Document Title	Specification of RTE
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	084
Document Classification	Standard

Document Version	3.2.0
Document Status	Final
Part of Release	4.0
Revision	3

Document Change History

Date	Version	Changed by	Change Description
26.10.2011	3.2.0	AUTOSAR Administration	• Adapted to new version of meta model • Support for mixed compu methods with categories SCALE_LINEAR_AND_TEXTTABLE and SCALE_RATIONAL_AND_TEXTTABLE added • Support for compatibility of partial record types added • Consolidation of signal invalidation, data conversion, and out-of-range handling • General consolidation and bug fixes

29.10.2010	3.1.0	AUTOSAR Administration	• Adapted to new version of meta model • Backward compatibility to implicit communication behavior of AUTOSAR 2.1/3.0/3.1 added • Support of inter-runnable variables extended to composite data types • Clarification which API calls shall be implemented as macro accesses to the component data structure in compatibility mode (see rte_sws_1156) • General consolidation and bug fixes
18.12.2009	3.0.0	AUTOSAR Administration	• Adapted to new version of meta model • RTE and Basic Software Scheduler merged • Support of multi core architectures added • Re-scaling at ports added • API enhancements added

04.02.2009	2.1.0	AUTOSAR Administration	• updated VFB-Tracing: changes rte_sws_1327,rte_sws_1328 • unconnected R-Ports are supported: changed rte_sws_1329, rte_sws_3019; added rte_sws_1330, rte_sws_1331, rte_sws_1333, rte_sws_1334, rte_sws_1336, rte_sws_1337, rte_sws_1346, rte_sws_2621, rte_sws_2638, rte_sws_2639, rte_sws_2640, rte_sws_3785, rte_sws_5099, rte_sws_5100, rte_sws_5101, rte_sws_5102 • incompatible function declarations: changed rte_sws_1018, rte_sws_1019, rte_sws_1020; added rte_sws_5107, rte_sws_5108, rte_sws_5109; removed rte_sws_6030. • Insufficient RTE server mapping requirement: changed rte_sws_2204.
15.02.2008	2.0.1	AUTOSAR Administration	Layout adaptations
20.12.2007	2.0.0	AUTOSAR Administration	• Adapted to new version of meta model • "RTE ECU Configuration" added • Calibration and measurement revised • Document meta information extended • Small layout adaptations made
31.01.2007	1.1.1	AUTOSAR Administration	• "Advice for users" revised • "Revision Information" added

01.12.2006	1.1.0	AUTOSAR Administration	Updated for AUTOSAR Release 2.1. • Adapted to new version of meta model • New feature 'debouncing of runnable activation' • New feature 'runnable activation offset' • 'Measurement and Calibration' added • Semantics of implicit communication enhanced • Legal disclaimer revised
18.07.2006	1.0.1	AUTOSAR Administration	Second release. Additional features integrated, adapted to updated version of meta-model.
05.05.2006	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

This specification and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the specification.

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.
For any other purpose, no part of the specification may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice to users of AUTOSAR Specification Documents

AUTOSAR specifications may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

References

[1] Virtual Functional Bus
AUTOSAR_EXP_VFB.pdf

[2] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[3] Specification of Communication
AUTOSAR_SWS_COM.pdf

[4] Specification of Operating System
AUTOSAR_SWS_OS.pdf

[5] Requirements on ECU Configuration
AUTOSAR_RS_ECUConfiguration.pdf

[6] Methodology
AUTOSAR_TR_Methodology.pdf

[7] Specification of ECU State Manager with fixed state machine
AUTOSAR_SWS_ECUStateManagerFixed.pdf

[8] System Template
AUTOSAR_TPS_SystemTemplate.pdf

[9] Basic Software Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

[10] Generic Structure Template
AUTOSAR_TPS_GenericStructureTemplate.pdf

[11] Glossary
AUTOSAR_TR_Glossary.pdf

[12] Specification of Multi-Core OS Architecture
AUTOSAR_SWS_MultiCoreOS.pdf

[13] Specification of Interoperability of AUTOSAR Tools
AUTOSAR_TR_InteroperabilityOfAutosarTools.pdf

[14] Specification of Timing Extensions
AUTOSAR_TPS_TimingExtensions.pdf

[15] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[16] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[17] Specification of ECU Resource Template
AUTOSAR_TPS_ECUResourceTemplate.pdf

[18] Specification of I/O Hardware Abstraction
AUTOSAR_SWS_IOHardwareAbstraction.pdf

[19] Requirements on Operating System
AUTOSAR_SRS_OS.pdf

[20] Requirements on Multi-Core OS Architecture
AUTOSAR_SRS_MultiCoreOS.pdf

[21] Requirements on Communication
AUTOSAR_SRS_COM.pdf

[22] ASAM MCD 2MC ASAP2 Interface Specification
http://www.asam.net
ASAP2-V1.51.pdf

[23] Specification of NVRAM Manager
AUTOSAR_SWS_NVRAMManager.pdf

[24] API Specification of Development Error Tracer
AUTOSAR_SWS_DevelopmentErrorTracer.pdf

[25] Gemeinsames Subset der MISRA C Guidelines
HIS_SubSet_MISRA_C_1.0.3.pdf

[26] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

[27] Specification of Debugging in AUTOSAR
AUTOSAR_SWS_Debugging.pdf

[28] Specification of Compiler Abstraction
AUTOSAR_SWS_CompilerAbstraction.pdf

[29] Specification of Standard Types
AUTOSAR_SWS_StandardTypes.pdf

[30] Specification of Diagnostic Log and Trace
AUTOSAR_SWS_DiagnosticLogAndTrace.pdf

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

Note on XML examples

This specification includes examples in XML based on the AUTOSAR metamodel available at the time of writing. These examples are included as illustrations of configurations and their expected outcome but should not be considered part of the specification.

