# MICROSAR RTE

## Safety Guide

Version 4.8.0


| | |
|---|---|
| Authors | Sascha Sommer, Bernd Sigle |
| Status | Released |

# Document Information

## History

| Author | Date | Version | Remarks |
|---|---|---|---|
| 4.1.0 | 2013-04-15 | Sascha Sommer | Initial Creation for RTE 4.1 (AUTOSAR 4) |
| 4.2.0 | 2013-10-29 | Sascha Sommer<br>Bernd Sigle | Updated for RTE 4.2<br>Explained MICROSAR OS interrupt locking APIs.<br>Corrected review findings especially the used abbreviations. |
| 4.3.0 | 2014-02-05 | Sascha Sommer | Updated for RTE 4.3<br>Clarified Assumptions about VFB Trace Hooks<br>Described Inter-ECU sender/receiver from the ASIL partition<br>Support for mapped client/server calls between partitions<br>Multicore Support<br>SuspendAllInterrupts is no longer used |
| 4.4.0 | 2014-06-11 | Sascha Sommer | Updated for RTE 4.4 |
| 4.5.0 | 2014-10-15 | Bernd Sigle | Updated for RTE 4.5<br>Rte_DRead added |
| 4.6.0 | 2014-12-10 | Sascha Sommer | Updated for RTE 4.6 |
| 4.7.0 | 2015-03-18 | Sascha Sommer | Updated for RTE 4.7 |
| 4.8.0 | 2015-07-15 | Sascha Sommer | Updated for RTE 4.8<br>Described APIs/scheduling of ASIL BSW |

## Reference Documents

| No. | Source | Title | Version |
|---|---|---|---|
| [1] | AUTOSAR | AUTOSAR_SWS_RTE.pdf | 3.2.0 |
| [2] | AUTOSAR | AUTOSAR_SWS_OS.pdf | 5.0.0 |
| [3] | AUTOSAR | AUTOSAR_SWS_StandardTypes.pdf | 1.3.0 |
| [4] | AUTOSAR | AUTOSAR_SWS_PlatformTypes.pdf | 2.5.0 |
| [5] | AUTOSAR | AUTOSAR_SWS_CompilerAbstraction.pdf | 3.2.0 |

| | | | |
|---|---|---|---|
| [6] | AUTOSAR | AUTOSAR_SWS_MemoryMapping.pdf | |
| [7] | Vector | Technical Reference MICROSAR RTE | 4.8.0 |
| [8] | ISO | ISO/DIS 26262 | 2009 |

Scope of the Document

This document describes the use of the MICROSAR RTE with regards to functional safety. All general aspects of the MICROSAR RTE are described in a separate document [7], which is also part of the delivery.

**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 Purpose

The SEooC is developed based on assumptions on the intended functionality, use and context, including external interfaces. To have a complete safety case, the validity of these assumptions has to be checked in the context of the actual item after integration of the SEooC.
The application conditions for SEooC provide the assumptions made on the requirements (including safety requirements) that are placed on the SEooC by higher levels of design and also on the design external to the SEooC and the assumed safety requirements and assumptions related to the design of the SEooC.
The ASIL capability of this SEooC designates the capability of the SEooC to comply with assumed safety requirements assigned with the given ASIL.
Information given by this document helps to check if the SEooC does fulfil the item requirements, or if a change to the SEooC will be necessary in accordance with the requirements of ISO 26262.

The following document describes the SEooC MICROSAR RTE in the version 4.8.

# 2 Assumptions on the scope of the MICROSAR RTE

## 2.1 MICROSAR RTE overview

The MICROSAR RTE implements the AUTOSAR Standard of a Runtime Environment for AUTOSAR Software Components (SWCs) and Basic Software Modules (BSW). This means that the RTE is responsible for triggering the execution of SWC and BSW specific code in the form of runnable and schedulable entities. Moreover, the RTE provides APIs, for example for inter-ECU and intra-ECU communication and for exclusive area accesses. These APIs can be used by the runnable entities and BSW modules.
The MICROSAR RTE is a generic software component that is not tied to a specific item. Item specific functionality will be provided by the SWCs. The SWCs therefore also determine the ASIL that is required for the RTE. Consequently, the MICROSAR RTE can be seen as Safety Element out of Context (SEooC) according to ISO26262-10. This document provides the assumptions regarding the software safety requirements and the architectural design specification that were used for the development of the MICROSAR RTE. These assumptions have to be confirmed during item development.
The MICROSAR RTE is completely generated by the MICROSAR RTE Generator that is developed according to the established SPICE certified process (further referred to as QM).
If the generated code shall be used in an ASIL context, it has to be qualified according to the requirements of ISO 26262-6.
This document describes how the RTE configuration needs to look like so that it is in line with the safety assumptions and so that the complexity of the generated RTE code for ASIL SWCs is kept low enough to be reviewable for qualification. Review hints are provided in chapter 6.
The final integration of the RTE into a safety related item then needs to be done by a functional safety expert.

Please note that this document is an extension to the Technical Reference of the MICROSAR RTE with focus on safety related issues. Refer to the Technical Reference [7] for general topics like the RTE configuration, integration of the RTE into an ECU and a description of the RTE APIs.
An overall description of the RTE and AUTOSAR in general can be found in the AUTOSAR specifications.

**Caution**
The MICROSAR RTE Generator was not developed according to ISO26262. This document gives hints on what needs to be done in order to use the generated code within an item that is developed according to ISO26262.

Safety goals identified for the development result from the following hazard and risk list

| ID | Description of hazards that could occur |
|---|---|
| **H&R_RTE_1** | |
| **H&R_RTE_2** | |
| **H&R_RTE_3** | |
| | |
| | |

Table 2-1 Hazards

According to ISO26262, the hazard analysis and risk assessment shall be based on the item definition. As the development started with the unit design, the hazards have to be identified by the integrator for the specific item in which the RTE shall be integrated.

## 2.2 Standards and Legal requirements

The MICROSAR RTE Generator was developed according to the AUTOSAR RTE specification. The generated code can be qualified so that it can be used within an item that is developed according to ISO26262.

## 2.3 Functions of the MICROSAR RTE

The MICROSAR RTE provides the following functionality:

- AUTOSAR Runtime Environment according to [1] for QM SWCs and BSW:
  - communication between different runnables within the same SWC (explicit and implicit inter-runnable variables)
  - communication between different SWCs on the same ECU (queued and non-queued explicit and implicit sender/receiver communication, client/server communication, mode communication)
  - communication between SWCs and BSW modules located on the same ECU (queued and non-queued explicit and implicit sender/receiver communication, client/server communication, mode communication)
  - communication between SWCs on different ECUs (queued and non-queued explicit and implicit sender/receiver communication)
  - Calibration Parameters
  - Per-Instance Memories
  - Exclusive Areas

  Please see the RTE Technical Reference [7] for a full list of supported features.

- AUTOSAR Runtime Environment for ASIL SWCs and BSW when the generated RTE code is qualified according to the requirements of ISO26262 (Code description and configuration limitations described in this document):
  - Possibility to assign SWCs with different safety levels to distinct OS applications, so that a MPU can be used to provide freedom from interference with regards to memory
  - Support for Basic Tasks
  - Cyclic triggering of runnable entities
  - Cyclic triggering of schedulable entities
  - Per-Instance Memories
  - Explicit Inter-Runnable Variables
  - Explicit intra-ECU Sender/Receiver Communication with last-is best behaviour between SWCs with the same and different safety levels
  - Explicit inter-ECU Sender/Receiver Communication with last-is best behaviour
  - Direct Synchronous Client/Server calls inside the same OS application
  - Calibration Parameters
  - Explicit Exclusive Areas

- AUTOSAR Runtime Environment for ASIL SWCs when the generated RTE code is qualified according to the requirements of ISO26262 (Not handled in this document due to the many possible code variants):
  - Support for Extended Tasks
  - Init, Background, DataReceived, DataReceptionError, DataSendCompleted Triggers
  - Implicit Sender/Receiver communication
  - Queued Sender/Receiver communication
  - Synchronous and Asynchronous Client/Server calls to mapped server runnables in different OS applications

Table 2-2 summarizes the RTE features that are available for ASIL and QM SWCs.

| Feature | | QM SWCs in the BSW partition | SWCs separated from the BSW [1] | ASIL SWCs/BSW |
|---|---|---|---|---|
| Multicore Support | | ■ | ■ | ■ |
| Runnable Triggers | TimingEvent | ■ | ■ | ■ |
| | InitEvent | ■ | ■ | |
| | BackgroundEvent | ■ | ■ | |
| | DataReceivedEvent[4] | ■ | ■ | |
| | DataReceivedErrorEvent[4] | ■ | ■ | |
| | DataSendCompletedEvent | ■ | ■ | |
| | OperationInvokedEvent[4] | ■ | ■ | ■ |
| | AsynchronousServerCallReturnEvent[4] | ■ | ■ | |
| | ModeSwitchEvent[4] | ■ | | |
| | ModeSwitchAckEvent | ■ | | |
| SWC Settings | Source Code | ■ | ■ | ■ |
| | Object Code | ■ | ■ | |
| | Multiple Instantiation | ■ | ■ | |
| | Indirect API | ■ | ■ | |
| Runnable Settings | Minimum Start Interval | ■ | ■ | |
| Task Settings | Basic Tasks | ■ | ■ | ■ |
| | Extended Tasks | ■ | ■ | |
| Calibration Support | Rte_CData API | ■ | ■ | ■ |
| | Rte_Prm API | ■ | ■ | ■ |
| | Online Calibration | ■ | | |
| Per-Instance Memories | Rte_Pim API | ■ | ■ | ■ |
| Inter-Runnable Variables | Rte_IrvWrite API | ■ | ■ | ■ |
| | Rte_IrvRead API | ■ | ■ | ■ |
| | Rte_IrvIWrite API | ■ | ■ | |
| | Rte_IrvIRead API | ■ | ■ | |
| Sender/Receiver Communication | Rte_Write API | ■ | ■ | ■ |
| | Rte_Invalidate API | ■ | ■ | |
| | Rte_Read API | ■ | ■ | ■ |

[1] SWCs can either be assigned to the same partition as the BSW (recommended for QM SWCs, see column 1), or one or more separate partitions can be created. In this case, no features that require special handling when the RTE is initialized by the BSW and no features that require direct access to the BSW can be used. The marked features can be used for QM and ASIL SWCs but only the APIs marked in the column “ASIL SWCs” are described in this document.

| Feature | | QM SWCs in the BSW partition | SWCs separated from the BSW [1] | ASIL SWCs/BSW |
|---|---|---|---|---|
| | Rte_DRead API | ■ | ■ | |
| | Rte_IWrite API | ■ | ■ | |
| | Rte_IWriteRef API | ■ | ■ | |
| | Rte_IInvalidate API | ■ | ■ | |
| | Rte_IRead API | ■ | ■ | |
| | Rte_IStatus API | ■ | ■ | |
| | Rte_Feedback API | ■ | ■ | |
| | Rte_IsUpdated API | ■ | ■ | ■ |
| | Rte_NeverReceived API[4] | ■ | ■ | |
| | Rte_Send API[2] | ■ | ■ | |
| | Rte_Receive API[2] | ■ | ■ | |
| | inter-ECU communication | ■ | ■ | ■ |
| | intra-ECU communication | ■ | ■ | ■ |
| | Unconnected Ports | ■ | ■ | |
| Client/Server Communication | transmission acknowledgement | ■ | ■ | |
| | alive timeout | ■ | ■ | |
| | rx filters[4] | ■ | ■ | |
| | Rte_Call API | ■ | ■ | ■ |
| | Rte_Result API | ■ | ■ | |
| | synchronous calls to unmapped runnables[3] | ■ | ■ | ■ |
| | synchronous calls to mapped runnables on same task | ■ | ■ | ■ |
| | synchronous calls to runnables on different tasks[2] | ■ | ■ | |
| | asynchronous calls | ■ | ■ | |
| Mode communication | Rte_Switch API | ■ | | |
| | Rte_Mode API[4] | ■ | ■ | ■ |
| | Enhanced Rte_Mode API[4] | ■ | ■ | |
| | Rte_SwitchAck API | ■ | | |

[2] Only supported when all senders/callers are within the same partition. The servers/receivers can be in a different partition.

[3] Please note that this might not be possible when the server runnable is located in a different OS Application as the server is executed with the access rights of the caller. Also no additional protection measures are applied when the communication is between SWCs with different safety level.

