard MICROSAR OS implements an additional service which is called Os_CallNonTrustedFunction() (see chapter 5.3 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.

3.4.4 Dependencies

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

3.5 Interrupt Source API

3.5.1 Description

MICROSAR 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

(See 5.4 for API details).

3.6 Pre-Start Task

3.6.1 Description

MICROSAR 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 5.13 lists the OS API functions which may be used within the Pre-Start task.

3.6.2 Activation

> Define a basic task

> 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

3.6.3 Usage

1. Execute Startup Code
2. Call `Os_InitMemory()`
3. Call `Os_Init()`
4. Call `Os_EnterPreStartTask()` (see 5.2 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 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.

3.6.4 Dependencies

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

3.7 X-Signals

3.7.1 Description

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

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

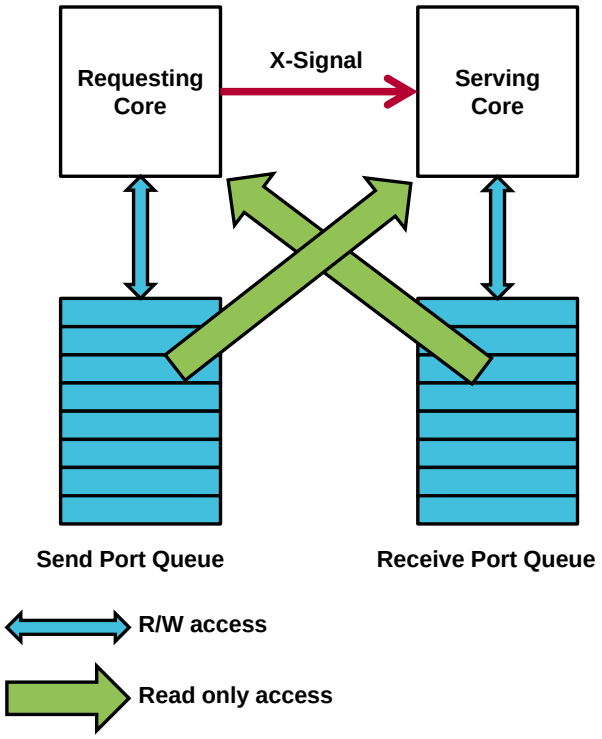

Figure 3-1 X-Signal

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.

3.7.1.1 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.

Additionally category 2 ISRs may only use cross core API services if they allow nesting.

The following table gives an overview.

Logical Priority	ISR Nesting	Synchronous Cross Core API Calls
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!

3.7.1.2 Notes on Mixed Criticality Systems

MICROSAR 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.

3.7.2 Activation

X-Signals must be configured explicitly in a multi core environment. See chapter 4.5 for details.

3.8 Timing Hooks

3.8.1 Description

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

> Activation (arrival) of a task or ISR

- These allow an external tool to trace all activations of task as well as further arrivals (e.g. setting of an event or the release of a semaphore with transfer to another task.
- They allow external tools to visualize the arrivals and to measure the time between them in order to allow a schedule-ability analysis.

> Context switch

- These allow external tools to trace all context switches from task to ISR and back as well as between tasks. So external tools may visualize the information or measure the execution time of tasks and ISRs.

> 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.

Within MICROSAR OS code the timing hooks are called. Additionally it provides empty hooks by default.

The application may decide to implement any of the hooks by itself. The empty OS default hook is then replaced by the application implemented hook.

3.8.2 Activation

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

3.8.3 Usage

The timing hooks may be implemented in the configuration specified header. All available macros are introduced in chapter 5.11.

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

Thus the timing hooks must be disabled for a safety serial production system.

3.9 Kernel Panic

If MICROSAR 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 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 5.12)
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

3.10 Generate callout stubs

3.10.1 Description

MICROSAR 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”.