1 Introduction

This document contains the software specification of the AUTOSAR Run-Time Environment (*RTE*) and the *Basic Software Scheduler*. Basically, the RTE together with the OS, AUTOSAR COM and other Basic Software Modules is the implementation of the Virtual Functional Bus concepts (*VFB*, [1]). The RTE implements the AUTOSAR Virtual Functional Bus interfaces and thereby realizes the communication between AUTOSAR software-components.

This document describes how these concepts are realized within the RTE. Furthermore, the Application Programming Interface (*API*) of the RTE and the interaction of the RTE with other basic software modules is specified.

The *Basic Software Scheduler* offers concepts and services to integrate Basic Software Modules Hence, the *Basic Software Scheduler*

- embed *Basic Software Module* implementations into the AUTOSAR OS context
- trigger main processing functions of the *Basic Software Module*s
- apply data consistency mechanisms for the *Basic Software Module*s
- to communicate modes between *Basic Software Module*s

1.1 Scope

This document is intended to be the main reference for developers of an RTE generator tool or of a concrete RTE implementation respectively. The document is also the reference for developers of AUTOSAR software-components and basic software modules that interact with the RTE, since it specifies the application programming interface of the RTE and therefore the mechanisms for accessing the RTE functionality. Furthermore, this specification should be read by the AUTOSAR working groups that are closely related to the RTE (see Section 1.2 below), since it describes the interfaces of the RTE to these modules as well as the behavior / functionality the RTE expects from them.

This document is structured as follows. After this general introduction, Chapter 2 gives a more detailed introduction of the concepts of the RTE. Chapter 3 describes how an RTE is generated in the context of the overall AUTOSAR methodology. Chapter 4 is the central part of this document. It specifies the RTE functionality in detail. The RTE API is described in Chapter 5.

The appendix of this document consists of five parts: Appendix A lists the restrictions to the AUTOSAR metamodel that this version of the RTE specification relies on. Appendix B explicitly lists all external requirements, i.e. all requirements that are not about the RTE itself but specify the assumptions on the environment and the input of an RTE generator. In Appendix C some HIS MISRA rules are listed that are likely to be violated by RTE code, and the rationale why these violations may occur.

Note that Chapters 1 and 2, as well as Appendix C do not contain any requirements and are thus intended for information only.

Chapters 4 and 5 are probably of most interest for developers of an RTE Generator. Chapters 2, 3, 5 are important for developers of AUTOSAR software-components and basic software modules. The most important chapters for related AUTOSAR work packages would be Chapters 4, 5, as well as Appendix B.

The specifications in this document do not define details of the implementation of a concrete RTE or RTE generator respectively. Furthermore, aspects of the ECU- and system-generation process (like e.g. the mapping of SW-Cs to ECUs, or schedulability analysis) are also not in the scope of this specification. Nevertheless, it is specified what input the RTE generator expects from these configuration phases.

1.2 Dependency to other AUTOSAR specifications

The main documents that served as input for the specification of the RTE are the specification of the Virtual Functional Bus [1] and the specification of the Software Component Template [2]. Also of primary importance are the specifications of those Basic Software modules that closely interact with the RTE (or vice versa). These are especially the communication module [3] and the operating system [4]. The main input of an RTE generator is described (among others) in the ECU Configuration Description. Therefore, the corresponding specification [5] is also important for the RTE specification. Furthermore, as the process of RTE generation is an important part of the overall AUTOSAR Methodology, the corresponding document [6] is also considered.

The following list shows the specifications that are closely interdependent to the specification of the RTE:

- Specification of the Virtual Functional Bus [1]
- Specification of the Software Component Template [2]
- Specification of AUTOSAR COM [3]
- Specification of AUTOSAR OS [4]
- Specification of ECU State Manager and Communication Manager [7]
- Specification of ECU Configuration [5]
- Specification of System Description / Generation [8]

- AUTOSAR Methodology [6]
- Specification of BSW Module Description Template [9]
- AUTOSAR Generic Structure Template [10]

1.3 Acronyms and Abbreviations

All abbreviations used throughout this document – except the ones listed here – can be found in the official AUTOSAR glossary [11].

1.4 Technical Terms

All technical terms used throughout this document – except the ones listed here – can be found in the official AUTOSAR glossary [11] or the Software Component Template Specification [2].

Term	Description
mode switch port	The port for receiving (or sending) a mode switch notification. For this purpose, a *mode switch port* is typed by a `ModeSwitchInterface`.
mode user	An *AUTOSAR SW-C* or *AUTOSAR Basic Software Module* that depends on modes by *ModeDisablingDependency*, *SwcModeSwitchEvent*, *BswModeSwitchEvent*, or simply by reading the current state of a mode is called a `mode user`. A mode user is defined by having a require `mode switch port` or a *requiredModeGroup ModeDeclarationGroupPrototype*. See also section 4.4.1.
mode manager	Entering and leaving modes is initiated by a *mode manager*. A *mode manager* is defined by having a provide `mode switch port` or a *providedModeGroup ModeDeclarationGroupPrototype*. A *mode manager* might be either an `application mode manager` or a *Basic Software Module* that provides a service including mode switches, like the ECU State Manager. See also section 4.4.2.
application mode manager	An *application mode manager* is an *AUTOSAR software-component* that provides the service of switching modes. The modes of an `application mode manager` do not have to be standardized.