[4] Please note that this feature is not possible for QM SWCs when the sender is an ASIL SWC

| Feature | | QM SWCs in the BSW partition | SWCs separated from the BSW [1] | ASIL SWCs/BSW |
|---|---|---|---|---|
| Exclusive Areas | mode switch acknowledgement | ■ | | |
| | mode disablings[4] | ■ | | |
| | implicit exclusive areas | ■ | ■ | |
| | explicit exclusive areas | ■ | ■ | ■ |
| | Rte_Enter API | ■ | ■ | ■ |
| | Rte_Exit API | ■ | ■ | ■ |
| | Implementation Method All InterruptBlocking | ■ | ■ | |
| | Implementation Method OS InterruptBlocking | ■ | ■ | ■ |
| | Implementation Method OsResources | ■ | ■ | |
| BSW Module Support (SchM) | | | | |
| | Implementation Method CooperativeRunnablePlacement | ■ | ■ | |
| | TimingEvent | ■ | ■ | ■ |
| | Background Event | ■ | ■ | |
| | explicit exclusive areas | ■ | ■ | ■ |
| | SchM_Enter API | ■ | ■ | ■ |
| | SchM_Exit API | ■ | ■ | ■ |
| | EA Implementation Method All InterruptBlocking | ■ | ■ | ■ |
| | EA Implementation Method OS InterruptBlocking | ■ | ■ | ■ |

Table 2-2 RTE features for ASIL and QM SWCs

## 2.4 Operating conditions

The MICROSAR RTE is part of the MICROSAR Safe Architecture (Figure 2-1).

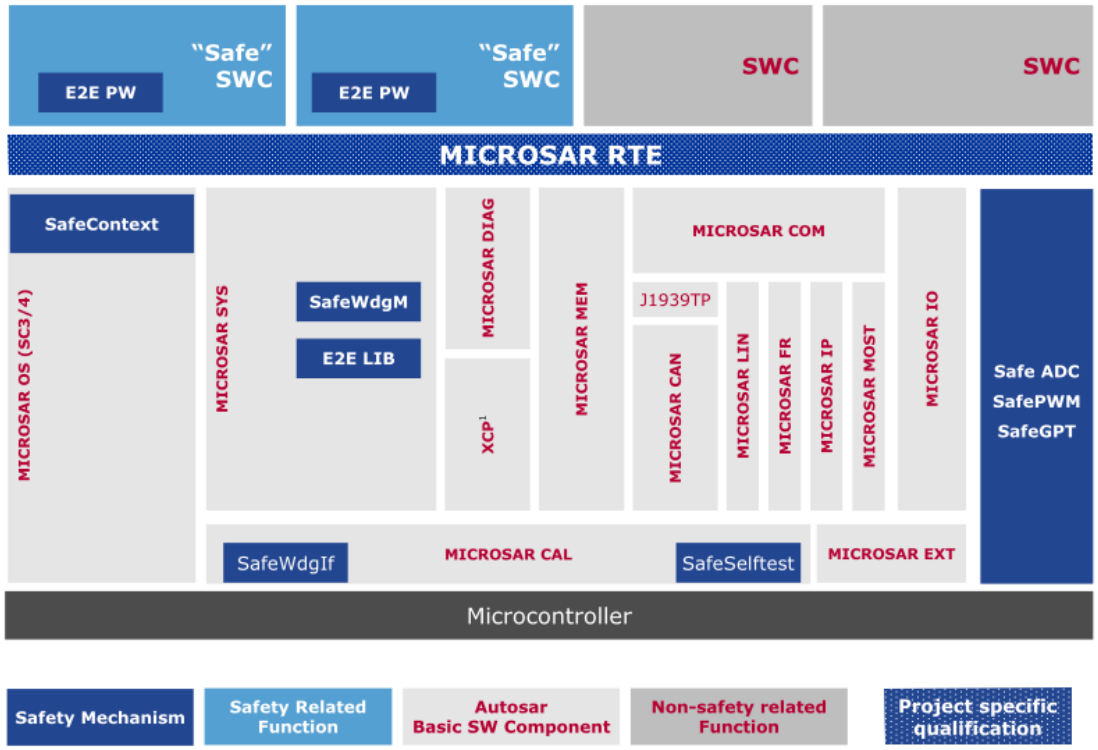


Figure 2-1 MICROSAR Safe Architecture

This architecture is based on the MICROSAR AUTOSAR stack developed with Vector's ISO9001 and SPICE based standard quality management.

The Add-On MICROSAR Safe Context extends this stack with an Operating System with memory protection in order to use the MICROSAR BSW with application software with a safety integrity level up to ASIL D.

ASIL Decomposition in the MICROSAR Safe Architecture is implemented through software partitioning as described in ISO 26262 (Figure 2-2).

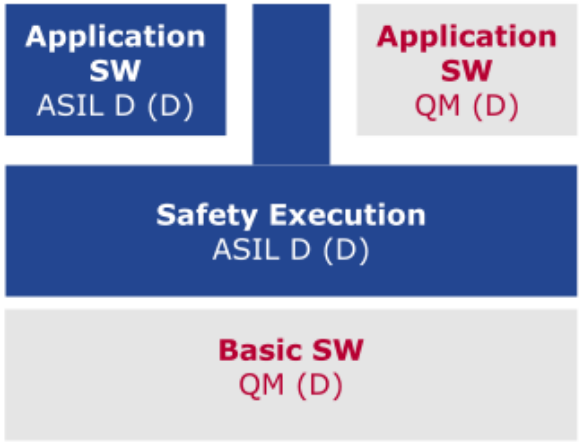


Figure 2-2 ASIL Decomposition

This means MICROSAR Safe Context provides freedom from interference with regards to memory so that the QM parts of the software cannot overwrite memory from the ASIL application. An additional Safe Watchdog module provides freedom from interference with regards to CPU runtime.

For Safe Communication between different ECUs, the AUTOSAR E2E Library can be used.

The MICROSAR RTE extends the MICROSAR Safe Context concept to the software component level. SWCs inherit the ASIL from the safety requirements that are allocated to them. With the MICROSAR RTE, it is possible to use SWCs with different ASIL as well as QM SWCs and BSW within the same ECU. The MICROSAR RTE furthermore provides communication mechanisms that can be used to implement communication between ASIL and QM SWCs.

A list of assumptions regarding the overall system architecture and development process is given in the next chapter.

## 2.5 Assumptions

| ID | Description of assumption on the scope of the MICROSAR RTE |
|---|---|
| **ASS_RTE_1** | The OS provides freedom from interference for different OS Applications with regards to memory. This means that code in one OS Application cannot destroy memory in another OS Application. |
| **ASS_RTE_2** | The AUTOSAR Memory Abstraction for the target platform and the OS make it possible to assign RTE/BSW variables to specific OS Applications so that they can only be written by code that is executed within this OS Application.<br>Moreover in case of Multicore, the RTE variables are mapped to noncacheable RAM so that they can be accessed by all cores. |
| **ASS_RTE_3** | The tool chain initializes global variables or the API Rte_InitMemory is called before the OS is started. Rte_InitMemory initializes variables from different OS Applications. Therefore it needs to be started without memory protection. |
| **ASS_RTE_4** | The OS allows non protected reads to RTE/BSW variables within the same and foreign OS Applications. |
| **ASS_RTE_5** | Freedom from interference with regards to CPU runtime is provided through external means, for example with the help of a control flow monitor. The mechanisms for it are either implemented in a way that the RTE cannot deactivate them or a review is performed that checks that the RTE does not impact their operation. |
| **ASS_RTE_6** | The OS APIs that are used by the RTE in ASIL parts of the code can be called from different contexts without interference:<br>> TerminateTask<br>> SuspendOSInterrupts |

| | |
|---|---|
| | > osDisableLevelUM (MICROSAR OS)<br>> osDisableLevelKM (MICROSAR OS)<br>> osDisableLevelAM (MICROSAR OS)<br>> osDisableGlobalUM (MICROSAR OS)<br>> osDisableGlobalKM (MICROSAR OS)<br>> osDisableGlobalAM (MICROSAR OS)<br>> ResumeOSInterrupts<br>> osRteEnableLevelUM (MICROSAR OS)<br>> osRteEnableLevelKM (MICROSAR OS)<br>> osRteEnableLevelAM (MICROSAR OS)<br>> osRteEnableGlobalUM (MICROSAR OS)<br>> osRteEnableGlobalKM (MICROSAR OS)<br>> osRteEnableGlobalAM (MICROSAR OS)<br>> GetSpinlock (Multicore Systems)<br>> ReleaseSpinlock (Multicore Systems) |
| **ASS_RTE_7** | The OS provides at least the APIs<br>> SuspendOSInterrupts<br>> osDisableLevelUM (MICROSAR OS)<br>> osDisableLevelKM (MICROSAR OS)<br>> osDisableLevelAM (MICROSAR OS)<br>> osDisableGlobalUM (MICROSAR OS)<br>> osDisableGlobalKM (MICROSAR OS)<br>> osDisableGlobalAM (MICROSAR OS)<br>> ResumeOSInterrupts<br>> osRteEnableLevelUM (MICROSAR OS)<br>> osRteEnableLevelKM (MICROSAR OS)<br>> osRteEnableLevelAM (MICROSAR OS)<br>> osRteEnableGlobalUM (MICROSAR OS)<br>> osRteEnableGlobalKM (MICROSAR OS)<br>> osRteEnableGlobalAM (MICROSAR OS)<br>with the same or higher ASIL than the SWCs |

| | |
|---|---|
| | In Multicore Systems, the OS also needs to provide the APIs<br>> GetSpinlock<br>> ReleaseSpinlock<br>with the same or higher ASIL than the SWCs |
| **ASS_RTE_8** | The RTE configuration is chosen in such a way that the OS/System mechanisms for freedom from interference (memory and runtime) can also be used to implement freedom from interference for SWCs with different ASIL. This makes it necessary to map SWCs with different ASIL to different OS Applications. All OS Applications with SWCs that do not have the highest ASIL need to be nontrusted. This includes the OS Application of the BSW. See also chapter 5.2. |
| **ASS_RTE_9** | The RTE configuration is chosen in such a way that no OS APIs need to be called in the RTE APIs or the TASK bodies that violate the safety requirements of the ASIL SWCs.<br>The RTE code calls the following OS APIs:<br>> SetRelAlarm<br>> CancelAlarm<br>> SetEvent<br>> GetEvent<br>> ClearEvent<br>> WaitEvent<br>> GetTaskID<br>> ActivateTask<br>> TerminateTask<br>> Schedule<br>> ChainTask<br>> GetResource<br>> ReleaseResource<br>In case of multicore systems also the API<br>> GetCoreID<br>is called. |
| **ASS_RTE_10** | The RTE configuration is chosen in such a way that no SWC needs to directly call methods in (Service-) SWCs with lower ASIL and no (Service-) SWCs with lower ASIL needs to call methods in ASIL SWCs except for the case when the SWCs explicitly allow this kind of usage. If necessary, this work is delegated to wrapper SWCs in the same OS Application as the called/calling SWC. Direct calls can moreover be avoided when the server runnables are mapped to tasks. See also chapter 5.2. |
| **ASS_RTE_11** | The RTE configuration is chosen in such a way that the RTE APIs or |

| | |
|---|---|
| | TASKS for a SWC do not contain calls to BSW modules with lower ASIL than the SWC itself that might cause interference. If necessary, this work is delegated to wrapper SWCs in the same OS Application as the BSW modules. For external communication, the RTE proxies the calls to the Com module. See also chapter 5.2. |
| **ASS_RTE_12** | The RTE does not need to provide freedom from interference for communication. In an AUTOSAR system, the E2ELibrary that is directly called by the SWCs is responsible for Safe communication. Nevertheless, the RTE provides APIs that can be called by the E2ELibrary. |
| **ASS_RTE_13** | The Generated RTE code for ASIL SWCs is qualified according to the requirements of ISO26262 by the integrator so that it reaches the same ASIL as the SWCs themselves. This is necessary because the RTE Generator was only developed with Vectors standard quality management (QM). |
| **ASS_RTE_14** | The hardware is suited for safety relevant software according to the requirements of ISO26262. The hardware requirements are mostly determined by the SWCs that shall be supported by the RTE. The MICROSAR RTE does not impose other hardware safety requirements as those that are already required by the SWCs and the OS. |
| **ASS_RTE_15** | The development tool chain (for example editors, compilers, linkers, make environment, flash utilities) is suited for the development of safety relevant software according to the requirements of ISO26262. All tools need to reach the appropriate Tool Qualification Level (TCL). |

Table 2-3 Assumptions regarding the system architecture and environment

# 3 Assumptions on the safety goals of the MICROSAR RTE

| ID | ASIL | Description of hazards that could occur | Ref assumption | Ref H & R |
|---|---|---|---|---|
| **SG_RTE_1** | | | | |
| **SG_RTE_2** | | | | |
| | | | | |
| | | | | |
| | | | | |

Table 3-1 Safety Goals

According to ISO26262, safety goals are determined for each hazardous event evaluated in the hazard analysis. As the hazard analysis could not be done due to the unknown target item, the safety goals have to be identified by the integrator once the hazard analysis is done.

| ID | Description of safe state | Ref safety goal |
|---|---|---|
| **SS_RTE_1** | | |
| **SS_RTE_2** | | |
| | | |
| | | |
| | | |

Table 3-2 Safe States

Due to the missing safety goals, no safe states could be identified that can be used to achieve a safety goal. The safe states have to be identified by the integrator once the safety goals are known.