3.10.2 Activation

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

3.10.3 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>
 *********************************************************************/

}
```

4 Integration

4.1 Compiler Optimization Assumptions

MICROSAR 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

4.1.1 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”

4.2 Hardware Software Interfaces (HSI)

The following chapter describes the Hardware-Software Interface for the supported processor families of the MICROSAR OS.

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

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.

Additionally platform specific characteristics of the OS are described here.

4.2.1 TriCore Aurix Family

4.2.1.1 Context

- A0-A15
- D0-D15
- PSW
- PCXI
- DPR0L, DPR0H

Note
The register A8 is exclusively used by the OS to hold the pointer to the current thread. Thus any addressing modes which would use A8 register are not possible.

4.2.1.2 Core Registers

- ICR
- SYSCON
- PCXI
- FCX
- LCX
- PSW
- PC
- DBGSR
- DPRxL, DPRxH
- CPRxL, CPRxH
- DPRE0 – DPRE3
- DPWE0 – DPWE3
- CPXE0 – CPXE3

4.2.1.3 Interrupt Registers

- INT_SRC0 – INTSRC255

4.2.1.4 GPT Registers

- T2, T3, T6
- T2CON, T3CON, T6CON
- CAPREL

4.2.1.5 STM Registers

- TIM0, TIM5, TIM6
- CMCON
- CAP
- CMP0, CMP1

4.2.1.6 Aurix Special Characteristics

- The exception handler for trap class 1 is implemented by the OS
- The exception handler for trap class 6 is implemented by the OS

Caution

The TriCore Hardware enforces that a configured MPU region must be followed by at least 15 padding bytes before the next region may be started.

MICROSAR OS obey to this rule within the generated linker scripts. For other additional configured MPU regions the user has to take care to fulfill this requirement

Figure 4-1 Padding bytes between MPU regions

Caution

Due to MPU granularity all start addresses and end addresses of configured MPU regions must be a multiple of 8.

MICROSAR OS programs the MPU to grant access to the memory region between start address and end address.

> Access to configured start address itself is granted

> Access to configured end address is prohibited

Caution

MICROSAR OS does not use the System MPU to achieve freedom of interference between cores.

This has to be done by the application.

The system MPU has to be initialized by a lockstep core. It must not be accessed by non-lockstep cores.

Note

All stack sizes shall be configured to be a multiple of 8

Expert Knowledge

For proper context management exception handling the LCX should be initialized during startup code that it does not point to the last available CSA.

In this way some CSAs are reserved which can be used within the context exception handling for further function calls.

4.2.1.7 PSW handling

PSW.RM bit handling	MICROSAR OS sets the rounding mode bits to 0 upon start of a thread.
PSW.S bit handling	MICROSAR OS sets the safety task identifier bit to 1 for trusted software parts and to 0 for non-trusted software parts.
PSW.IS bit handling	MICROSAR OS sets the interrupt stack bit to 1. Thus automatic hardware stack switch is not supported.
PSW.GW bit handling	MICROSAR OS sets the global address register write permission to 0. Write permission to A0, A1, A8 and A9 are not allowed.
PSW.CDE bit handling	MICROSAR OS sets the call depth enable bit to 1 upon start of a thread. Call depth counting is enabled.
PSW.CDC bits handling	MICROSAR OS sets the call depth counter to 1 upon start of a thread.

4.2.2 RH850 Family

4.2.2.1 Context

- R1 ... R31
- PC
- PSW
- PMR
- LP
- SP
- EIPC, EIPSW
- FPSR, FPEPC
- ASID
- MPLA0, MPUA0

4.2.2.2 Core Registers

- PC
- PSW
- PMR
- LP
- SP
- ASID
- SCCFG
- SCBP
- EIPC
- EIPSW
- EIWR
- FPSR
- FPEPC
- EBASE
- INTBP
- INTCFG
- CTPC
- EIIC
- FEIC
- FEPC
- FEPSW
- HTCFG0

Note

The register EIWR is exclusively used by the OS to hold the pointer to the current thread.

4.2.2.3 MPU Registers

- MPM
- MPRC
- MPLA0 ... MPLA15
- MPUA0 ... MPUA15
- MPAT0 ... MPAT15

4.2.2.4 INTC Registers

- EIC0 ... EIC511
- IBD0 … IBD511

4.2.2.5 Inter Processor Interrupt Control Registers

- IPIR_CH0
- IPIR_CH1
- IPIR_CH2
- IPIR_CH3

4.2.2.6 Timer TAUJ Registers

- TAUJnCDR
- TAUJnCNT