mode switch notification	The communication of a mode switch from the `mode manager` to the `mode user` using either the `ModeSwitchInterface` or *providedModeGroup* and *requiredModeGroup ModeDeclarationGroupPrototype*s is called *mode switch notification*.
mode machine instance	The instances of mode machines or *ModeDeclarationGroup*s are defined by the *ModeDeclarationGroupPrototype*s of the `mode managers`. Since a mode switch is not executed instantaneously, The RTE or *Basic Software Scheduler* has to maintain it's own states. For each `mode manager`'s ModeDeclarationGroupPrototype, RTE or *Basic Software Scheduler* has one state machine. This state machine is called *mode machine instance*. For all `mode users` of the same `mode manager`'s *ModeDeclarationGroupPrototype*, RTE and *Basic Software Scheduler* uses the same *mode machine instance*. See also section 4.4.2.
common mode machine instance	A 'common mode machine instance' is a special 'mode machine instance' shared by BSW Modules and SW-Cs: The RTE Generator creates only one `mode machine instance` if a *ModeDeclarationGroupPrototype* instantiated in a port of a software-component is synchronized (*synchronizedModeGroup* of a *SwcBswMapping*) with a *providedModeGroup ModeDeclarationGroupPrototype* of a Basic Software Module instance. The related `mode machine instance` is called `common mode machine instance`.
ModeDisablingDependency	An *RTEEvent* and *BswEvent* that starts a *Runnable Entity* respectively a *Basic Software Schedulable Entity* can contain a *disabledInMode* association which *references* a *ModeDeclaration*. This association is called *ModeDisablingDependency* in this document.
mode disabling dependent ExecutableEntity	A mode disabling dependent *Runnable Entity* or a *Basic Software Schedulable Entity* is triggered by an `RTEEvent` respectively a `BswEvent` with a `ModeDisablingDependency`. RTE and *Basic Software Scheduler* prevent the start of those *Runnable Entity* or *Basic Software Schedulable Entity* by the `RTEEvent` / `BswEvent`, when the corresponding `mode disabling` is active. See also section 4.4.1.

mode disabling	When a 'mode disabling' is active, RTE and *Basic Software Scheduler* disables the start of `mode disabling dependent ExecutableEntitys`. The 'mode disabling' is active during the mode that is referenced in the mode disabling dependency and during the transitions that enter and leave this mode. See also section 4.4.1.
OnEntry ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchEvent* respectively a *BswModeSwitchEvent* with *ModeActivationKind* 'entry' is triggered on entering the mode. It is called *OnEntry ExecutableEntity*. See also section 4.4.1.
OnExit ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchEvent* respectively a *BswModeSwitchEvent* with *ModeActivationKind* 'exit' is triggered on exiting the mode. It is called *OnExit ExecutableEntity*. See also section 4.4.1.
OnTransition ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchEvent* respectively a *BswModeSwitchEvent* with *ModeActivationKind* 'transition' is triggered on a transition between the two specified modes. It is called *OnTransition ExecutableEntity*. See also section 4.4.1.
mode switch acknowledge ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchedAckEvent* respectively a *BswModeSwitchedAckEvent* connected to the `mode manager`'s *ModeDeclarationGroupPrototype*. It is called *mode switch acknowledge ExecutableEntity*. See also section 4.4.1.
server runnable	A server that is triggered by an *OperationInvokedEvent*. It has a mixed behavior between a runnable and a function call. In certain situations, RTE can implement the client server communication as a simple function call.
runnable activation	The activation of a runnable is linked to the `RTEEvent` that leads to the execution of the runnable. It is defined as the incident that is referred to by the `RTEEvent`. E.g., for a timing event, the corresponding runnable is activated, when the timer expires, and for a data received event, the runnable is activated when the data is received by the RTE.

Basic Software Schedulable Entity activation	The activation of a *Basic Software Schedulable Entity* is defined as the activation of the task that contains the *Basic Software Schedulable Entity* and eventually includes setting a flag that tells the glue code in the task which *Basic Software Schedulable Entity* is to be executed.
runnable start	A runnable is started by the calling the C-function that implements the runnable from within a started task.
Basic Software Schedulable Entity start	A *Basic Software Schedulable Entity* is started by the calling the C-function that implements the *Basic Software Schedulable Entity* from within a started task.
Trigger Emitter	A *Trigger Emitter* has the ability to release triggers which in turn are activating `triggered ExecutableEntitys`. *Trigger Emitter* are described by the meta model with provide `trigger ports`, `Trigger` in role `releasedTrigger`, `InternalTriggeringPoints` and `BswInternalTriggeringPoints`.
Trigger Source	A *Trigger Source* administrate the particular *Trigger* and informs the RTE or *Basic Software Scheduler* if the *Trigger* is raised. A *Trigger Source* has dedicated provide `trigger port`(s) or / and *releasedTrigger Trigger*(s) to communicate to the `Trigger Sink`(s).
Trigger Sink	A *Trigger Sink* relies on the activation of *Runnable Entities* or *Basic Software Schedulable Entities* if a particular *Trigger* is raised. A *Trigger Sink* has a dedicated require `trigger port`(s) or / and *requiredTrigger Trigger*(s) to communicate to the `Trigger Source`(s).
trigger port	A `PortPrototype` which is typed by an `TriggerInterface`
triggered ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered at least by one `ExternalTriggerOccurredEvent` / `BswExternalTriggerOccurredEvent` or `InternalTriggerOccurredEvent` / `BswInternalTriggerOccurredEvent`. In particular cases, the Trigger Event Communication or the *Inter Runnable Triggering* is implemented by RTE or *Basic Software Scheduler* as a direct function call of the *triggered ExecutableEntity* by the triggering *ExecutableEntity*.