# 4 Safety concept of the MICROSAR RTE

## 4.1 Functional concept

The MICROSAR RTE was developed with the following assumptions regarding the safety requirements. No references to safety goals and target ASIL are listed for the requirements as these depend on the particular context into which the RTE is integrated.

| ID | Description of safety requirement | ASIL | Ref SG, ASS |
|---|---|---|---|
| **SR_RTE_1** | The RTE (with the help of a MPU and an appropriate OS) shall provide freedom from interference with regards to memory for ASIL SWCs. This means that the RTE shall protect ASIL SWCs from BSW with lower ASIL. Additionally, the RTE shall also protect ASIL SWCs also from other SWCs with lower ASIL. The protection shall include the inter-runnable variables, per instance memories, sender buffers and stacks of the SWCs. Moreover, the protection shall be transparent to the SWCs, e.g. it shall be possible to access the protected inter-runnable variables and per-instance memories with the default AUTOSAR RTE APIs Rte_IrvWrite, Rte_IrvRead and Rte_Pim. | | |
| **SR_RTE_2** | The RTE shall provide intra-ECU communication mechanism for SWCs with the same and different ASIL. The communication shall be possible through the AUTOSAR RTE APIs Rte_Read and Rte_Write that are used for non-queued sender/receiver communication. | | |
| **SR_RTE_3** | The RTE shall provide mechanisms for data consistency that can be used by the ASIL SWCs to prevent concurrent accesses to shared ressources. (explicit exclusive areas). The realization of the exclusive areas shall be possible through the AUTOSAR RTE APIs Rte_Enter and Rte_Exit. | | |
| **SR_RTE_4** | The RTE shall provide access to calibration parameters for the ASIL SWCs. It shall be possible to access the calibration parameters with the default AUTOSAR RTE APIs Rte_Prm and Rte_CData. | | |

Table 4-1 Safety Requirements

## 4.2 Safe state and degradation concept

The MICROSAR RTE does not support degradation to the safe state as the safe state depends on the functionality of the item. Safe state degradation therefore also has to be implemented by the application or by the OS. Memory protection faults are supposed to be handled by the OS.

## 4.3 Fault tolerance and diagnostics concept

The fault tolerance and diagnostics concept depends on the requirements of the Application SWCs. It has to be implemented within the application SWCs.

# 5 Integration of the MICROSAR RTE in a new particular context

## 5.1 Assumptions

| ID | ASIL | Description of assumptions, safety goals, safety requirements | Validity check |
|---|---|---|---|
| **ASS_RTE_1** | | The OS provides freedom from interference for different OS Applications with regards to memory. This means that code in one OS Application cannot destroy memory in another OS Application. | |
| **ASS_RTE_2** | | The AUTOSAR Memory Abstraction for the target platform and the OS make it possible to assign RTE/BSW variables to specific OS Applications so that they can only be written by code that is executed within this OS Application.<br>Moreover in case of Multicore, the RTE variables are mapped to noncacheable RAM so that they can be accessed by all cores. | |
| **ASS_RTE_3** | | The tool chain initializes global variables or the API Rte_InitMemory is called before the OS is started. Rte_InitMemory initializes variables from different OS Applications. Therefore it needs to be started without memory protection. | |
| **ASS_RTE_4** | | The OS allows non protected reads to RTE/BSW variables within the same and foreign OS Applications. | |
| **ASS_RTE_5** | | Freedom from interference with regards to CPU runtime is provided through external means, for example with the help of a control flow monitor. The mechanisms for it are either implemented in a way that the RTE cannot deactivate them or a review is performed that checks that the RTE does not impact their operation. | |
| **ASS_RTE_6** | | The OS APIs that are used by the RTE in ASIL parts of the code can be called from different contexts without interference:<br>> TerminateTask<br>> SuspendOSInterrupts<br>> osDisableLevelUM (MICROSAR OS)<br>> osDisableLevelKM (MICROSAR OS)<br>> osDisableLevelAM (MICROSAR OS)<br>> osDisableGlobalUM (MICROSAR OS)<br>> osDisableGlobalKM (MICROSAR OS)<br>> osDisableGlobalAM (MICROSAR OS) | |

| | | | |
|---|---|---|---|
| | | > ResumeOSInterrupts<br>> osRteEnableLevelUM (MICROSAR OS)<br>> osRteEnableLevelKM (MICROSAR OS)<br>> osRteEnableLevelAM (MICROSAR OS)<br>> osRteEnableGlobalUM (MICROSAR OS)<br>> osRteEnableGlobalKM (MICROSAR OS)<br>> osRteEnableGlobalAM (MICROSAR OS)<br>> GetSpinlock (Multicore Systems)<br>> ReleaseSpinlock (Multicore Systems) | |
| **ASS_RTE_7** | | The OS provides at least the APIs<br>> SuspendOSInterrupts<br>> osDisableLevelUM (MICROSAR OS)<br>> osDisableLevelKM (MICROSAR OS)<br>> osDisableLevelAM (MICROSAR OS)<br>> osDisableGlobalUM (MICROSAR OS)<br>> osDisableGlobalKM (MICROSAR OS)<br>> osDisableGlobalAM (MICROSAR OS)<br>> ResumeOSInterrupts<br>> osRteEnableLevelUM (MICROSAR OS)<br>> osRteEnableLevelKM (MICROSAR OS)<br>> osRteEnableLevelAM (MICROSAR OS)<br>> osRteEnableGlobalUM (MICROSAR OS)<br>> osRteEnableGlobalKM (MICROSAR OS)<br>> osRteEnableGlobalAM (MICROSAR OS)<br>with the same or higher ASIL than the SWCs<br>In Multicore Systems, the OS also needs to provide the APIs<br>> GetSpinlock<br>> ReleaseSpinlock<br>with the same or higher ASIL than the SWCs | |
| **ASS_RTE_8** | | The RTE configuration is chosen in such a way that the<br>OS/System mechanisms for freedom from interference (memory and runtime) can also be used | |

| | | | |
|---|---|---|---|
| | | to implement freedom from interference for SWCs with different ASIL. This makes it necessary to map SWCs with different ASIL to different OS Applications. All OS Applications with SWCs that do not have the highest ASIL need to be nontrusted. This includes the OS Application of the BSW. See also chapter 5.2. | |
| **ASS_RTE_9** | | The RTE configuration is chosen in such a way that no OS APIs need to be called in the RTE APIs or the TASK bodies that violate the safety requirements of the ASIL SWCs.<br>The RTE codes calls the following APIs:<br>> SetRelAlarm<br>> CancelAlarm<br>> SetEvent<br>> GetEvent<br>> ClearEvent<br>> WaitEvent<br>> GetTaskID<br>> ActivateTask<br>> TerminateTask<br>> Schedule<br>> ChainTask<br>> GetResource<br>> ReleaseResource<br>In case of multicore systems also the API<br>> GetCoreID<br>is called. | |
| **ASS_RTE_10** | | The RTE configuration is chosen in such a way that no SWC needs to directly call methods in (Service-) SWCs with lower ASIL and no (Service-) SWCs with lower ASIL needs to call methods in ASIL SWCs except for the case when the SWCs explicitly allow this kind of usage. If necessary, this work is delegated to wrapper SWCs in the same OS Application as the called/calling SWC. Direct calls can moreover be avoided when the server runnables are mapped to tasks. See also chapter 5.2. | |
| **ASS_RTE_11** | | The RTE configuration is chosen in such a way that the RTE APIs or TASKS for a SWC do not contain calls to BSW modules with lower ASIL than the SWC itself that might cause interference. If necessary, this work is delegated to wrapper SWCs in the same OS | |

| | | | |
|---|---|---|---|
| | | Application as the BSW modules. For external communication, the RTE proxies the calls to the Com module. See also chapter 5.2. | |
| **ASS_RTE_12** | | The RTE does not need to provide freedom from interference for communication. In an AUTOSAR system, the E2ELibrary that is directly called by the SWCs is responsible for Safe communication. Nevertheless, the RTE provides APIs that can be called by the E2ELibrary. | |
| **ASS_RTE_13** | | The Generated RTE code for ASIL SWCs is qualified according to the requirements of ISO26262 by the integrator so that it reaches the same ASIL as the SWCs themselves. This is necessary because the RTE Generator was only developed with Vectors standard quality management (QM). | |
| **ASS_RTE_14** | | The hardware is suited for safety relevant software according to the requirements of ISO26262. The hardware requirements are mostly determined by the SWCs that shall be supported by the RTE. The MICROSAR RTE does not impose other hardware safety requirements as those that are already required by the SWCs and the OS. | |
| **ASS_RTE_15** | | The development tool chain (for example editors, compilers, linkers, make environment, flash utilities) is suited for the development of safety relevant software according to the requirements of ISO26262. All tools need to reach the appropriate Tool Qualification Level (TCL). | |
| **SR_RTE_1** | | The RTE (with the help of a MPU and an appropriate OS) shall provide freedom from interference with regards to memory for ASIL SWCs. This means that the RTE shall protect ASIL SWCs from BSW with lower ASIL. Additionally, the RTE shall also protect ASIL SWCs also from other SWCs with lower ASIL. The protection shall include the inter-runnable variables, per instance memories, sender buffers and stacks of the SWCs. Moreover, the protection shall be transparent to the SWCs, e.g. it shall be possible to access the protected inter-runnable variables and per-instance memories with the default AUTOSAR RTE APIs Rte_IrvWrite, Rte_IrvRead and Rte_Pim. | |
| **SR_RTE_2** | | The RTE shall provide intra-ECU communication mechanism for SWCs with the same and different ASIL. The communication shall be possible through the AUTOSAR RTE APIs Rte_Read and Rte_Write that are used for non-queued sender/receiver communication. | |
| **SR_RTE_3** | | The RTE shall provide mechanisms for data consistency that can be used by the ASIL SWCs to prevent concurrent accesses to shared ressources. (explicit exclusive areas). The realization of the | |

| | | | |
|---|---|---|---|
| | | exclusive areas shall be possible through the AUTOSAR RTE APIs Rte_Enter and Rte_Exit | |
| **SR_RTE_4** | | The RTE shall provide access to calibration parameters for the ASIL SWCs. It shall be possible to access the calibration parameters with the default AUTOSAR RTE APIs Rte_Prm and Rte_CData. | |

Table 5-1 Assumptions that need to be verified during the integration

The MICROSAR RTE Generator does not produce ASIL code. However, the following chapter tries to explain how an RTE configuration needs to look like so that it is both, in line with the assumptions given in the previous table, and that it can be reviewed to reach compliance with a certain ASIL level.

## 5.2 RTE Configuration

During the software design, it has to be decided if the SWCs need to be developed with a certain ASIL. In the MICROSAR RTE, OS Applications are used to partition the SWCs of an ECU according to their ASIL. Therefore, for every used ASIL, at least one OS Application has to be created. Furthermore an OS Application for the QM BSW needs to be created. It can also be used for the QM SWCs. All OS Applications apart from the OS Applications with the highest ASIL need to be nontrusted. An example is given in Figure 5-1.