- TAUJnCMUR
- TAUJnCSR
- TAUJnCSC
- TAUJnTE
- TAUJnTE0
- TAUJnTE1
- TAUJnTS
- TAUJnTS0
- TAUJnTS1
- TAUJnTT
- TAUJnTT0
- TAUJnTT1
- TAUJnTO
- TAUJnTO0
- TAUJnTO1
- TAUJnTOE
- TAUJnTOE0
- TAUJnTOE1
- TAUJnTOL
- TAUJnTOL0
- TAUJnTOL1
- TAUJnRDT
- TAUJnRDT0
- TAUJnRDT1
- TAUJnRSF
- TAUJnRSF0
- TAUJnRSF1
- TAUJnRSF2

- TAUJnCMOR
- TAUJnTPS
- TAUJnTPS0
- TAUJnBRS
- TAUJnBRS0
- TAUJnBRS1
- TAUJnTOM
- TAUJnTOM0
- TAUJnTOM1
- TAUJnTOC
- TAUJnTOC0
- TAUJnTOC1
- TAUJnRDE
- TAUJnRDE0
- TAUJnRDE1
- TAUJnRDM
- TAUJnRDM0
- TAUJnRDM1

4.2.2.7 Timer STM Registers

- STMnCKSEL
- STMnTS
- STMnTT
- STMnCSTR
- STMnSTR
- STMnSTC
- STMnIS
- STMnRM
- STMnCNT0L
- STMnCNT0H
- STMnCMP0AL
- STMnCMP0AH
- STMnCMP0BL
- STMnCMP0BH
- STMnCMP0CL
- STMnCMP0CH
- STMnCMP0DL
- STMnCMP0DH
- STMnCNT1
- STMnCMP1A
- STMnCMP1B
- STMnCMP1C
- STMnCMP1D
- STMnCNT2
- STMnCMP2A
- STMnCMP2B
- STMnCMP2C
- STMnCMP2D
- STMnCNT3

- STMnCMP3A
- STMnCMP3B
- STMnCMP3C
- STMnCMP3D

4.2.2.8 Timer OSTM Registers

- OSTMnCMP
- OSTMnCNT
- OSTMnTO
- OSTMnTOE
- OSTMnTE
- OSTMnTS
- OSTMnTT
- OSTMnCTL
- OSTMnEMU

4.2.2.9 RH850 Special Characteristics

Note

> The exception handler for TRAP1 (offset = 0x50) is implemented by the OS.

Note

In SC3 / SC4 systems

> The exception handler for TRAP0 (offset = 0x40) is implemented by the OS.

> The exception handler for MIP/MDP (offset = 0x90) is implemented by the OS.

Caution

The MPU in RH850 has a granularity of 4Byte. Each data section must have 4Byte address alignement.

Caution

Due to MPU granularity the start address of any configured MPU region must be a multiple of 4Byte.

The end address of any configured MPU region must be the address of the last valid Byte of the section.

MICROSAR OS programs the MPU to grant access to the memory region from start address and end address.

Note

All stack sizes shall be configured to be a multiple of 4

Note

Tiny data area (TDA) and zero data area (ZDA) addressing are not supported.

Note

For multicore-core derivatives, the stack used before StartOS should be linked into the respective core local RAM areas.

4.2.2.10 PSW Register Handling

PSW.EBV	MICROSAR OS sets PSW.EBV to 1 upon Os_Init().
PSW.UM	MICROSAR OS sets PSW.UM to 0 for trusted software parts and to 1 for non-trusted software parts.
PSW.NP	MICROSAR OS sets PSW.NP to 1 to disable FE level interrupts and to 0 to enable FE level interrupts.
PSW.ID	MICROSAR OS sets PSW.ID to 1 to disable EI level interrupts and to 0 to enable EI level interrupts.
PSW.CU0	MICROSAR OS sets PSW.CU0 to 1 in order to support FPU.

4.2.2.11 Instructions

Caution

> The instructions "trap 16" … “trap 31” used for TRAP1 are exclusively used by the OS.

Caution

In SC3 / SC4 systems

> The instructions "trap 0" … “trap 15” used for TRAP0 are exclusively used by the OS.

> The instruction "syscall" is not supported and therefore shall not be used.

4.2.2.12 Exception and Interrupt Cause Address

Note

The exception and interrupt cause address from EIPC and FEPC is stored in register CTPC when unhandled EIINT, unhandled SYSCALL, MIP/MDP exception (SC3/SC4) or unhandled core exception is reported.

4.2.3 Power PC Family

4.2.3.1 Context

- R2
- R13-R31
- PID
- SP
- PC
- LR
- MSR
- INTC_CPR[0|1|2]
- SPEFSCR

4.2.3.2 Core Registers

- SPRG0, SPRG1
- SRR0, SRR1
- IVPR
- PIR

4.2.3.3 Interrupt Registers

- INTC_BCR
- INTC_CPR0 – INTC_CPR2
- INTC_IACKR0 – INTC_IACKR2
- INTC_EOIR0 – INTC_EOIR2
- INTC_SSCIRn
- INTC_PSRn

4.2.3.4 PIT Registers

- PIT_MCR
- PIT_LDVALn
- PIT_CVALn
- PIT_TCTRLn
- PIT_TFLGn

4.2.3.5 STM Registers

- STM_CR
- STM_CNT
- STM_CCRn
- STM_CIRn
- STM_CMPn

4.2.3.6 MPU Registers

Core MPU	System MPU
> CMPU_MAS0	> SMPU_CESR0
> CMPU_MAS1	> SMPU_RGDn_WRD0
> CMPU_MAS2	> SMPU_RGDn_WRD1
> CMPU_MAS3	> SMPU_RGDn_WRD2
> CMPU_MPU0CSR0	> SMPU_RGDn_WRD3
	> SMPU_RGDn_WRD4
	> SMPU_RGDn_WRD5

4.2.3.7 SEMA4 Registers

- SEMA42_GATE0

4.2.3.8 Power PC Special Characteristics

- The exception handler for Machine check is implemented by the OS
- The exception handler for Data Storage is implemented by the OS
- The exception handler for Instruction Storage is implemented by the OS
- The exception handler for External Input is implemented by the OS
- The exception handler for Program is implemented by the OS
- The exception handler for System call is implemented by the OS