triggered runnable	A *Runnable Entity* that is triggered at least by one `ExternalTriggerOccurredEvent` or `InternalTriggerOccurredEvent`. In particular cases, the Trigger Event Communication or the *Inter Runnable Triggering* is implemented by RTE as a direct function call of the *triggered runnable* by the triggering runnable.
triggered Basic Software Schedulable Entity	A *Basic Software Schedulable Entity* that is triggered at least by one `BswExternalTriggerOccurredEvent` or `BswInternalTriggerOccurredEvent`. In particular cases, the *Trigger Event Communication* or the *Inter Basic Software Schedulable Entity Triggering* is implemented by *Basic Software Scheduler* as a direct function call of the *triggered ExecutableEntity* by the triggering *ExecutableEntity*.
execution-instance	An execution-instance of a `ExecutableEntity` is one instance or call context of an `ExecutableEntity` with respect to concurrent execution, see section 4.2.3.
inter-ECU communication	The communication between ECUs, typically using COM is called `inter-ECU` communication in this document.
inter-partition communication	The communication within one ECU but between different partitions, represented by different OS applications, is called `inter-partition` communication in this document. It typically involves the use of OS mechanisms like IOC or trusted function calls. The partitions can be located on different cores or use different memory sections of the ECU.
intra-partition communication	The communication within one partition of one ECU is called `intra-partition` communication. In this case, RTE can make use of internal buffers and queues for communication.
intra-ECU communication	The communication within one ECU is called `intra-ECU` communication in this document. It is a super set of `inter-partition` communication and `intra-partition` communication.
SystemDesignTime Variability	Variability defined with an `VariationPoint` or *AttributeValueVariationPoint* with latest `bindingTime SystemDesignTime`.
CodeGenerationTime Variability	Variability defined with an `VariationPoint` or *AttributeValueVariationPoint* with latest `bindingTime CodeGenerationTime`.

PreCompileTime Variability	Variability defined with an `VariationPoint` or *AttributeValueVariationPoint* with latest `bindingTime` `PreCompileTime`.
LinkTime Variability	Variability defined with an `VariationPoint` or *AttributeValueVariationPoint* with latest `bindingTime` `LinkTime`.
PreBuild Variability	Variability defined with an `VariationPoint` or *AttributeValueVariationPoint* with latest `bindingTime` `SystemDesignTime`, `CodeGenerationTime`, `PreCompileTime` or `LinkTime`.
PostBuild Variability	Variability defined with an `VariationPoint` having an `postBuildVariantCriterion`
Preemption Area	A preemption area defines a set of tasks which are scheduled cooperatively. Therefore tasks of one preemption area are preempting each other only at dedicated schedule points. A schedule point is not allowed to occur during the execution of a `RunnableEntity`.
Copy Semantic	Copy semantic means, that the accessing entities are able to read or write the "copied" data from their execution context in a non concurrent and non preempting manner. If all accessing entities are in the same `Preemption Area` this might not require a real physical data copy.
Primitive Data Type	Primitive data types are the types implemented by a boolean, integer (up to 32 bits), floating point, or opaque type (up to 32 bits).
NvBlockSwComponent	NvBlockSwComponent is a `ComponentPrototype` typed an `NvBlockSwComponentType`.
'C' typed PerInstanceMemory	'C' typed PerInstanceMemory is defined with the class `PerInstanceMemory`. The type of the memory is defined with a 'C' typedef in the attribute `typeDefinition`.
`AutosarDataPrototype` implementation	Definitions and declarations for non automatic[1] memory objects which are allocated by the RTE and implementing `AutosarDataPrototype`s or their belonging status handling.
Implicit Read Access	`VariableAccess` aggregated in the role `dataReadAccess` to a `VariableDataPrototoype`
Implicit Write Access	`VariableAccess` aggregated in the role `dataWriteAccess` to a `VariableDataPrototoype`

[1] declaration with no static or external specifier defines an automatic variable

Incoherent Implicit Data Access	An `Implicit Read Access` or an `Implicit Write Access` which does not belong to a `Coherency Group`. Therefore it is NOT referenced by any `RteVariableReadAccessRef` or `RteVariableWriteAccessRef` belonging to a `RteImplicitCommunication` container which `RteCoherentAccess` parameter is set to true.
Incoherent Implicit Read Access	An `Implicit Read Access` which does not belong to a `Coherency Group`. Therefore it is NOT referenced by any `RteVariableReadAccessRef` belonging to a `RteImplicitCommunication` container which `RteCoherentAccess` parameter is set to true.
Incoherent Implicit Write Access	An `Implicit Write Access` which does not belong to a `Coherency Group`. Therefore it is NOT referenced by any `RteVariableWriteAccessRef` belonging to a `RteImplicitCommunication` container which `RteCoherentAccess` parameter is set to true.
Coherency Group	A set of `Implicit Read Accesses` and `Implicit Write Accesses` for which the RTE cares for data coherency. Please note that in the context of this specification the definition of coherency includes that • read data values of different `VariableDataPrototypes` have to be from the same age, except the values are changed by `Implicit Write Accesses` belonging to the Coherency Group • written data values of different `VariableDataPrototypes` are communicated to readers NOT belonging to the Coherency Group after the last `Implicit Write Access` belonging to the Coherency Group.
Coherent Implicit Data Access	An `Implicit Read Access` or an `Implicit Write Access` which belongs to `Coherency Group`. Therefore it is referenced by a `RteVariableReadAccessRef` or `RteVariableWriteAccessRef` belonging to a `RteImplicitCommunication` container which `RteCoherentAccess` parameter is set to true.

Coherent Implicit Read Access	An `Implicit Read Access` which belongs to `Coherency Group`. Therefore it is referenced by a `RteVariableReadAccessRef` belonging to a `RteImplicitCommunication` container which `RteCoherentAccess` parameter is set to true.
Coherent Implicit Write Access	An `Implicit Write Access` which belongs to `Coherency Group`. Therefore it is referenced by a `RteVariableReadAccessRef` or `RteVariableWriteAccessRef` belonging to a `RteImplicitCommunication` container which `RteCoherentAccess` parameter is set to true.

1.5 Document Conventions

Requirements in the SRS are referenced using [RTE<n>] where <n> is the requirement id. For example, [RTE00098].