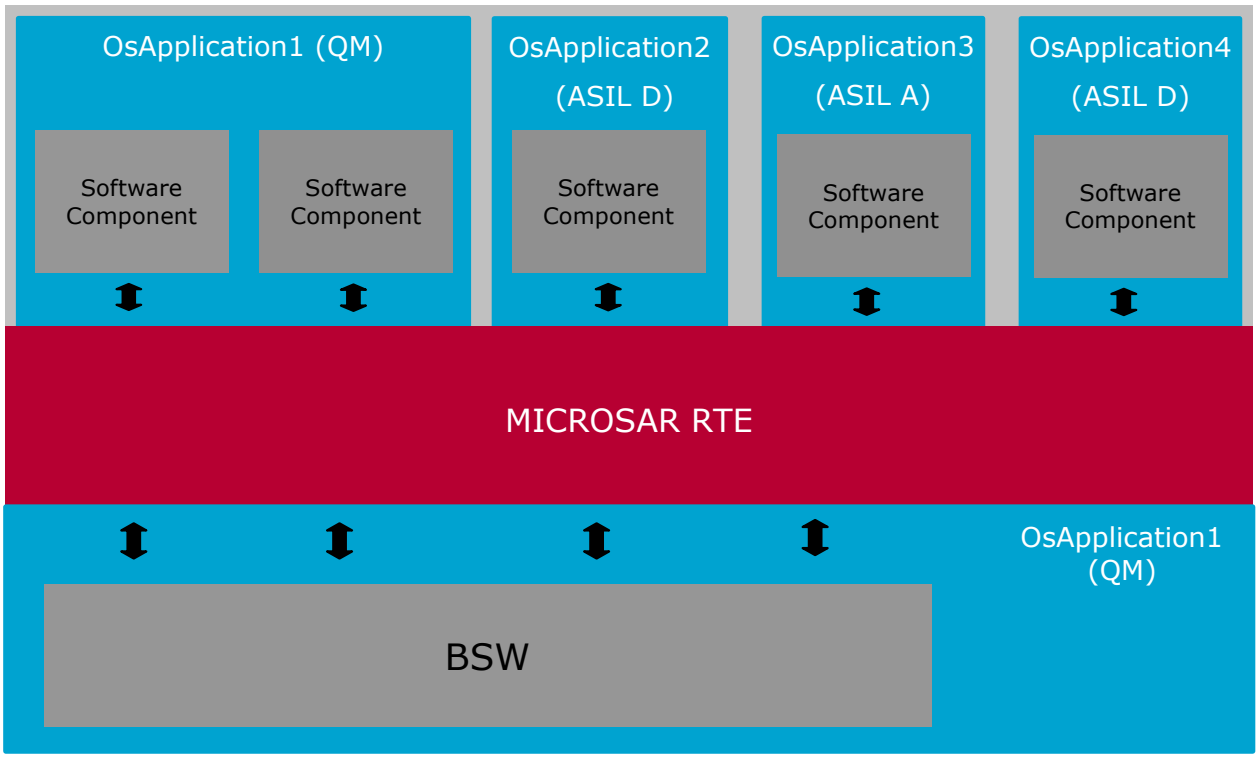


Figure 5-1 SWC to OsApplication Mapping

The assignment of the SWCs to the OS Applications happens through the task mapping.

That means the RTE tasks need to be assigned to the OS Applications. Every SWC then needs to be mapped to the appropriate OS Application by mapping its runnables to tasks of that OS Application.

As the BSW OS Application is nontrusted, the following memory protection limitations from the RTE Technical Reference apply for all SWCs:

- All schedulable entities of QM BSW Modules need to be assigned to the BSW OS Application
- All SWCs with mode provide ports need to be assigned to the BSW OS Application.
- All SWCs that contain runnables with mode disabling dependencies or mode triggers need to be assigned to the BSW OS Application.
- Direct client/server calls between OS Applications are not allowed. Exceptions are possible when the servers explicitly allow that they are run within the contexts of the client OS Applications. The RTE generator issues a warning when it detects direct client/server calls between OS Applications.

**Caution**
When a client directly calls a server in another OS Application, the server runnable will run within the OS Application of the client. This means it can access resources e.g. memory that are normally only supposed to be accessed by runnables in the client OS Application. Moreover the server is not able to access resources that can only be accessed by his own OS Application.
This might violate the safety requirements of the SWCs.

It is assumed that the MICROSAR RTE is used together with MICROSAR OS Safe Context. While most APIs of MICROSAR OS Safe Context can be called from arbitrary contexts without causing interference, only certain APIs are implemented in a way that ASIL code can rely on them. Therefore, the following RTE features that rely on ASIL OS functionality cannot be used in ASIL SWCs:

- Extended Tasks
- Minimum Start Interval
- Exclusive Areas with implementation methods other than Interrupt Blocking
- Alive timeout
- Triggering of runnables in other SWCs e.g. by OnDataReception, OnDataReceptionError, OnDataSendCompletion triggers of a sender/receiver port

The following general RTE feature cannot be used when ASIL SWCs are present because it requires calls to non ASIL BSW modules:

- Measurement with XcpEvents

To simplify the review, it is also recommended to only use a subset of the RTE features in the ASIL SWCs. The review chapter in this document only describe the RTE APIs for the case when the following features are not used:

- VFB Trace Hooks
- Invalidation
- Rx Filters
- Implicit Exclusive Areas
- Implicit Inter-Runnable Variables
- Multiple Instantiation
- Object Code SWCs
- Unconnected ports
- Implicit sender/receiver communication
- Online calibration
- Transmission acknowledgement
- Never Received API
- Enhanced Rte_Mode API
- Development Error Tracer (DET)
- Data Prototype Mappings

Summarized, ASIL SWCs may use the following RTE features without violating the RTE safety assumptions and with the goal in mind to have easy to review code:

- Runnables with cyclic triggers
- Runnables with OperationInvokedTriggers
- Basic tasks. (This means all runnables on an ASIL task need to share the same cycle time and offset.)
- Explicit Intra-ECU Sender/Receiver communication (Last-Is-Best)
- Explicit Inter-ECU Sender/Receiver communication (Last-Is-Best)
- Rte_IsUpdated API
- Synchronous Client/Server calls to unmapped runnables or runnables with CanBeInvokedConcurrently set to true. The client and server need to be mapped to the same OS Application. Exceptions are possible when the servers explicitly allow such usage.
- Explicit Inter-Runnable variables
- Mode require ports without mode triggers and mode disabling dependencies

- Explicit exclusive areas
- Per-instance memories
- SWC local Calibration parameters
- Calibration ports

Please note that the names of the objects in the RTE configuration are used in identifiers in the generated RTE and OS C code. The names have to be chosen in such a way that the identifiers do not exceed the limits of the target compiler and that they do not conflict with other identifiers from other modules.

Also the filenames of the generated files are created from object names in the RTE configuration. It also needs to be checked that the file names do not exceed the limits of the target compiler.

## 5.3 RTE Generation

Once the RTE is configured it can be generated with the MICROSAR RTE Generator.

The MICROSAR RTE Generator will run some checks prior to the generation.

Errors in the Configuration are reported with an [Error] prefix. The generator will abort the generation in this case. Warnings are reported with [Warning]. Every warning has to be checked and it needs to be assured that the warnings do not cause any harm.

Moreover after the generation, it has to be checked that the output directory contains no old files from previous generations. The RTE Generator provides a magic number pre-processor check at the end of the files that will issue a compile error when it detects an old file. Please note that selective file generation needs to be disabled in order to use the magic number check.

# 6 Qualification of generated RTE Code

The following section gives some hints on what needs to be verified when the RTE shall be used for ASIL SWCs. The API descriptions show how the RTE code is supposed to look like according to the generator design when the configuration is based on the description in chapter 5.2. If the generated code diverges from the code descriptions, the integrator has to verify that the differences do not cause any harm.

**Caution**
The MICROSAR RTE generator does not generate ASIL code. If the RTE code shall be used for ASIL SWCs, the generated code has to be qualified. ISO26262 lists various methods that can or have to be applied to reach a certain ASIL. The integrator has to decide which methods are suited for his project and take the required actions.

## 6.1 Introduction

As the generated RTE code heavily depends on the names of the objects from the RTE configuration, the API descriptions use the following placeholders:

<oa> OS Application

<soa> sender OS Application

<roa> receiver OS Application

<bswoa> BSW OS Application

<c> component type name

<bsw> BSW module name

<sc> sender component type name

<rc> receiver component type name

<ci> component instance name

<sci> sender component instance name

<rci> sender component instance name

<p> port prototype

<sp> sender port prototype

<rp> receiver port prototype

<d> data element prototype

<sd> sender data element prototype

<rd> sender data element prototype

<o> operation prototype

<re> runnable entity name

<res> runnable/schedulable entity symbol

<sre> server runnable entity name

<sres> server runnable entity symbol

<signalid> identifier for signal specific buffers

<nocache> _NOCACHE extension for variables that are accessed from multiple cores

<t> data type

<tp> pointer to data type

<name> per-instance memory, calibration parameter, exclusive area or inter-runnable variable name

<Lock> Interrupt Locking / Spinlock function as described in chapter 6.3.4

<UnLock> Interrupt Unlocking / Spinlock function as described in chapter 6.3.4

<Rte_MemCpy> Memory Copy function as described in chapter 6.9.1

Placeholders written in upper case, for example <P>, mean that the replacement string is written in upper case.

## 6.2 Compiler and Memory Abstraction

The RTE code uses the AUTOSAR compiler and memory abstraction for functions, function prototypes and variable declarations.

```
#define RTE_START_SEC_<secname>
#include "MemMap.h"

<Object0>
[<Object1>]
[<ObjectN>]

#define RTE_STOP_SEC_<secname>
#include "MemMap.h"
```

The memory abstraction is used to assign RTE variables to OS Applications. <secname> is the used memory section, for example CODE, VAR, CONST.
All variables are assigned to the OS Application in which they are written.

Section defines contain the name of the OS Application with the exception of the section defines for the OS Application that contains the BSW. It directly uses the section defines of the RTE.

In case of multicore systems, the memory abstraction defines use an additional _NOCACHE extension when variables are accessed from multiple cores. It has to be assured that variables with this extension are mapped to noncacheable RAM and that all variables that are accessed from multiple cores are mapped with this extension.

The RTE includes the file MemMap.h that has to issue the correct compiler pragmas for the platform so that variables within the memory abstraction are mapped to the correct protected memory sections. See the Technical Reference of the OS of how this can be accomplished. During integration it has to be assured that the mapping mechanisms function properly. The RTE section and compiler abstraction defines are described in the RTE Technical Reference.

Besides these, the MICROSAR RTE uses the following macros from the compiler abstraction:

- FUNC
- AUTOMATIC
- STATIC
- NULL_PTR
- FUNC_P2CONST
- P2VAR
- P2CONST
- CONST
- CONSTP2CONST
- P2FUNC
- VAR

Their functionality needs to be verified for correctness on the target platform.

## 6.3 DataTypes

### 6.3.1 Imported Types

The MICROSAR RTE imports the following types from Std_Types.h and the Platform_Types.h header that is included by Std_Types.h. It needs to be assured that they are mapped to the correct platform specific types:

- boolean
- uint8
- uint16
- uint32
- uint64
- sint8
- sint16
- sint32
- sint64
- float32

- float64
- uint8_least
- uint16_least
- uint32_least
- sint8_least
- sint16_least
- sint32_least
- Std_ReturnType

Furthermore the following imported defines need to be correct:

- STD_ON
- STD_OFF

### 6.3.2 Application Types Generated by the RTE

The RTE Generator generates the data types from the configuration to the header Rte_Type.h. It has to be checked that Rte_Type.h contains all configured data types and that the datatypes are in line with the configuration. The RTE generator only generates implementation data types.

Besides the name of the generated data type, also its properties have to be checked. For primitive types, this means that the upper and lower limit defines are identical to the ones that are specified in the configuration and that the base type that is used for the data type covers its full range. Upper and lower Limits are only generated to the file Rte_<c>_Type.h when a components uses the application data type that defines the limits.

For complex array types, it has to be checked that the base type is the same as the one that is configured in the configuration. Furthermore, the length of the array needs to be identical to the one from the configuration for array types.

For complex record types, the types, names and order of the contained elements needs to be the same as specified in the configuration.

For enumerations, all generated enumeration literals also need to be defined to the values that are specified in the configuration. It has to be checked that the list of literals is complete and that no literal conflicts with other identifiers. The literals are generated to Rte_<c>_Type.h when a component uses a datatype that references a compu method with a texttable.

### 6.3.3 Handling of Array and String Data Types

In the RTE APIs, arrays are passed as pointer to the array base type.

For simplicity, the code descriptions in the following chapters only show examples with primitive integer types.

### 6.3.4 Datatype specific handling of Interrupt Locks and Spinlocks

The RTE implements data consistency mechanisms in some of its APIs. Data consistency is provided with the SuspendOSInterrupts and ResumeOSInterrupts OS APIs on a single

core. As the calls to these OS APIs significantly increase the runtime of the RTE APIs, the RTE tries to optimize them away when they are not needed.

The APIs are optimized away when the variables can be written atomically by the ECU.

The behaviour is controlled by the setting of the parameter AtomicVariableAccess in the configuration of the EcuC module in the ECUC configuration file.