Note

The register SPRG0 is exclusively used by the OS to hold the identifier of the current thread.

The register SPRG1 is exclusively used by the OS to hold the address of the INTC_CPR register.

The register SEMA42_GATE0 is exclusively used by the OS to provide mutual exclusion in multicore systems for spinlock handling.

Thus these registers must not be used otherwise.

Caution

Due to MPU granularity all start addresses of configured MPU regions for the SystemMPU must be a multiple of 32. The configured end addresses must be a multiple of 32 minus one byte.

MICROSAR OS programs the MPU to grant access to the memory region between start address and end address.

> Access to configured start address and end address itself is granted

Note
For the CoreMPU, no restrictions on start address and end address apply.

MICROSAR OS programs the MPU to grant access to the memory region between start address and end address.

> Access to configured start address and end address itself is granted

Note
All stack sizes shall be configured to be a multiple of 8

Caution
MICROSAR OS assumes that Power PC multi core derivatives are booted as a master / slave system (as described in 2.15.4).

Note
For System MPU regions only the format FMT1 is supported to setting up the SMPUx_RGDn_WORD2.

4.2.3.9 MSR Handling

MSR.SPV bit handling	MICROSAR OS sets the SPV bit to 1 upon start of a thread.
MSR.EE bit handling	MICROSAR OS sets the external interrupt enable bit to 0 for non-interruptible threads without TimingProtection supervision, and to 1 for interruptible or TimingProtection supervised threads.
MSR.PR bit handling	MICROSAR OS sets the problem state bit to 0 for trusted software parts and to 1 for non-trusted software parts.
MSR.ME bit handling	MICROSAR OS sets the machine check enable bit to 1. Asynchronous Machine Check interrupts are enabled.
MSR remaining bits handling	MICROSAR OS sets all other MSR bits to 0 upon start of a thread.

4.2.4 ARM Family

4.2.4.1 Cortex-R derivatives

	Generic Cortex-R	Traveo Family	UltraScale Family
Context Registers	> R4-R11 > PC > LR > SP > PSR		
	---	> IRQPLM	> ICCPMR
Core Registers	> SCTLR > TPIDRURO		
MPU Registers	> DRBAR > DRSR > DRACR > RGNR		
Bootrom Registers	---	> UNLOCK > CNFG > UNDEFINACT > SVCINACT > PABORTINACT > DABORTINACT	---
INTC Registers	---	> NMIST > IRQST > NMIVAn > IRQVAn > NMIPLn > IRQPLn > NMIS > NMIR > IRQSn > IRQRn > IRQCESn > IRQCERn > NMIHC > IRQHC > UNLOCK	> ICCICR > ICCBPR > ICCIAR > ICCEOIR > ICDDCR > ICDISRn > ICDISERn > ICDICERn > ICDISPRn > ICDICPRn > ICDIPRn > ICDIPTRn > ICDSGIR
FRT Registers	---	> TCDT > TCCS	---

		> TCCSC > TCCSS	
Output Compare Registers	---	> OCCP0, OCCP1 > OCS > OCSC > OCSS	---
TTC Registers	---	---	> Clock_Control > Counter_Control > Counter_Value > Interval_Counter > Match_1_Counter > Match_2_Counter > Match_3_Counter > Interrupt_Register > Interrupt_Enable > Event_Control_Timer > Event_Register

4.2.4.2 Cortex-A derivatives

	Generic Cortex-A	iMX6 Family
Context Registers	> R4-R11 > PC > LR > SP > PSR	
Core Registers	> SCTLR > VBAR	
INTC Registers	---	> ICCICR > ICCBPR > ICCIAR > ICCEOIR > ICDDCR > ICDISRn > ICDISERn > ICDICERn > ICDISPRn > ICDICPRn > ICDIPRn > ICCPMR > ICDIPTRn > ICDSGIR

4.2.4.3 ARM Special Characteristics

- The exception handler for Supervisor Call is implemented by the OS
- The exception handler for Undefined Instruction is implemented by the OS
- The exception handler for Prefetch Abort is implemented by the OS
- The exception handler for Data Abort is implemented by the OS

Caution
Due to MPU hardware restriction the sizes of MPU regions and stack sizes must be configured with power of 2 values.

Caution with UltraScale derivatives

The exception vector table of each core must be located in tightly coupled RAM memory at address 0x0.

Either the debugger or the startup code has to copy the exception vector table from ROM section “OS_EXCVEC_CORE<core ID>_CODE” to address 0x0.

During OS startup OS code assumes that the exception vector table has already been copied.

Caution

To avoid memory violations during startup phase, the startup code shall initialize the stack pointer to use the OS kernel stack.

e.g. usage of the following function within startup code