Requirements in the SWS are marked with **[rte_sws_<n>]**⌈ as the first text in a paragraph. The scope of the requirement is marked with the half brackets. ⌋*()*

External requirements on the input of the RTE are marked with **[rte_sws_ext_<n>]**.

Technical terms are typeset in monospace font, e.g. `Warp Core`.

API function calls are also marked with monospace font, like `Rte_ejectWarpCore()`.

1.6 Requirements Traceability

Requirement	Satisfied by
[BSW00300] Module naming convention	rte_sws_1171 rte_sws_1003 rte_sws_7122 rte_sws_7922 rte_sws_7139 rte_sws_1157 rte_sws_1158 rte_sws_1169 rte_sws_1161 rte_sws_7504 rte_sws_7295 rte_sws_7288 rte_sws_7284
[BSW00305] Self-defined data types naming convention	rte_sws_1150 rte_sws_3714 rte_sws_3733 rte_sws_2301 rte_sws_3731 rte_sws_1055
[BSW00307] Global variables naming convention	rte_sws_1171 rte_sws_3712 rte_sws_7284
[BSW00308] Definition of global data	rte_sws_3786 rte_sws_7121 rte_sws_7921 rte_sws_7502

[BSW00310] API naming convention	rte_sws_1071 rte_sws_1072 rte_sws_2631 rte_sws_1206 rte_sws_1083 rte_sws_2725 rte_sws_1091 rte_sws_7394 rte_sws_1092 rte_sws_1102 rte_sws_1111 rte_sws_1118 rte_sws_1252 rte_sws_3928 rte_sws_3929 rte_sws_3741 rte_sws_3744 rte_sws_5509 rte_sws_3800 rte_sws_3550 rte_sws_3553 rte_sws_3560 rte_sws_3565 rte_sws_1120 rte_sws_1123 rte_sws_7367 rte_sws_7390 rte_sws_2569 rte_sws_7556
[BSW00312] Shared code shall be reentrant	rte_sws_1012
[BSW00326] Transition from ISRs to OS tasks	rte_sws_3600 rte_sws_3594 rte_sws_3530 rte_sws_3531 rte_sws_3532
[BSW00327] Error values naming convention	rte_sws_1058 rte_sws_1060 rte_sws_1064 rte_sws_1317 rte_sws_1061 rte_sws_1065 rte_sws_7384 rte_sws_7655 rte_sws_2571 rte_sws_7289 rte_sws_7290 rte_sws_7562 rte_sws_7563
[BSW00330] Usage of macros / inline functions instead of functions	rte_sws_1274
[BSW00336] Shutdown interface	rte_sws_7274 rte_sws_7275 rte_sws_7277
[BSW00337] Classification of errors	rte_sws_6630 rte_sws_7676 rte_sws_6631 rte_sws_6632 rte_sws_6633 rte_sws_6634 rte_sws_7684 rte_sws_6635 rte_sws_6637 rte_sws_7675 rte_sws_7685 rte_sws_7682 rte_sws_7683
[BSW00338] Detection and Reporting of development errors	rte_sws_6630 rte_sws_7676 rte_sws_6631 rte_sws_6632 rte_sws_6633 rte_sws_6634 rte_sws_7684 rte_sws_6635 rte_sws_6637 rte_sws_7675 rte_sws_7685 rte_sws_7682 rte_sws_7683
[BSW00342] Usage of source code and object code	rte_sws_7511
[BSW00345] Pre–compile–time configuration	rte_sws_5103
[BSW00346] Basic set of module files	rte_sws_6638
[BSW00347] Naming separation of different instances of BSW drivers	rte_sws_6535 rte_sws_6536 rte_sws_6532 rte_sws_7250 rte_sws_7253 rte_sws_7255 rte_sws_7260 rte_sws_7694 rte_sws_7263 rte_sws_7266 rte_sws_7093 rte_sws_7282 rte_sws_7504 rte_sws_7295 rte_sws_7528

[BSW00353] Platform specific type header	rte_sws_7641 rte_sws_1163 rte_sws_7104
[BSW00397] Pre-compile-time parameters	rte_sws_5103
[BSW00399] Loadable Post-build time parameters	rte_sws_5104
[BSW004] Version check	rte_sws_7692
[BSW00400] Selectable Post-build time parameters	rte_sws_5104
[BSW00405] Reference to multiple configuration sets	rte_sws_6544 rte_sws_6545
[BSW00407] Get version info keyword	rte_sws_7278 rte_sws_7279 rte_sws_7280 rte_sws_7281
[BSW00415] User dependent include files	rte_sws_7501 rte_sws_7504 rte_sws_7505 rte_sws_7506 rte_sws_7510 rte_sws_7503 rte_sws_7295 rte_sws_7296 rte_sws_7500
[BSW00447] Standardizing Include file structure of BSW Modules Implementing Autosar Service	rte_sws_7120
[BSW007] HIS MISRA C	rte_sws_7086 rte_sws_3715 rte_sws_1168 rte_sws_7300
[BSW101] Initialization interface	rte_sws_7270 rte_sws_7271 rte_sws_7273
[BSW161] Microcontroller abstraction	rte_sws_2734
[RTE00003] Tracing of sender-receiver communication	rte_sws_1357 rte_sws_1238 rte_sws_1240 rte_sws_1241 rte_sws_3814 rte_sws_7639 rte_sws_1242
[RTE00004] Tracing of client-server communication	rte_sws_1357 rte_sws_1238 rte_sws_1240 rte_sws_1241 rte_sws_3814 rte_sws_7639 rte_sws_1242
[RTE00005] Support for 'trace' build	rte_sws_3607 rte_sws_1320 rte_sws_1322 rte_sws_1323 rte_sws_1327 rte_sws_1328 rte_sws_5093 rte_sws_5091 rte_sws_5092 rte_sws_5106 rte_sws_8000
[RTE00008] VFB tracing configuration	rte_sws_3607 rte_sws_1320 rte_sws_1236 rte_sws_1321 rte_sws_1322 rte_sws_1323 rte_sws_1324 rte_sws_1325 rte_sws_5093 rte_sws_5091 rte_sws_5092 rte_sws_8000