For simplicity, the code descriptions in the following chapters only show APIs in which the interrupt locks are not optimized away. When the RTE code for the ASIL SWCs is verified, it needs to be checked that the optimization is correct, e.g. that a variable can really be accessed atomically by the ECU.

Some versions of MICROSAR OS provide optimized interrupt locking APIs. The RTE will use these APIs when they are available.

For nontrusted OS Applications the RTE may call

- osDisableLevelUM
- osDisableLevelAM
- osDisableGlobalUM
- osDisableGlobalAM

to disable the interrupts and

- osEnableLevelUM
- osEnableLevelAM
- osEnableGlobalUM
- osEnableGlobalAM

to reenable the interrupts.

For trusted OS Applications the RTE may call

- osDisableLevelKM
- osDisableLevelAM
- osDisableGlobalKM
- osDisableGlobalAM

to disable the interrupts and

- osEnableLevelKM
- osEnableLevelAM
- osEnableGlobalKM
- osEnableGlobalAM

to reenable the interrupts.

The optimized APIs are mapped to the macros Rte_DisableOSInterrupts/Rte_DisableAllInterrupts and Rte_EnableOSInterrupts/Rte_EnableAllInterrupts.

In the following code examples, <Lock> resolves to SuspendOSInterrupts, or Rte_DisableOSInterrupts/Rte_DisableAllInterrupts. <UnLock> resolves to ResumeOSInterrupts or Rte_EnableOSInterrupts/Rte_EnableAllInterrupts. The RTE generator tries to use the “Disable” variant whenever possible as it is usually faster. It has to be assured that this variant is only used, when there are no nested calls to the locking APIs. They cannot be used when the runnable is configured to enter or to run in an exclusive area. For every locking operation the matching unlocking operation needs to be called.

It needs to be assured that no included header breaks these operation (e.g. by redefining them).

On multicore systems, locking the interrupts will only provide data consistency on the core for which the RTE API is called. In order to provide data consistency also when variables are accessed from different cores, the <Lock> and <UnLock> operations are extended with additional GetSpinlock and ReleaseSpinlock calls.

Example:

```
SuspendOSInterrupts();
(void)GetSpinlock(<SpinlockId>);

Access data structures.

(void)ReleaseSpinlock(<SpinlockId>);
ResumeOSInterrupts();
```

All places where the protected data structures are accessed need to be protected by the same Spinlock (same <SpinlockId>)

The interrupt lock APIs can be omitted when the spinlock cannot be accessed by multiple tasks on the same core.

Instead of SuspendOSInterrupts and ResumeOSInterrupts also Rte_DisableOSInterrupts/Rte_DisableAllInterrupts and Rte_EnableOSInterrupts/Rte_EnableAllInterrupts can be used, the distinction is the same as explained above.

## 6.4 SWC Implementation

The RTE is the module that glues the SWCs to the AUTOSAR stack. For this, the RTE generator generates a list of files that provide the datatypes and APIs for the SWCs and that call the runnable entities of the SWCs. A description of all generated files can be found in the RTE Technical Reference.

From the generated files, the SWCs shall only include the appropriate RTE header Rte_<c>.h directly. It provides the SWC specific functionality.

The SWC implementation shall at least contain all configured runnable entities.

The signature of the runnable entities is

```
FUNC(void, <c>_CODE) <res>(<parglist><arglist>)
```

or

```
FUNC(Std_ReturnType, <c>_CODE) <res>(<parglist><arglist>)
```

for server runnables with return type.

<arglist> is “void” for non-server runnables, otherwise it contains the arguments of the server operation.

<parglist> is empty for runnables without port defined arguments, otherwise it contains the port defined arguments.

```
# define <c>_START_SEC_CODE
# include "MemMap.h"

FUNC(void, <c>_CODE) <res>(<parglist><arglist>)
{

}

# define <c>_STOP_SEC_CODE
# include "MemMap.h"
```

Runnable entity implementations shall be surrounded by <c>_CODE memory abstraction defines as shown above.

Every runnable entity shall have a prototype in Rte_<c>.h that is also surrounded by the same memory abstraction defines.

Server runnables with return value shall return a value in all return paths.

The value shall be RTE_E_OK or one of the application return codes from the configuration. The application return code defines are contained in the file Rte_<c>.h. It needs to be checked that the return defines provide the same value as described in the configuration.

Runnable entities in ASIL SWC should only contain calls to the following RTE APIs:

- Std_ReturnType Rte_Write_<p>_<d>(<data>)
- Std_ReturnType Rte_Read_<p>_<d>(<data>)
- boolean Rte_IsUpdated_<p>_<d>()
- Std_ReturnType Rte_Call_<p>_<o>(<data_1>, <data_n>)
- <type> Rte_Pim_<name>()
- <return> Rte_CData_<name>()
- <return> Rte_Prm_<p>_<name>()
- <return> Rte_IrvRead_<re>_<name>()
- void Rte_IrvWrite_<re>_<name>(<data>)
- void Rte_Enter_<name>()
- void Rte_Exit_<name>()
- Rte_ModeType_<m> Rte_Mode_<p>_<d>()

The Rte_Write, Rte_Read, Rte_IsUpdated, Rte_Call, Rte_IrvRead, Rte_IrvWrite, Rte_Enter, Rte_Exit APIs are only allowed to be called from a runnable when the runnable is configured to access the port data element/port operation/inter-runnable variable/exclusive area for which the API is generated.

The Rte_CData and Rte_Pim APIs are only allowed to be called from runnables in the SWCs in which they are configured.

The Rte_Prm API is only allowed to be called from runnables in the SWC that contains the matching calibration receiver port.

For every API that is called by the SWC implementation, it needs to be checked, that the called API is configured for the SWC and that Rte_<c>.h declares the API.

Furthermore, it needs to be assured that RTE and OS variables are only modified with afore mentioned RTE API calls. The variables are not allowed to be modified directly within the runnable code.

It also has to be assured that the RTE APIs with parameters are called with the correct parameters with regards to type and access rights. When pointers are passed to the RTE APIs, it has to be assured that the pointers stay valid during the whole runtime of the RTE API and that the underlying objects are not modified outside the RTE API during the runtime of the RTE API.

All RTE APIs for a SWC are declared in the header Rte_<c>.h. APIs are implemented either as macro or as function. When the APIs are implemented as functions, the

implementation is contained in the file Rte_<oa>.c of the OS Application to which the SWC is mapped.

It needs to be assured that all functions that are called from within RTE code are imported from the correct versions of the AUTOSAR, Com and OS source and header files.

The inclusion of files from these and other modules shall not re-define any identifier that is defined in the generated RTE code, e.g. through #define macros. Exceptions are the RTE memory section defines that can be redeclared in MemMap. However, it needs to be checked that the mapping of the variables to the code sections works as expected.

## 6.5 BSW Implementation

The RTE is the module that glues the BSW to the AUTOSAR stack. For this, the RTE generator generates a list of files that provide the datatypes and APIs for the BSW and that call the schedulable entities of the BSW. A description of all generated files can be found in the RTE Technical Reference.

From the generated files, the BSW shall only include the appropriate RTE header SchM_<bsw>.h directly. It provides the BSW specific functionality.

The BSW implementation shall at least contain all configured schedulable entities.

The signature of the schedulable entities is

FUNC(void, <BSW>_CODE ) <res>()

Schedulable entity implementations shall be surrounded by <BSW>_CODE memory abstraction defines.

Every schedulable entity shall have a prototype in SchM_<bsw>.h that is also surrounded by the same memory abstraction defines.

Schedulable entities in ASIL BSW should only contain calls to the following RTE APIs:

- void SchM_Enter_<bsw>_<name>()
- void SchM_Exit_<bsw>_<name>()

All RTE APIs for a BSW module are declared in the header SchM_<bsw>.h. APIs are implemented either as macro or as function. When the APIs are implemented as functions, the implementation is contained in the file Rte_<oa>.c of the OS Application to which the BSW is mapped.

It needs to be assured that all functions that are called from within RTE code are imported from the correct versions of the AUTOSAR, Com and OS source and header files.

The inclusion of files from these and other modules shall not re-define any identifier that is defined in the generated RTE code, e.g. through #define macros. Exceptions are the RTE memory section defines that can be redeclared in MemMap. However, it needs to be checked that the mapping of the variables to the code sections works as expected.

## 6.6 SWC specific RTE APIs

### 6.6.1 Rte_Write

#### 6.6.1.1 Configuration Variant Intra-ECU Without IsUpdated

- source code SWC
- no support for multiple instantiation
- no indirect API
- intra-ECU communication
- receiver is connected
- no receiver triggered on data reception
- no receiver triggered on data reception error
- no transmission acknowledgement
- no rx filtering
- no data prototype mapping
- no invalidation
- is updated is not configured for the receivers
- never received is not configured for the receiver

#### 6.6.1.2 Generated Code Intra-ECU Without IsUpdated

Rte_<c>.h defines Rte_Write as follows:

```
#define Rte_Write_<p>_<d> Rte_Write_<c>_<p>_<d>
```

When the attribute “EnableTakeAddress” is not set for the port and when the data element can be accessed atomically by the ECU and when the data element is not an array or string type, Rte_Write_<c>_<p>_<d> is declared as macro that writes to a global RTE variable.

```
# define RTE_START_SEC_VAR_<oa><nocache>_INIT_UNSPECIFIED
# include "MemMap.h"
```

```
extern VAR(<t>, RTE_VAR_INIT<nocache>) Rte_<ci>_<p>_<d>;

# define RTE_STOP_SEC_VAR_<oa><nocache>_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_Write_<c>_<p>_<d>(data) (Rte_<ci>_<p>_<d> = (data),
   ((Std_ReturnType)RTE_E_OK))
```

Otherwise, the API is implemented in Rte_<oa>.c

```
#define RTE_START_SEC_CODE
#include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Write_<c>_<p>_<d>(<t> data)
{
  Std_ReturnType ret = RTE_E_OK;

  Rte_WriteHook_<c>_<p>_<d>_Start(data);
  <Lock>();
  Rte_<ci>_<p>_<d> = *(&data);
  <UnLock>();
  Rte_WriteHook_<c>_<p>_<d>_Return(data);
  return ret;
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

and Rte_<c>.h only contains the prototype:

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Write_<c>_<p>_<d>(<t> data);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"
```

In both cases the global variable in Rte_<oa>.c is declared as

```
#define RTE_START_SEC_VAR_<oa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
VAR(<dt>, RTE_VAR_INIT<nocache>) Rte_<ci>_<p>_<d> = <initializer>;
#define RTE_STOP_SEC_VAR_<oa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the port data element.

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to do the initialization:

```
Rte_<ci>_<p>_<d> = <initializer>;
```

If the data element is of a string or array type, no direct assignments are used. Instead, the assignment is replaced by a memcpy

```
<Rte_MemCpy>(Rte_<ci>_<p>_<d>, *(data), sizeof(<t>));
```

If the datatype can be read and written atomically, the <Lock>() and <UnLock>() calls are omitted from the Rte_Write API.

When Rte_Write is not a macro, it needs to be assured that the macros Rte_WriteHook_<c>_<p>_<d>_Start(data) and Rte_WriteHook_<c>_<p>_<d>_Return(data) do not have any side effects.

#### 6.6.1.3 Configuration Variant Intra-ECU With IsUpdated

- source code SWC
- no support for multiple instantiation
- no indirect API
- intra-ECU communication
- receiver is connected
- no receiver triggered on data reception
- no receiver triggered on data reception error
- no transmission acknowledgement
- no rx filtering
- no data prototype mapping
- no invalidation
- is updated is configured for one receiver
- never received is not configured for the receiver

#### 6.6.1.4 Generated Code Intra-ECU With IsUpdated

Rte_Write with configured IsUpdated is similar to the variant without IsUpdated. However, in the IsUpdated case, Rte_Write is always implemented as function in Rte_<oa>.c.