```
__asm void HwSpec_ChangeToKernelStackAndReturn(void)
{
  ldr r0, =OsCfg_Stack_KernelStacks
  ldr r0, [r0] /* r0 = OsCfg_Stack_KernelStacks[0] */
  ldr r0, [r0] /* r0 = OsCfg_Stack_KernelStacks[0]->StackRegionEnd */
  mov sp, r0
  bx lr
}
```

4.3 Memory Mapping Concept

MICROSAR 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.

4.3.1 Provided MemMap Section Specifers

MICROSAR 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 ZERO_INIT	BOOLEAN 8BIT 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

Table 4-1 Provided MemMap Section Specifiers

4.3.1.1 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.

4.3.1.2 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
OS_<Callout>_CODE	> Code of Tasks, ISRs and OS Hooks

Table 4-2 MemMap 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 Id>_CONST	> Interrupt vector table of one specific core
OS_EXCVEC_<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 4-3 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_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	
OS_SPINLOCK_<SpinlockName>_VAR_NOCACHE_NOINIT	User spinlocks. Only the OS must have access to them. Potentially they will be accessed from all cores
OS_SPINLOCK_<SpinlockName>_VAR_FAST_NOCACHE_NOINIT	
OS_STACK_<StackName>_VAR_NOINIT	OS stacks. Only the current running software has access to the stack. Software which runs on a foreign must not have access to it.

Table 4-4 MemMap Variable 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.

4.3.1.3 Access Rights to Variable Sections

The table shows the recommended access rights to the sections.

Section	Local Core Trusted	Local core non trusted	Foreign core trusted	Foreign core non trusted
OS_VAR_NOCACHE	RW	RO	RW	RO
OS_VAR_NOCACHE_NOINIT				
OS_VAR_FAST_NOCACHE				
OS_VAR_FAST_NOCACHE_NOINIT				
OS_CORE<Core Id>_VAR	RW	RO	RO	RO
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	RW	RO	RW	RO
OS_PUBLIC_CORE<Core Id>_VAR_FAST_NOINIT				
OS_NONAUTOSAR_CORE<Core Id>_VAR	RW	RO	RW	RO
OS_NONAUTOSAR_CORE<Core Id>_VAR_FAST				
OS_NONAUTOSAR_CORE<Core Id>_VAR_NOINIT				
OS_NONAUTOSAR_CORE<Core Id>_VAR_FAST_NOINIT				
OS_GLOBALSHARED_VAR	RW	RW	RW	RW
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				

Section	Local Core Trusted	Local core non trusted	Foreign core trusted	Foreign core non trusted
OS_<ApplicationName>_VAR	RW	RW	RW	RO
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				
OS_<Task/IsrName>_VAR	RW	RW	RW	RO
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	RW	RO	RW	RO
OS_BARRIER_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT				
OS_CORESTATUS_CORE<Core Id>_VAR_ NOCACHE_NOINIT	RW	RO	RW	RO
OS_CORESTATUS_CORE<Core Id>_VAR_FAST_NOCACHE_NOINIT				
OS_SPINLOCK_<SpinlockName>_VAR_NOCACHE_NOINIT	RW	RO	RW	RO
OS_SPINLOCK_<SpinlockName>_VAR_FAST_NOCACHE_NOINIT				

Table 4-5 Recommended Section Access Rights

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

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

4.3.2 Link Sections

Once variables have been put into OS sections (by usage of the section specifiers described in 4.3.1.1) the sections would have to be linked.

Therefore MICROSAR 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 4-6 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.

4.3.2.1 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_Tasking.lsl
```