[RTE00011] Support for multiple application software component instances	rte_sws_2001 rte_sws_2008 rte_sws_2009 rte_sws_2002 rte_sws_3015 rte_sws_2015 rte_sws_1148 rte_sws_1012 rte_sws_1013 rte_sws_3806 rte_sws_3793 rte_sws_7132 rte_sws_3718 rte_sws_3719 rte_sws_1349 rte_sws_3720 rte_sws_3721 rte_sws_3716 rte_sws_3717 rte_sws_7225 rte_sws_3722 rte_sws_3711 rte_sws_1016
[RTE00012] Multiple instantiated AUTOSAR software-components delivered as binary code shall share code	rte_sws_3015 rte_sws_2015 rte_sws_1007
[RTE00013] Per-instance memory	rte_sws_3790 rte_sws_7045 rte_sws_7161 rte_sws_2303 rte_sws_2304 rte_sws_7133 rte_sws_3782 rte_sws_7134 rte_sws_7135 rte_sws_2305 rte_sws_7182 rte_sws_8304 rte_sws_7183 rte_sws_7184 rte_sws_5062 rte_sws_2301 rte_sws_2302 rte_sws_8303
[RTE00017] Rejection of inconsistent component implementations	rte_sws_1004 rte_sws_2751 rte_sws_1276 rte_sws_7123 rte_sws_7510

[RTE00018] Rejection of invalid configurations	rte_sws_1358 rte_sws_7402 rte_sws_3526 rte_sws_3010 rte_sws_7007 rte_sws_7403 rte_sws_3012 rte_sws_3018 rte_sws_3014 rte_sws_3605 rte_sws_7170 rte_sws_7101 rte_sws_3527 rte_sws_2733 rte_sws_2706 rte_sws_2500 rte_sws_2662 rte_sws_2663 rte_sws_2664 rte_sws_7157 rte_sws_7686 rte_sws_4525 rte_sws_7642 rte_sws_7681 rte_sws_3019 rte_sws_2750 rte_sws_2670 rte_sws_2724 rte_sws_2738 rte_sws_3951 rte_sws_3970 rte_sws_7190 rte_sws_7191 rte_sws_7654 rte_sws_7810 rte_sws_7811 rte_sws_7812 rte_sws_7670 rte_sws_5116 rte_sws_7803 rte_sws_7808 rte_sws_7809 rte_sws_7181 rte_sws_7075 rte_sws_7516 rte_sws_7192 rte_sws_7006 rte_sws_5506 rte_sws_3755 rte_sws_2526 rte_sws_2723 rte_sws_7662 rte_sws_3764 rte_sws_2529 rte_sws_2579 rte_sws_5111 rte_sws_5054 rte_sws_3817 rte_sws_3820 rte_sws_3823 rte_sws_3826 rte_sws_3831 rte_sws_7545 rte_sws_7548 rte_sws_7039 rte_sws_7549 rte_sws_7610 rte_sws_3594 rte_sws_7353 rte_sws_7621 rte_sws_7343 rte_sws_7356 rte_sws_7357 rte_sws_7667 rte_sws_2254 rte_sws_7044 rte_sws_3950 rte_sws_7564 rte_sws_2730 rte_sws_7057 rte_sws_7524 rte_sws_7005 rte_sws_7028 rte_sws_2051 rte_sws_2009 rte_sws_2204 rte_sws_7347 rte_sws_6502 rte_sws_6503 rte_sws_6504 rte_sws_6505 rte_sws_6547 rte_sws_6548 rte_sws_6508 rte_sws_6509 rte_sws_6511 rte_sws_6610 rte_sws_6613 rte_sws_5149 rte_sws_7640 rte_sws_3851 rte_sws_3813 rte_sws_7175 rte_sws_7135 rte_sws_1287 rte_sws_1313 rte_sws_7638 rte_sws_6719 rte_sws_6724 rte_sws_7026 rte_sws_7588
[RTE00019] RTE is the communication infrastructure	rte_sws_6000 rte_sws_6011 rte_sws_5500 rte_sws_4527 rte_sws_6023 rte_sws_4526 rte_sws_6024 rte_sws_3760 rte_sws_3761 rte_sws_3762 rte_sws_4515 rte_sws_4516 rte_sws_7662 rte_sws_4520 rte_sws_4522 rte_sws_8001 rte_sws_8002 rte_sws_2527 rte_sws_2528 rte_sws_3769 rte_sws_1231 rte_sws_5063 rte_sws_3007 rte_sws_3008 rte_sws_3000 rte_sws_3001 rte_sws_3002 rte_sws_3775 rte_sws_2612 rte_sws_2610 rte_sws_5084 rte_sws_3004 rte_sws_3005 rte_sws_3776 rte_sws_5065 rte_sws_2611 rte_sws_5085 rte_sws_1264 rte_sws_3795 rte_sws_3796
[RTE00020] Access to OS	rte_sws_2250
[RTE00021] Per-ECU RTE customization	rte_sws_5000 rte_sws_1316
[RTE00022] Interaction with call-backs	rte_sws_1165