```
/**********************************************************************************
 * Update Flags for each Receiver with enableUpdate != 0
 *********************************************************************************/
# define RTE_START_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

VAR(Rte_<oa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<oa>_RxUpdateFlags = {
  0
};

# define RTE_STOP_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_<oa>_RxUpdateFlagsInit() (Rte_MemClr(&Rte_<oa>_RxUpdateFlags,
sizeof(Rte_<oa>_RxUpdateFlagsType)))



#define RTE_START_SEC_CODE
#include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Write_<c>_<p>_<d>(<t> data)
{
  Std_ReturnType ret = RTE_E_OK;

  Rte_WriteHook_<c>_<p>_<d>_Start(data);
  <Lock>();
  Rte_<ci>_<p>_<d> = *(&data);
  Rte_<oa>_RxUpdateFlags.Rte_RxUpdate_<rci>_<rp>_<rd>_Sender =
!Rte_<roa>_RxUpdateFlags.Rte_RxUpdate_<rci>_<rp>_<rd>;
  <UnLock>();
  Rte_WriteHook_<c>_<p>_<d>_Return(data);
  return ret;
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

Rte_<roa>.c declares the variable Rte_<roa>_RxUpdateFlags as follows:

```
# define RTE_START_SEC_VAR_<roa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

VAR(Rte_<roa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<roa>_RxUpdateFlags = {
  0
};

# define RTE_STOP_SEC_VAR_<roa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_<roa>_RxUpdateFlagsInit() (Rte_MemClr(&Rte_<roa>_RxUpdateFlags,
sizeof(Rte_<roa>_RxUpdateFlagsType)))
```

The extern declaration for Rte_<roa>_RxUpdateFlags is declared in Rte_Type.h:

```
# define RTE_START_SEC_VAR_<roa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(Rte_<roa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<roa>_RxUpdateFlags;

# define RTE_STOP_SEC_VAR_<roa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"
```

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to call Rte_<oa>_RxUpdateFlagsInit() and Rte_<roa>_RxUpdateFlagsInit().

The Update flag types are declared in Rte_Type.h

```
typedef struct
{
  Rte_BitType Rte_RxUpdate_<rci>_<rp>_<rd>_Sender : 1;
} Rte_<oa>_RxUpdateFlagsType;


typedef struct
{
  Rte_BitType Rte_RxUpdate_<rci>_<rp>_<rd> : 1;
} Rte_<roa>_RxUpdateFlagsType;
```

#### 6.6.1.5 Configuration Variant Inter-ECU

- source code SWC
- no support for multiple instantiation
- no indirect API
- pure inter-ECU communication
- no transmission acknowledgement
- no invalidation
- no data prototype mapping

#### 6.6.1.6 Generated Code Inter-ECU

Rte_<c>.h defines Rte_Write as follows:

```
#define Rte_Write_<p>_<d> Rte_Write_<c>_<p>_<d>
```

The API is implemented in Rte_<oa>.c

```
# define RTE_START_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

VAR(Rte_<oa>_TxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<oa>_TxUpdateFlags = {
  0,
  0,
};

# define RTE_STOP_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_<oa>_TxUpdateFlagsInit() (Rte_MemClr(&Rte_<oa>_TxUpdateFlags,
sizeof(Rte_<oa>_TxUpdateFlagsType)))

#define RTE_START_SEC_CODE
#include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Write_<c>_<p>_<d>(<t> data)
{
  Std_ReturnType ret = RTE_E_OK;

  Rte_WriteHook_<c>_<p>_<d>_Start(data);
  <Lock>();
```

```
  Rte_<signalid> = *(&data);
 Rte_<oa>_TxUpdateFlags.Rte_TxUpdate_<c>_<p>_<d>                                    =
RTE_COM_SENDSIGNALPROXY_SEND;
  Rte_<oa>_TxUpdateFlags.Rte_TxUpdateProxy_<c>_<p>_<d> =
!Rte_<bswoa>_TxUpdateFlags.Rte_TxUpdateProxy__<c>_<p>_<d>;
  <UnLock>();
  Rte_WriteHook_<c>_<p>_<d>_Return(data);
  return ret;
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

and Rte_<c>.h only contains the prototype:

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Write_<c>_<p>_<d>(<t> data);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"
```

The extern declaration for Rte_<oa>_TxUpdateFlags is declared in Rte_Type.h:

```
typedef struct
{
  Rte_BitType Rte_TxUpdate_<c>_<p>_<e> : 2;
  Rte_BitType Rte_TxUpdateProxy_<c>_<p>_<e> : 1;
} Rte_<oa>_TxUpdateFlagsType;

# define RTE_START_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(Rte_<oa>_TxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<oa>_TxUpdateFlags;

# define RTE_STOP_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"
```

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to call Rte_<oa>_TxUpdateFlagsInit().

In both cases the global variable in Rte_<oa>.c is declared as

```
#define RTE_START_SEC_VAR_<oa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
VAR(<dt>, RTE_VAR_INIT<nocache>) Rte_<signalid> = <initializer>;
#define RTE_STOP_SEC_VAR_<oa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the port data element.

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to do the initialization:

```
Rte_<signalid> = <initializer>;
```

If the data element is of a string or array type, no direct assignments are used. Instead, the assignment is replaced by a memcpy

```
<Rte_MemCpy>(Rte_<signalid>, *(data), sizeof(<t>));
```

If the datatype can be read and written atomically, the <Lock>() and <UnLock>() calls are omitted from the Rte_Write API.

When Rte_Write is not a macro, it needs to be assured that the macros Rte_WriteHook_<c>_<p>_<d>_Start(data) and Rte_WriteHook_<c>_<p>_<d>_Return(data) do not have any side effects.

### 6.6.2 Rte_Read

#### 6.6.2.1 Configuration Variant Without IsUpdated

- source code SWC
- no support for multiple instantiation
- no indirect API
- alive timeout is not configured
- invalidation is not configured
- is updated is not configured
- never received is not configured
- no data prototype mapping
- sender is connected

> intra-ECU or inter-ECU communication

#### 6.6.2.2 Generated Code Without IsUpdated

Rte_<c>.h defines Rte_Read as follows:

```
#define Rte_Read_<p>_<d> Rte_Read_<c>_<p>_<d>
```

When the attribute “EnableTakeAddress” is not set for the port and when the data element can be accessed atomically by the ECU, Rte_Read_<c>_<p>_<d> is declared as macro that reads from a global RTE variable.

```
# define RTE_START_SEC_VAR_<soa><nocache>_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(<t>, RTE_VAR_INIT<nocache>) Rte_<sci>_<sp>_<sd> ;

# define RTE_STOP_SEC_VAR_<soa><nocache>_INIT_UNSPECIFIED
# include "MemMap.h"

#  define Rte_Read_<c>_<p>_<d>(data) (*(data) = Rte_<sci>_<sp>_<sd> ,
    ((Std_ReturnType)RTE_E_OK))
```

Otherwise, the API is implemented in Rte_<oa>.c

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Read_<c>_<p>_<d>(<tp> data)
{
  Std_ReturnType ret = RTE_E_OK;

  Rte_ReadHook_<c>_<p>_<d>_Start(data);
  <Lock>();
  *(data) = Rte_<sci>_<sp>_<sd> ;
  <UnLock>();
  Rte_ReadHook_<c>_<p>_<d>_Return(data);

  return ret;
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

and Rte_<c>.h only contains the prototype:

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Read_<c>_<p>_<d>(<tp> data);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"
```

In both cases the global variable in Rte_<soa>.c is declared as

```
#define RTE_START_SEC_VAR_<soa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
VAR(<dt>, RTE_VAR_INIT<nocache>) Rte_<sci>_<sp>_<sd>  = <initializer>;
#define RTE_STOP_SEC_VAR_<soa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the sender port data element.

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to do the initialization:

```
Rte_<sci>_<sp>_<sd>  = <initializer>;
```

If the data element is a of a string or array type, no direct assignments are used. Instead, the assignment is replaced by a memcpy

```
<Rte_MemCpy>(*(data), Rte_<sci>_<sp>_<sd> , sizeof(<t>));
```

The extern declaration for the global variable is contained in Rte_Type.h:

```
#define RTE_START_SEC_VAR_<soa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
extern VAR(<dt>, RTE_VAR_INIT<nocache>) Rte_<sci>_<sp>_<sd>;
#define RTE_STOP_SEC_VAR_<soa><nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
```

If the datatype can be read and written atomically, the <Lock>() and <UnLock>() calls are omitted from the Rte_Read API.

When Rte_Read is not a macro, it needs to be assured that the macros Rte_ReadHook_<c>_<p>_<d>_Start(data) and Rte_ReadHook_<c>_<p>_<d>_Return(data) do not have any side effects.

When Rte_Read reads data from a component or BSW with lower safety level, sanity checks have to be applied to the input data.

### 6.6.2.3 Configuration Variant With IsUpdated

- source code SWC
- no support for multiple instantiation
- no indirect API
- alive timeout is not configured
- invalidation is not configured
- no data prototype mapping
- is updated is configured
- never received is not configured
- sender is connected
- intra-ECU or inter-ECU communication

### 6.6.2.4 Generated Code With IsUpdated

Rte_Read with configured IsUpdated is similar to the variant without IsUpdated. However, in the IsUpdated case, Rte_Read is always implemented as function in Rte_<oa>.c.

```
/**********************************************************************************
 * Update Flags for each Receiver with enableUpdate != 0
 **********************************************************************************/
# define RTE_START_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

VAR(Rte_<oa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<oa>_RxUpdateFlags = {
  0
};

# define RTE_STOP_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_<oa>_RxUpdateFlagsInit() (Rte_MemClr(&Rte_<oa>_RxUpdateFlags,
```

```
sizeof(Rte_<oa>_RxUpdateFlagsType)))

# define RTE_START_SEC_CODE
# include "MemMap.h"


FUNC(Std_ReturnType, RTE_CODE) Rte_Read_<c>_<p>_<d>(<tp> data)
{
  Std_ReturnType ret = RTE_E_OK;

  Rte_ReadHook_<c>_<p>_<d>_Start(data);
  <Lock>();
  *(data) = Rte_<sci>_<sp>_<sd> ;
  Rte_<oa>_RxUpdateFlags.Rte_RxUpdate_<ci>_<p>_<d> =
Rte_<soa>_RxUpdateFlags.Rte_RxUpdate_<ci>_<p>_<d>_Sender;
  <UnLock>();
  Rte_ReadHook_<c>_<p>_<d>_Return(data);

  return ret;
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

The variable Rte_<soa>_RxUpdateFlags is declared in Rte_<soa>.c as:

```
# define RTE_START_SEC_VAR_<soa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

VAR(Rte_<soa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<soa>_RxUpdateFlags = {
  0
};

# define RTE_STOP_SEC_VAR_<soa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_<soa>_RxUpdateFlagsInit() (Rte_MemClr(&Rte_<soa>_RxUpdateFlags,
sizeof(Rte_<soa>_RxUpdateFlagsType)))
```

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to call Rte_<oa>_RxUpdateFlagsInit() and Rte_<soa>_RxUpdateFlagsInit().

The Update flags are declared in Rte_Type.h

```
typedef struct
{
  Rte_BitType Rte_RxUpdate_<rci>_<rp>_<rd> : 1;
} Rte_<oa>_RxUpdateFlagsType;

typedef struct
{
  Rte_BitType Rte_RxUpdate_<rci>_<rp>_<rd>_Sender : 1;
} Rte_<soa>_RxUpdateFlagsType;


# define RTE_START_SEC_VAR_<soa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(Rte_<soa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<soa>_RxUpdateFlags;

# define RTE_STOP_SEC_VAR_<soa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"


# define RTE_START_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(Rte_<oa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<oa>_RxUpdateFlags;

# define RTE_STOP_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
# include "MemMap.h"
```

Please note that in case of inter-ECU sender/receiver communication <soa> is the BSW OS Application.

### 6.6.3 Rte_IsUpdated

#### 6.6.3.1 Configuration Variant

- source code SWC
- no support for multiple instantiation
- no indirect API
- alive timeout is not configured
- invalidation is not configured
- is updated is configured
- never received is not configured
- sender is connected
- intra-ECU or inter-ECU communication

#### 6.6.3.2 Generated Code

Rte_<c>.h defines Rte_IsUpdated as follows:

```
#  define Rte_IsUpdated_<p>_<d> Rte_IsUpdated_<c>_<p>_<d>
#  define Rte_IsUpdated_<c>_<p>_<d>()
((Rte_<oa>_RxUpdateFlags.Rte_RxUpdate_<rci>_<rc>_<rd> ==
Rte_<soa>_RxUpdateFlags.Rte_RxUpdate_<rci>_<rp>_<rd>_Sender) ? FALSE : TRUE)
```

The Update flags are declared in Rte_Type.h

```
/**********************************************************************************
 *  LOCAL DATA TYPES AND STRUCTURES
 *********************************************************************************/

typedef unsigned int Rte_BitType;

typedef struct
{
  Rte_BitType Rte_RxUpdate_<rci>_<rp>_<rd> : 1;
} Rte_<oa>_RxUpdateFlagsType;

typedef struct
{
  Rte_BitType Rte_RxUpdate_<rci>_<rp>_<rd>_Sender :1;
} Rte_<soa>_RxUpdateFlagsType;
```

```
#  define RTE_START_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
#  include "MemMap.h"

extern VAR(Rte_<oa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<oa>_RxUpdateFlags;

#  define RTE_STOP_SEC_VAR_<oa><nocache>_ZERO_INIT_UNSPECIFIED
#  include "MemMap.h"

#  define RTE_START_SEC_VAR_<soa><nocache>_ZERO_INIT_UNSPECIFIED
#  include "MemMap.h"

extern VAR(Rte_<soa>_RxUpdateFlagsType, RTE_VAR_ZERO_INIT<nocache>)
Rte_<soa>_RxUpdateFlags;

#  define RTE_STOP_SEC_VAR_<soa><nocache>_ZERO_INIT_UNSPECIFIED
#  include "MemMap.h"
```

Variable initialization happens in Rte.c as described for the Rte_Read and Rte_Write APIs (see 6.6.1.4 and 6.6.2.4).