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

4.3.2.2 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 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.

4.3.2.3 Selecting OS constants

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1
OS_LINK_CONST_KERNEL	_NEAR _FAR

Table 4-7 OS constants linker define group

Example

```
#define OS_LINK_CONST_KERNEL
#include Os_Link_Core0_Tasking.lsl
```

Selects all OS constants from Os_Link_Core0_Tasking.lsl file

Example

```
#define OS_LINK_CONST_KERNEL_NEAR
#include Os_Link_Core0_Tasking.lsl
```

Selects all near addressable OS constants only from Os_Link_Core0_Tasking.lsl file

4.3.2.4 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 4-8 OS variables linker define group

Example

```
#define OS_LINK_VAR_KERNEL
#include Os_Link_Core0_Tasking.lsl
```

Selects all OS variables from Os_Link_Core0_Tasking.lsl file

Example

```
#define OS_LINK_VAR_KERNEL_NEAR_CACHE
#include Os_Link_Core0_Tasking.lsl
```

Selects all OS variables from Os_Link_Core0_Tasking.lsl file which are near addressable and cacheable

4.3.2.5 Selecting OS Barriers, Core Status and Trace 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 4-9 OS Barriers and Core status linker define group

Example

```
#define OS_LINK_KERNEL_BARRIERS
#include Os_Link_Core0_Tasking.lsl
```

Selects all OS Barriers from Os_Link_Core0_Tasking.lsl file

Example

```
#define OS_LINK_KERNEL_CORESTATUS
#include Os_Link_Core0_Tasking.lsl
```

Selects all OS core state variables from Os_Link_Core0_Tasking.lsl file

4.3.2.6 Selecting OS Spinlocks

These are hierarchical linker defines

Prefix	Optional Hierarchy level 1
OS_LINK_SPINLOCKS	_NEAR _FAR

Example

```
#define OS_LINK_SPINLOCKS
#include Os_Link_Tasking.lsl
```

Selects all OS core state variables from Os_Link_Tasking.lsl file

4.3.2.7 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 4-10 User constants linker define group

Example

```
#define OS_LINK_CONST_APP_<ApplicationName>
#include Os_Link_Core0_Tasking.lsl
```

Selects all constants from Os_Link_Core0_Tasking.lsl file which belong to the OS application <ApplicationName>

Example

```
#define OS_LINK_CONST_ISR_<ISRName>_FAR
#include Os_Link_Core0_Tasking.lsl
```

Selects all constants from Os_Link_Core0_Tasking.lsl file which belong to the ISR <ISRName> which have far addressing

4.3.2.8 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 4-11 User variables linker define group

Example

```
#define OS_LINK_VAR_APP_<ApplicationName>
#include Os_Link_Core0_Tasking.lsl
```

Selects all variables from Os_Link_Core0_Tasking.lsl file which belong to the OS application <ApplicationName>

Example

```
#define OS_LINK_VAR_APP_<ApplicationName>_FAR_CACHE_INIT
#include Os_Link_Core0_Tasking.lsl
```

Selects all variables from Os_Link_Core0_Tasking.lsl file which belong to the OS application <ApplicationName> which have far addressing, are cacheable and are initialized

4.3.3 Section Symbols

The linker command files described in 4.3.2 also generate section start and stop symbols which may be used to configure start and end addresses of MPU region objects or access check region objects.

These have the syntax

OS_<SectionType>_START

OS_<SectionType>_END

Example

OS_MyAppl_VAR_FAST_START

OS_ MyAppl_VAR_FAST_END

4.4 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.

4.5 Configuration of X-Signals

This chapter describes how X-Signals are configured for cross core API calls.