[RTE00023] RTE Overheads	rte_sws_5053
[RTE00024] Source-code AUTOSAR software components	rte_sws_1315 rte_sws_1000 rte_sws_7120 rte_sws_1195
[RTE00025] Static communication	rte_sws_6026
[RTE00027] VFB to RTE mapping shall be semantic preserving	rte_sws_2200 rte_sws_2201 rte_sws_1274
[RTE00028] 1:n Sender-receiver communication	rte_sws_6023 rte_sws_4526 rte_sws_6024 rte_sws_1071 rte_sws_7824 rte_sws_7826 rte_sws_2635 rte_sws_1082 rte_sws_1072 rte_sws_7825 rte_sws_7827 rte_sws_2633 rte_sws_2631 rte_sws_1091 rte_sws_7394 rte_sws_1092 rte_sws_1135
[RTE00029] n:1 Client-server communication	rte_sws_5110 rte_sws_5163 rte_sws_6019 rte_sws_4519 rte_sws_4517 rte_sws_3763 rte_sws_3770 rte_sws_3767 rte_sws_3768 rte_sws_2579 rte_sws_3769 rte_sws_1102 rte_sws_1109 rte_sws_1133 rte_sws_1359 rte_sws_1166 rte_sws_7023 rte_sws_7024 rte_sws_7025 rte_sws_7026 rte_sws_7027 rte_sws_5193
[RTE00031] Multiple Runnable Entities	rte_sws_2202 rte_sws_1126 rte_sws_1132 rte_sws_1016 rte_sws_6713 rte_sws_1130
[RTE00032] Data consistency mechanisms	rte_sws_3514 rte_sws_3500 rte_sws_3504 rte_sws_5164 rte_sws_3595 rte_sws_3503 rte_sws_7005 rte_sws_3516 rte_sws_3517 rte_sws_3519 rte_sws_2740 rte_sws_2741 rte_sws_2743 rte_sws_2744 rte_sws_2745 rte_sws_2746 rte_sws_1122 rte_sws_3739 rte_sws_3740 rte_sws_3812
[RTE00033] Serialized execution of Server Runnable Entities	rte_sws_4515 rte_sws_4518 rte_sws_4522 rte_sws_8001 rte_sws_8002 rte_sws_2527 rte_sws_2528 rte_sws_2529 rte_sws_2530 rte_sws_7008
[RTE00036] Assignment to OS Applications	rte_sws_7347
[RTE00045] Standardized VFB tracing interface	rte_sws_1319 rte_sws_1250 rte_sws_1251 rte_sws_1321 rte_sws_1326 rte_sws_1238 rte_sws_1239 rte_sws_1240 rte_sws_1241 rte_sws_3814 rte_sws_7639 rte_sws_1242 rte_sws_1243 rte_sws_1244 rte_sws_1245 rte_sws_1246 rte_sws_1247 rte_sws_1248 rte_sws_1249

[RTE00046] Support for 'Executable Entity runs inside' Exclusive Areas	rte_sws_3500 rte_sws_3515 rte_sws_7522 rte_sws_7523 rte_sws_7524 rte_sws_2740 rte_sws_2741 rte_sws_2743 rte_sws_2744 rte_sws_2745 rte_sws_2746 rte_sws_1120 rte_sws_1122 rte_sws_1123 rte_sws_7250 rte_sws_7251 rte_sws_7252 rte_sws_7578 rte_sws_7579 rte_sws_7253 rte_sws_7254
[RTE00048] RTE Generator input	rte_sws_5001
[RTE00049] Construction of task bodies	rte_sws_7516 rte_sws_2251 rte_sws_2254 rte_sws_2204
[RTE00051] RTE API mapping	rte_sws_3014 rte_sws_3605 rte_sws_7170 rte_sws_2679 rte_sws_2730 rte_sws_1269 rte_sws_1148 rte_sws_1274 rte_sws_3706 rte_sws_3707 rte_sws_3837 rte_sws_1156 rte_sws_1153 rte_sws_1146 rte_sws_2619 rte_sws_2613 rte_sws_3602 rte_sws_2614 rte_sws_2615 rte_sws_3603 rte_sws_1354 rte_sws_1355 rte_sws_1280 rte_sws_1281 rte_sws_2632 rte_sws_1282 rte_sws_1283 rte_sws_1284 rte_sws_1285 rte_sws_1286 rte_sws_1287 rte_sws_2676 rte_sws_2677 rte_sws_2678 rte_sws_1289 rte_sws_7396 rte_sws_1313 rte_sws_7395 rte_sws_1288 rte_sws_1290 rte_sws_1293 rte_sws_1294 rte_sws_6639 rte_sws_1296 rte_sws_1297 rte_sws_1298 rte_sws_1312 rte_sws_1299 rte_sws_1119 rte_sws_1300 rte_sws_3927 rte_sws_3952 rte_sws_3930 rte_sws_1301 rte_sws_1268 rte_sws_1302 rte_sws_3746 rte_sws_3747 rte_sws_5510 rte_sws_5511 rte_sws_3801 rte_sws_1303 rte_sws_1304 rte_sws_3555 rte_sws_1305 rte_sws_3562 rte_sws_1306 rte_sws_3567 rte_sws_1307 rte_sws_1123 rte_sws_1308 rte_sws_1276 rte_sws_3718 rte_sws_3719 rte_sws_1349 rte_sws_3720 rte_sws_3721 rte_sws_3716 rte_sws_3717 rte_sws_7225 rte_sws_3723 rte_sws_3733 rte_sws_2608 rte_sws_2588 rte_sws_1363 rte_sws_1364 rte_sws_2607 rte_sws_1365 rte_sws_1366 rte_sws_3734 rte_sws_2666 rte_sws_2589 rte_sws_7136 rte_sws_2301 rte_sws_2302 rte_sws_3739 rte_sws_3740 rte_sws_3812 rte_sws_2616 rte_sws_2617 rte_sws_3799 rte_sws_3731 rte_sws_7137 rte_sws_7138 rte_sws_7677 rte_sws_3730 rte_sws_2620 rte_sws_2621 rte_sws_1055 rte_sws_3726 rte_sws_2618 rte_sws_1343 rte_sws_1342 rte_sws_1053 rte_sws_3835 rte_sws_3949 rte_sws_3725 rte_sws_3752 rte_sws_2623 rte_sws_3791 rte_sws_7226 rte_sws_7227 rte_sws_7228 rte_sws_1309 rte_sws_1310 rte_sws_1159 rte_sws_1266 rte_sws_1197 rte_sws_1132 rte_sws_6713 rte_sws_7291