### 6.6.4 Rte_IrvWrite

#### 6.6.4.1 Configuration Variant

> source code SWC

> no support for multiple instantiation

#### 6.6.4.2 Generated Code

When the inter runnables variable can be accessed atomically by the ECU Rte_IrvWrite_<re>_<name> is declared as macro that writes to a global RTE variable.

```
# define RTE_START_SEC_VAR_<oa>_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(<t>, RTE_VAR_INIT) Rte_Irv_<ci>_<name>;

# define RTE_STOP_SEC_VAR_<oa>_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_IrvWrite_<re>_<name>(data) (Rte_Irv_<ci>_<name> = (data))
```

Otherwise, the API is implemented in Rte_<oa>.c

```
#define RTE_START_SEC_CODE
#include "MemMap.h"

FUNC(void, RTE_CODE) Rte_IrvWrite_<c>_<re>_<name>(<t> data)
{
  Rte_IrvWriteHook_<c>_<re>_<name>_Start(data);
  <Lock>();
  Rte_Irv_<ci>_<name> = data;
  <UnLock>();
  Rte_IrvWriteHook_<c>_<re>_<name>_Return(data);
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

and Rte_<c>.h only contains a define to the function:

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(void, RTE_CODE) Rte_IrvWrite_<c>_<re>_<name>(<t> data);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"

#define Rte_IrvWrite_<re>_<name> Rte_IrvWrite_<c>_<re>_<name>
```

In both cases the global variable in Rte_<oa>.c is declared as

```
#define RTE_START_SEC_VAR_<oa>_INIT_UNSPECIFIED
#include "MemMap.h"
VAR(<dt>, RTE_VAR_INIT) Rte_Irv_<ci>_<name> = <initializer>;
#define RTE_STOP_SEC_VAR_<oa>_INIT_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the inter-runnable variable.

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to do the initialization:
Rte_Irv_<ci>_<name> = <initializer>;

If the datatype can be read and written atomically, the <LockInterupts>() and <UnLock>() calls are omitted from the Rte_IrvWrite API.

When Rte_IrvWrite is not a macro, it needs to be assured that the macros Rte_IrvWriteHook_<c>_<re>_<name>_Start(data) and Rte_IrvWriteHook_<c>_<re>_<name>_Return(data) do not have any side effects.

### 6.6.5 Rte_IrvRead

#### 6.6.5.1 Configuration Variant

> source code SWC

> no support for multiple instantiation

#### 6.6.5.2 Generated Code

When the inter runnables variable can be accessed atomically by the ECU Rte_IrvRead_<re>_<name> is declared as macro that reads a global RTE variable.

```
# define RTE_START_SEC_VAR_<oa>_INIT_UNSPECIFIED
# include "MemMap.h"

extern VAR(<t>, RTE_VAR_INIT) Rte_Irv_<ci>_<name>;

# define RTE_STOP_SEC_VAR_<oa>_INIT_UNSPECIFIED
# include "MemMap.h"

# define Rte_IrvRead_<re>_<name>(data) Rte_Irv_<ci>_<name>
```

Otherwise, the API is implemented in Rte_<oa>.c

```
#define RTE_START_SEC_CODE
#include "MemMap.h"

FUNC(<t>, RTE_CODE) Rte_IrvRead_<c>_<re>_<name>(void)
{
  <t> irvValue;

  Rte_IrvReadHook_<c>_<re>_<name>_Start();

  <Lock>();
  irvValue = Rte_Irv_<ci>_<name>;
  <UnLock>();

  Rte_IrvReadHook_<c>_<re>_<name>_Return();

  return irvValue;
}

#define RTE_STOP_SEC_CODE
#include "MemMap.h"
```

and Rte_<c>.h only contains a define to the function:

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(<t>, RTE_CODE) Rte_IrvRead_<c>_<re>_<name>(void);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"

#define Rte_IrvRead_<re>_<name> Rte_IrvRead_<c>_<re>_<name>
```

In both cases the global variable in Rte_<oa>.c is declared as

```
#define RTE_START_SEC_VAR_<oa>_INIT_UNSPECIFIED
#include "MemMap.h"
VAR(<dt>, RTE_VAR_INIT) Rte_Irv_<ci>_<name> = <initializer>;
#define RTE_STOP_SEC_VAR_<oa>_INIT_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the inter-runnable variable.

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to do the initialization:

```
Rte_Irv_<ci>_<name> = <initializer>;
```

If the datatype can be read and written atomically, the <Lock>() and <UnLock>() calls are omitted from the Rte_IrvRead API.

When Rte_IrvRead is not a macro, it needs to be assured that the macros Rte_IrvReadHook_<c>_<re>_<name>_Start() and Rte_IrvReadHook_<c>_<re>_<name>_Return() do not have any side effects.

### 6.6.6 Rte_Pim

#### 6.6.6.1 Configuration Variant

> source code SWC

> no support for multiple instantiation

#### 6.6.6.2 Generated Code

The API Rte_Pim is declared as access to a global RTE variable in Rte_<c>.h

```
#   define RTE_START_SEC_VAR_DEFAULT_RTE_<oa>_PIM_GROUP_UNSPECIFIED
#   include "MemMap.h"

extern VAR(<t>, RTE_VAR_DEFAULT_RTE_<oa>_PIM_GROUP) Rte_<ci>_<name>;

#   define RTE_STOP_SEC_VAR_DEFAULT_RTE_<oa>_PIM_GROUP_UNSPECIFIED
#   include "MemMap.h"



#  define Rte_Pim_<name>() \\
  (&Rte_<ci>_<name>)
```

Depending on the configuration of the memory section (see RTE Technical Reference) the DEFAULT_RTE_<oa>_PIM_GROUP string is replaced by the configured group name.

The Rte_<c>_<name> variable is declared in Rte_<oa>.c:

```
# define RTE_START_SEC_VAR_DEFAULT_RTE_<oa>_PIM_GROUP_UNSPECIFIED
# include "MemMap.h"

VAR(<t>, RTE_VAR_DEFAULT_RTE_<d>_PIM_GROUP) Rte_<ci>_<name>;

# define RTE_STOP_SEC_VAR_DEFAULT_RTE_<oa>_PIM_GROUP_UNSPECIFIED
# include "MemMap.h"
```

### 6.6.7 Rte_CData

#### 6.6.7.1 Configuration Variant

> source code SWC

> no support for multiple instantiation

> no online calibration

#### 6.6.7.2 Generated Code

The API Rte_CData is declared as access to a global RTE variable in Rte_<c>.h

```
#   define RTE_START_SEC_CONST_DEFAULT_RTE_CDATA_GROUP_UNSPECIFIED
#   include "MemMap.h"

extern CONST(<t>, RTE_CONST_DEFAULT_RTE_CDATA_GROUP) Rte_<c>_<name>;

#   define RTE_STOP_SEC_CONST_DEFAULT_RTE_CDATA_GROUP_UNSPECIFIED
#   include "MemMap.h"

#  define Rte_CData_<name>() (Rte_<c>_<name>)
```

Depending on the configuration of the memory section (see RTE Technical Reference) the DEFAULT_RTE_CDATA_GROUP string is replaced by the configured group name.
The Rte_<c>_<name> variable is declared in Rte_<oa>.c:

```
#define RTE_START_SEC_CONST_DEFAULT_RTE_CDATA_GROUP_UNSPECIFIED
#include "MemMap.h"

CONST(<t>, RTE_CONST_DEFAULT_RTE_CDATA_GROUP) Rte_<c>_<name> = <initializer>;

#define RTE_STOP_SEC_CONST_DEFAULT_RTE_CDATA_GROUP_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the calibration parameter = <initializer> is omitted when the calibration parameter does not have an init value.

### 6.6.8 Rte_Prm

#### 6.6.8.1 Configuration Variant

- source code SWC
- no support for multiple instantiation
- no indirect API
- no online calibration
- calibration port is connected

#### 6.6.8.2 Generated Code

When the attribute “EnableTakeAddress” is not set for the calibration port, the API Rte_Prm is declared as access to a global RTE variable in Rte_<c>.h

```
#   define RTE_START_SEC_CONST_DEFAULT_RTE_CALPRM_GROUP_UNSPECIFIED
#   include "MemMap.h"

extern CONST(<t>, RTE_CONST_DEFAULT_RTE_CALPRM_GROUP) Rte_<sc>_<sp>_<sd>;

#   define RTE_STOP_SEC_CONST_DEFAULT_RTE_CALPRM_GROUP_UNSPECIFIED
#   include "MemMap.h"

#  define Rte_Prm_<p>_<d>() (Rte_<sc>_<sp>_<sd> )
```

Depending on the configuration of the memory section (see RTE Technical Reference) the DEFAULT_RTE_CALPRM_GROUP string is replaced by the configured group name.

When “EnableTakeAddress” is set for the calibration port Rte_<c>.h maps the API to a function in Rte_<oa>.c

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(<t>, RTE_CODE) Rte_Prm_<c>_<p>_<d>(void);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"

#  define Rte_Prm_<p>_<d> Rte_Prm_<c>_<p>_<d>
```

Rte_<oa>.c then contains:

```
FUNC(<t>, RTE_CODE) Rte_Prm_<c>_<p>_<d>(void)
{
   return Rte_<sc>_<sp>_<sd>;
}
```

In both cases the Rte_<sc>_<sp>_<sd> variable is declared in Rte.c:

```
#define RTE_START_SEC_CONST_DEFAULT_RTE_CALPRM_GROUP_UNSPECIFIED
#include "MemMap.h"

CONST(<t>, RTE_CONST_DEFAULT_RTE_CALPRM_GROUP) Rte_<sc>_<sp>_<sd>  =
<initializer>;

#define RTE_STOP_SEC_CONST_DEFAULT_RTE_CALPRM_GROUP_UNSPECIFIED
#include "MemMap.h"
```

where <initializer> is the C representation of the init value that is configured for the calibration parameter. No initializer is used when the calibration parameter does not have an init value.

### 6.6.9 Rte_Mode

#### 6.6.9.1 Configuration Variant

> source code SWC

> no support for multiple instantiation

> no indirect API

> mode port is connected

#### 6.6.9.2 Generated Code

The API Rte_Mode is declared as access to a global RTE variable in Rte_<c>.h

When the attribute “EnableTakeAddress” is not set for the mode port, the API is implemented as macro.

```
#  define RTE_START_SEC_VAR<nocache>_INIT_UNSPECIFIED
#  include "MemMap.h"
extern VAR(<t>, RTE_VAR_INIT<nocache>) Rte_ModeMachine_<sci>_<sp>_<sd>;
#  define RTE_STOP_SEC_VAR<nocache>_INIT_UNSPECIFIED
#  include "MemMap.h"

#  define Rte_Mode_<p>_<d>() Rte_ModeMachine_<sci>_<sp>_<sd>
```

Otherwise the API is defined to a function

```
# define RTE_START_SEC_CODE
# include "MemMap.h"
FUNC(<t>, RTE_CODE) Rte_Mode_<c>_<p>_<d>(void);
# define RTE_STOP_SEC_CODE
# include "MemMap.h"

#  define Rte_Mode_<p>_<d> Rte_Mode_<c>_<p>_<d>
```

that is implemented in Rte_<oa>.c

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(<t>, RTE_CODE) Rte_Mode_<c>_<p>_<d>(void)
{
  return Rte_ModeMachine_<sci>_<sp>_<sd>;
}

# define RTE_STOP_SEC_CODE
# include "MemMap.h"
```

In both cases the global variable in Rte_<soa>.c is declared as

```
#define RTE_START_SEC_VAR<nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
VAR(<t>, RTE_VAR_INIT<nocache>) Rte_ModeMachine_<sci>_<sp>_<sd> = <mode>;
#define RTE_STOP_SEC_VAR<nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
```

where <mode> is the mode mode define for the configured init mode.