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 chapters for details on your used platform). 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 chapters for details on your used platform).
7. Add the "OsIsrXSignalReceiver" to the receiver ISR and select the provided APIs (callable from the sender core) with the "OsIsrXSignalReceiverProvidedApis" attribute.

Note

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

4.5.1 TriCore Aurix Family

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. The offset of the SRC register of the used interrupt source has to be specified for OsIsrInterruptSource.

4.5.2 RH850 Family

Logical Priority	A low number for OsIsrInterruptPriority attribute means a high 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	Only interrupt source 0 is possible.

4.5.3 Power PC Family

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 of the available software interrupts may be used.

4.5.4 ARM Family

	NVIC Interrupt Controller	GIC Interrupt Controller
Logical Priority	A low number for OsIsrInterruptPriority attribute means a high 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.	The interrupt sources 0..15 have to be used for the X-Signal ISR.

4.5.5 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.

4.6 OS generated objects

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

4.6.1 System Application

Type	OsApplication
Name	SystemApplication_<Core Id>
Condition	Is added when the core <Core Id> is configured to be an AUTOSAR core.
Features	> A system application contains the OS objects > Idle Task_<Core Id> > TpCounter_<Core Id> > XSignalIsr_<Core Id> > CounterIsr_TpCounter_<Core Id>

4.6.2 Idle Task

Type	OsTask
Name	IdleTask_<Core Id>
Condition	Is added when the core <Core Id> 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

4.6.3 Timer ISR

Type	OsIsr
Name	CounterIsr_<Core Id>
Condition	Is added if a hardware counter 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.

4.6.4 System Timer Counter

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

4.6.5 Timing Protection Counter

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

4.6.6 Timing protection ISR

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

4.6.7 Resource Scheduler

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

4.6.8 X Signal ISR

Type	OsIsr
Name	XSignalIsr_<Core Id>
Condition	Is added when an X signal channel is configured on the core.
Features	> Handles cross core requests.

4.6.9 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

4.7 VTT OS Specifics

4.7.1 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 modules (e.g. CAN) are inserted automatically by the respective BSW module.
- ➔ Other 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).

4.7.2 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 User include files

Within some features of MICROSAR 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!

5 API Description

This chapter lists all API service functions which are additionally provided by MICROSAR OS. AUTOSAR OS API service functions are not listed here. They can be looked up in [1] and [2].

5.1 Peripheral Access API

The API consists of read, write and bit manipulating functions for 8, 16 and 32 bit accesses.

5.1.1 Read Functions

Prototype

`FUNC(uint8, OS_CODE) Os_ReadPeripheral8(`
`  Os_PeripheralIdType PeripheralID,`
`  uint32 Address`
`)`

`FUNC(uint16, OS_CODE) Os_ReadPeripheral16(`
`  Os_PeripheralIdType PeripheralID,`
`  uint32 Address`
`)`

`FUNC(uint32, OS_CODE) Os_ReadPeripheral32(`
`  Os_PeripheralIdType PeripheralID,`
`  uint32 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 5-1 Read Peripheral API

Note
The former names of the API functions osReadPeripheral8(), osReadPeripheral16() and osReadPeripheral32() may also be used (the OS is backward compatible).

5.1.2 Write Functions

Prototype

`FUNC(void, OS_CODE) Os_WritePeripheral8(`
`  Os_PeripheralIdType PeripheralID,`
`  uint32 Address,`
`  uint8 Value`
`)`

`FUNC(void, OS_CODE) Os_WritePeripheral16(`
`  Os_PeripheralIdType PeripheralID,`
`  uint32 Address,`
`  uint16 Value`
`)`

`FUNC(void, OS_CODE) Os_WritePeripheral32(`
`  Os_PeripheralIdType PeripheralID,`
`  uint32 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 5-2 Write Peripheral APIs

Note
The former names of the API functions osWritePeripheral8(), osWritePeripheral16() and osWritePeripheral32() may also be used (the OS is backward compatible).

5.1.3 Bitmask Functions

Prototype	
`FUNC(void, OS_CODE) Os_ModifyPeripheral8( Os_PeripheralIdType PeripheralID, uint32 Address, uint8 ClearMask, uint8 SetMask )`	
`FUNC(void, OS_CODE) Os_ModifyPeripheral16( Os_PeripheralIdType PeripheralID, uint32 Address, uint16 ClearMask, uint16 SetMask )`	
`FUNC(void, OS_CODE) Os_ModifyPeripheral32( Os_PeripheralIdType PeripheralID, uint32 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 5-3 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 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 5-4 API Service Os_EnterPreStartTask

5.3 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 5-5 Call Non-Trusted Function API

5.4 Interrupt Source API

5.4.1 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 OS disables the interrupt source by modifying the interrupt controller registers.	
Particularities and Limitations	
> May be called for category 2 ISRs only.	