The extern declaration is contained in Rte_Type.h:

```
#define RTE_START_SEC_VAR<nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
extern VAR(<t>, RTE_VAR_INIT<nocache>) Rte_ModeMachine_<sci>_<sp>_<sd>;
#define RTE_STOP_SEC_VAR<nocache>_INIT_UNSPECIFIED
#include "MemMap.h"
```

For systems where the compiler does not initialize global variables, the API Rte_InitMemory needs to do the initialization:

```
Rte_ModeMachine_<sci>_<sp>_<sd> = <mode>;
```

### 6.6.10 Rte_Call

#### 6.6.10.1 Configuration Variant

- source code SWC
- no support for multiple instantiation
- no indirect API
- Synchronous call
- Direct call:
  - server runnable is not mapped
  - server runnable is mapped to the same task
  - server runnable has CanBeInvokedConcurrently set to true

#### 6.6.10.2 Generated Code

When the server runnable has a return code and no port defined arguments, the API Rte_Call is declared in Rte_<c>.h as macro:

```
#  define <c>_START_SEC_CODE
#  include "MemMap.h"
FUNC(Std_ReturnType, <c>_CODE) <sres>(<arglist>);
#  define <c>_STOP_SEC_CODE
#  include "MemMap.h"

#  define Rte_Call_<p>_<o> <sres>
```

When the server runnable does not have a return code or if there are port defined arguments and “EnableTakeAddress” is not set Rte_Call is declared as:

```
#  define <c>_START_SEC_CODE
#  include "MemMap.h"
FUNC(void, <c>_CODE) <sres>(<parglist><arglist>);
#  define <c>_STOP_SEC_CODE
#  include "MemMap.h"

#  define Rte_Call_<p>_<o>(<arglist>) (<sres>(<parglist><arglist>),
((Std_ReturnType)RTE_E_OK))
```

When the server runnable does not have a return code or if there are port defined arguments and “EnableTakeAddress” is set, Rte_Call is declared as:

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Call_<c>_<p>_<o>(<parglist><arglist>);

# define RTE_STOP_SEC_CODE
# include "MemMap.h"

# define Rte_Call_<p>_<o> Rte_Call_<c>_<p>_<o>
```

and Rte_<oa>.c contains the code

```
# define RTE_START_SEC_CODE
# include "MemMap.h"

FUNC(Std_ReturnType, RTE_CODE) Rte_Call_<c>_<p>_<o>(<parglist><arglist>)
{
  Std_ReturnType ret = RTE_E_OK;
  Rte_CallHook_<c>_<p>_<o>_Start(<parglist><arglist>);

  Rte_Runnable_<sc>_<sre>_Start(<parglist><arglist>);
  <sres>(<arglist>);
  Rte_Runnable_<sc>_<sre>_Return(<parglist><arglist>);

  Rte_CallHook_<c>_<p>_<o>_Return(<parglist><arglist>);
  return ret;
}

# define RTE_STOP_SEC_CODE
# include "MemMap.h"
```

The return value of the Rte_Call API needs to be evaluated when the server operation has configured application return codes.

When Rte_Call is not a macro, it needs to be assured that the Rte_CallHook_ and Rte_Runnable_ macros do not have any side effects.

### 6.6.11 Rte_Enter

#### 6.6.11.1 Configuration Variant

- source code SWC
- no support for multiple instantiation
- Implementation method is set to OS interrupt blocking

#### 6.6.11.2 Generated Code

The API Rte_Enter is declared in Rte_<c>.h as

```
#  define Rte_Enter_<name>() \
  { \
    Rte_EnterHook_<c>_<name>_Start(); \
    SuspendOSInterrupts(); \
    Rte_EnterHook_<c>_<name>_Return(); \
  }
```

It needs to be assured that the macros Rte_EnterHook_<c>_<name>_Start() and Rte_EnterHook_<c>_<name>_Return() do not have any side effects.

It has to be assured that no included non-OS header or SWC code redefines the SuspendOSInterrupts() and ResumeOSInterrupts() calls.

### 6.6.12 Rte_Exit

#### 6.6.12.1 Configuration Variant

> source code SWC

> no support for multiple instantiation

> Implementation method is set to OS interrupt blocking

#### 6.6.12.2 Generated Code

The API Rte_Exit is declared in Rte_<c>.h as

```
#  define Rte_Exit_<name>() \
  { \
    Rte_ExitHook_<c>_<name>_Start(); \
    ResumeOSInterrupts(); \
    Rte_ExitHook_<c>_<name>_Return(); \
  }
```

It has to be assured that no included non-OS header or SWC code redefines the SuspendOSInterrupts() and ResumeOSInterrupts() calls.

It needs to be assured that the macros Rte_ExitHook_<c>_<name>_Start() and Rte_ExitHook_<c>_<name>_Return() are do not have any side effects.

## 6.7 BSW specifc RTE APIs

### 6.7.1 SchM_Enter

#### 6.7.1.1 Configuration Variant

> source code BSW

> Implementation method is set to All or OS interrupt blocking

#### 6.7.1.2 Generated Code

The API SchM_Enter is declared in SchM_<bsw>.h as

```
#  define SchM_Enter_<bsw>_<name>() \
  { \
    SuspendAllInterrupts(); \
  }
```

for ImplementationMethod All Interrupt Blocking. Otherwise SuspendOSInterrupts() is called.

It has to be assured that no included non-OS header or BSW code redefines the SuspendAllInterrupts()/SuspendOSInterrupts() and ResumeAllInterrupts()/ResumeOSInterupts() calls.

### 6.7.2 SchM_Exit

#### 6.7.2.1 Configuration Variant

> source code BSW

> Implementation method is set to All or OS interrupt blocking

#### 6.7.2.2 Generated Code

The API SchM_Exit is declared in SchM_<bsw>.h as

```
#  define SchM_Exit_<bsw>_<name>() \
  { \
    ResumeAllInterrupts(); \
  }
```

for ImplementationMethod All Interrupt Blocking. Otherwise ResumeOSInterrupts() is called.

It has to be assured that no included non-OS header or BSW code redefines the SuspendAllInterrupts()/SuspendOSInterrupts() and ResumeAllInterrupts()/ResumeOSInterupts() calls.

## 6.8 RTE Lifecycle APIs

### 6.8.1 Rte_Start

Rte_Start is not called from the ASIL context.

### 6.8.2 Rte_Stop

Rte_Stop is not called from the ASIL context.

### 6.8.3 Rte_InitMemory

Rte_InitMemory needs to be called before the OS sets up the memory protection when the compiler does not initialize global variables. The method itself and the method it calls need to do all initialization for the APIs that are listed in chapter 6.6.

## 6.9 RTE Internal Functions

### 6.9.1 Rte_MemCpy

Rte_MemCpy is called by RTE code that is called from the ASIL SWCs in order to copy data from the memory location “source” to the memory location „destination“. When Rte_MemCpy is called from the ASIL SWCs it needs to be checked that the parameter num which specifies the number of bytes that shall be copied is of type uint16 and that the memory regions “destination” and “source” contain num bytes. Moreover destination needs to be writable. For larger data sizes, the generator uses a method Rte_MemCpy32 that copies in blocks of 4 byte when source and destination are aligned accordingly.

```
FUNC(void, RTE_CODE) Rte_MemCpy(P2VAR(void, AUTOMATIC, RTE_APPL_VAR)
destination, P2CONST(void, AUTOMATIC, RTE_APPL_DATA) source, uint16_least num)

FUNC(void, RTE_CODE) Rte_MemCpy32(P2VAR(void, AUTOMATIC, RTE_APPL_VAR)
destination, P2CONST(void, AUTOMATIC, RTE_APPL_DATA) source, uint16_least num)
```

The functionality of Rte_MemCpy and Rte_MemCpy32 needs to be checked. Rte_MemCpy and Rte_MemCpy32 must not read and write outside the specified memory regions and the alignment requirements of the target platform need to be fulfilled.

### 6.9.2 Rte_MemClr

Rte_MemClr is not called from the ASIL SWCs. It is used for the initialization of global variables with zeros within Rte.c and Rte_<oa>.c.

```
STATIC FUNC(void, RTE_CODE) Rte_MemClr(P2VAR(void, AUTOMATIC, RTE_VAR_NOINIT)
ptr, uint16_least num);
```

It has to be checked, that the first parameter is a pointer to a writable variable and that the second parameter is the length of the variable. Rte_MemClr is not allowed to write outside the specified memory region.

## 6.10 RTE Tasks

It needs to be checked that Rte_<oa>.c contains the implementation of all RTE tasks that are assigned to ASIL OS Applications:

```
TASK(<name>)
{
  Rte_Task_Dispatch(<name>);

  /* call runnable */
  Rte_Runnable_<c>_<re>_Start();
  <res>();
  Rte_Runnable_<c>_<re>_Return();

  (void)TerminateTask();
}
```

It needs to be checked that the trace hooks

- Rte_Task_Dispatch
- Rte_Runnable_<c>_<re>_Start
- Rte_Runnable_<c>_<re>_Return

do not have any side effects.
The task body is only allowed to contain calls to runnables and schedulable entities that are mapped to the task in the configuration and calls to TerminateTask. The call to TerminateTask needs to be the last operation in the task and it always needs to be done.

It has to be assured that no included non-OS header or RTE code redefines the TASK() and TerminateTask() calls.

The name within the TASK call needs to be the name of the configured OS task.

For schedulable entities, the calls to the hooks are omitted by default.

## 6.11 Verification of OS Configuration

The integrator is responsible for correctly configuring the OS.

It needs to be checked that the OS contains no trusted OS Applications that do not have the highest ASIL.

It needs to be checked that the OS contains no tasks that are assigned to an ASIL OS Application and that are not implemented with the given ASIL.

## 6.12 Verification of Memory Mapping Configuration

It needs to be verified that all variables Rte_<sci>_<p>_<d> of sender port data elements, Rte_<oa>_RxUpdateFlags, Rte_Irv_<ci>_<name> of inter-runnable variables and Rte_<ci>_<name> of per-instance memories from ASIL SWCs are mapped to memory sections of the SWC's OS Application so that they are protected from writes outside this OS Application.

When the variables are assigned to different sections or when other RTE variables are assigned to the section, memory protection faults might occur.

# 7 Safety Lifecycle Tailoring

The development of the MICROSAR RTE started as Safety Element out of context at the software unit level.

The unit design is based on the requirements of the AUTOSAR RTE specification [1].

Based on the requirements and the RTE design a set of test cases with typical configurations were derived. The RTE was generated for the test cases and compiled with SWC stubs. Finally, the code was runtime tested on different target platforms and a MISRA analysis was performed.

During the development of the MICROSAR RTE, assumptions regarding the architecture and the safety requirements were made. The integrator is responsible for creating a complete architecture design and for specifying the software safety requirements. He then needs to verify the assumptions that are listed within this document.

As the generated RTE code heavily depends on the input configuration, it is also the responsibility of the integrator to integrate and test the generated RTE code.

Furthermore, it needs to be verified that the generated code fulfils the safety requirements of the target system.

# 8 Glossary and Abbreviations

## 8.1 Glossary

| Term | Description |
|---|---|
| DaVinci DEV | DaVinci Developer: The SWC and RTE Configuration Editor. |
| DaVinci CFG | DaVinci Configurator: The BSW and RTE Configuration Editor. |
| E2E PW | E2E Protection Wrapper: Wrapper Functions to access the E2E Library |

Table 8-1 Glossary

## 8.2 Abbreviations

| Abbreviation | Description |
|---|---|
| API | Application Programming Interface |
| ASIL | Automotive Safety Integrity Level |
| AUTOSAR | Automotive Open System Architecture |
| BSW | Basis Software |
| E2E | AUTOSAR End-to-End Communication Protection Library |
| ECU | Electronic Control Unit |
| H&R | Hazard and Risk Analysis |
| HIS | Hersteller Initiative Software |
| ISO | International Organization for Standardization |
| ISR | Interrupt Service Routine |
| MICROSAR | Microcontroller Open System Architecture (the Vector AUTOSAR solution) |
| MPU | Memory Protection Unit (realized in hardware by the processor) |
| RTE | Runtime Environment |
| SEooC | Safety Element out of Context: a safety-related element which is not developed for a specific item |
| SWC | Software Component |
| OS | Operating System |
| OS Application | An OS Application is a set of tasks and ISRs with a partial common MPU setting |
| TCL | Tool Qualification Level |
| QM | Quality Management (used for software parts developed following only a standard quality management process) |

Table 8-2 Abbreviations

# 9 Contact

Visit our website for more information on

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

www.vector.com