Table 5-6 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.

5.4.2 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)
Functional Description	
MICROSAR 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 5-7 API Service Os_EnableInterruptSource

5.4.3 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)
Functional Description	
MICROSAR OS clears the interrupt pending flag by modifying the interrupt controller registers.	
Particularities and Limitations	
> May be called for category 2 ISRs only	

Table 5-8 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.

5.4.4 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 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 5-9 API Service Os_IsInterruptSourceEnabled

5.4.5 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 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 5-10 API Service Os_IsInterruptPending

5.5 Detailed Error API

5.5.1 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 5-11 API Service Os_GetDetailedError

5.5.2 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.	

Table 5-12 API Service Os_GetUnhandledIrq

5.5.3 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.	

Table 5-13 API Service Os_GetUnhandledExc

5.6 Stack Usage API

All Service API functions which calculate stack usage are working in the same way.

- The service performs error checks:
  - 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()
- Returns the stack usage in bytes
- 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

Table 5-14 Overview: Stack Usage Functions

Caution
Any stack usage function must not be used cross core with interrupts disabled.

5.7 RTE Interrupt API

MICROSAR OS provides optimized interrupt en-/disable functions for exclusive usage for the RTE module of Vector.

API Name	Alias (for backward compatibility)
Os_DisableLevelAM()	osDisableLevelAM()
Os_DisableLevelKM()	osDisableLevelKM()
Os_DisableLevelUM()	osDisableLevelUM()
Os_EnableLevelAM()	osEnableLevelAM()
Os_EnableLevelKM()	osEnableLevelKM()
Os_EnableLevelUM()	osEnableLevelUM()
Os_DisableGlobalAM()	osDisableGlobalAM()
Os_DisableGlobalKM()	osDisableGlobalKM()
Os_DisableGlobalUM()	osDisableGlobalUM()
Os_EnableGlobalAM()	osEnableGlobalAM()
Os_EnableGlobalKM()	osEnableGlobalKM()
Os_EnableGlobalUM()	osEnableGlobalUM()

Caution
RTE interrupt handling functions should not be used by the application and are listed here to avoid naming collisions.

5.8 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).

5.8.1 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 5-15 Conversion Macros from Time to Counter Ticks

5.8.2 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 5-16 Conversion Macros from Counter Ticks to Time

5.9 Access Check API

5.9.1 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
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 5-17 API Service CheckISRMemoryAccess

5.9.2 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
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 5-18 API Service CheckTaskMemoryAccess

5.10 OS Initialization

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.

Table 5-19 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 5-20 API Service Os_InitMemory

5.11 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

5.11.1 Timing Hooks for Activation

5.11.1.1 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 TerminateTask() 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	
> none	

5.11.1.2 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. On single core systems both core IDs are always identical.	
Particularities and Limitations	
> none	

5.11.2 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`	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`	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_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.
`CallerCoreId`	Identifier of the core which performs the thread switch
Return code	
none	
Functional Description	
This hook is called on the caller core when that core in case it performs a thread switch (from one task or ISR to another task or ISR). On single core systems both core IDs are always identical.	
Particularities and Limitations	
> None	

5.11.3 Timing Hooks for Locking Purposes

5.11.3.1 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	

5.11.3.2 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	

5.11.3.3 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	

5.11.3.4 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	

5.11.3.5 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	

5.11.3.6 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	

5.11.3.7 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	

5.11.3.8 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	

5.11.3.9 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	

5.11.3.10 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	

5.12 PanicHook

Prototype
`FUNC(void, OS_PANICHOOK_CODE) Os_PanicHook(void)`
Parameter
none
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

5.13 Calling Context Overview

The following table gives an overview about the valid context for MICROSAR 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	X	X	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	

Table 5-21 Calling Context Overview

6 Configuration

MICROSAR 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 Glossary

Term	Description
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.
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).
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.
OS-application	An OS object of type application.
OS system level	The lowest priority of all category 1 ISRs.
X-Signal	MICROSAR OS mechanism which realizes cross core service APIs.
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).
Thread	Umbrella Term for OS Task, OS hooks and OS ISR objects

8 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com