[RTE00052] Initialization and finalization of components	rte_sws_7046 rte_sws_3852 rte_sws_2503 rte_sws_2562 rte_sws_2707 rte_sws_2564
[RTE00055] Use of global namespace	rte_sws_1171 rte_sws_7104 rte_sws_7110 rte_sws_7111 rte_sws_6706 rte_sws_6707 rte_sws_6708 rte_sws_7114 rte_sws_7144 rte_sws_7115 rte_sws_7116 rte_sws_7117 rte_sws_7118 rte_sws_7119 rte_sws_7145 rte_sws_7146 rte_sws_7109 rte_sws_7148 rte_sws_7149 rte_sws_7166 rte_sws_7036 rte_sws_7037 rte_sws_7162 rte_sws_7163 rte_sws_7284
[RTE00059] RTE API passes 'in' primitive data types by value	rte_sws_1017 rte_sws_7661 rte_sws_7083 rte_sws_1020 rte_sws_7084 rte_sws_7069 rte_sws_8300 rte_sws_7070 rte_sws_7071 rte_sws_7072 rte_sws_7073 rte_sws_7074 rte_sws_7076 rte_sws_7077 rte_sws_7078 rte_sws_7079 rte_sws_7080 rte_sws_7081
[RTE00060] RTE API shall pass 'in' composite data types by reference	rte_sws_1018 rte_sws_5107 rte_sws_7086 rte_sws_7082 rte_sws_5108 rte_sws_7084
[RTE00061] 'in/out' and 'out' parameters	rte_sws_1017 rte_sws_1018 rte_sws_5107 rte_sws_7661 rte_sws_1019 rte_sws_7082 rte_sws_5108 rte_sws_7083 rte_sws_1020 rte_sws_5109 rte_sws_7084
[RTE00062] Local access to basic software components	rte_sws_2051
[RTE00064] AUTOSAR Methodology	see chapter 3
[RTE00065] Deterministic generation	rte_sws_2514 rte_sws_5150
[RTE00068] Signal initial values	rte_sws_7642 rte_sws_2517 rte_sws_7668 rte_sws_7046 rte_sws_3852 rte_sws_5078
[RTE00069] Communication timeouts	rte_sws_6002 rte_sws_6013 rte_sws_3754 rte_sws_3758 rte_sws_3759 rte_sws_3763 rte_sws_3770 rte_sws_3773 rte_sws_3771 rte_sws_3772 rte_sws_3767 rte_sws_3768 rte_sws_7056 rte_sws_7060 rte_sws_7059 rte_sws_1064 rte_sws_1095 rte_sws_1107 rte_sws_1114
[RTE00070] Invocation order of Runnable Entities	rte_sws_2207

[RTE00072] Activation of Runnable Entities	rte_sws_3526 rte_sws_7403 rte_sws_3527 rte_sws_7515 rte_sws_7575 rte_sws_7178 rte_sws_2697 rte_sws_7061 rte_sws_3530 rte_sws_3531 rte_sws_3532 rte_sws_1292 rte_sws_3523 rte_sws_3520 rte_sws_3524 rte_sws_2203 rte_sws_1131 rte_sws_7177 rte_sws_2512 rte_sws_1133 rte_sws_1359 rte_sws_1166 rte_sws_7023 rte_sws_7024 rte_sws_7025 rte_sws_7026 rte_sws_7027 rte_sws_5193 rte_sws_1135 rte_sws_1137 rte_sws_2758 rte_sws_7207 rte_sws_7208 rte_sws_7379
[RTE00073] Atomic transport of Data Elements	rte_sws_4527
[RTE00075] API for accessing per-instance memory	rte_sws_1118 rte_sws_1119
[RTE00077] Instantiation of per-instance memory	rte_sws_3790 rte_sws_7045 rte_sws_7161 rte_sws_2303 rte_sws_2304 rte_sws_7133 rte_sws_3782 rte_sws_2305 rte_sws_7182 rte_sws_8304 rte_sws_7183 rte_sws_7184 rte_sws_5062 rte_sws_8303
[RTE00078] Support for Data Element Invalidation	rte_sws_8004 rte_sws_5024 rte_sws_2594 rte_sws_2702 rte_sws_1206 rte_sws_1282 rte_sws_1231 rte_sws_5063 rte_sws_2626 rte_sws_3800 rte_sws_3801 rte_sws_3802 rte_sws_5064 rte_sws_3778 rte_sws_2599 rte_sws_2600 rte_sws_2603 rte_sws_2629 rte_sws_8501 rte_sws_2607 rte_sws_2666 rte_sws_2589 rte_sws_2590 rte_sws_2609
[RTE00079] Single asynchronous client-server interaction	rte_sws_3765 rte_sws_3766 rte_sws_3771 rte_sws_3772 rte_sws_2658 rte_sws_1105 rte_sws_1109 rte_sws_1133 rte_sws_1359 rte_sws_1166 rte_sws_7023 rte_sws_7024 rte_sws_7025 rte_sws_7026 rte_sws_7027 rte_sws_5193
[RTE00080] Multiple requests of servers	rte_sws_4516 rte_sws_4520
[RTE00082] Standardized communication protocol	rte_sws_2649 rte_sws_7346 rte_sws_2651 rte_sws_2652 rte_sws_2653 rte_sws_2579 rte_sws_5066 rte_sws_2654 rte_sws_2655 rte_sws_2656 rte_sws_2657 rte_sws_5067 rte_sws_5054 rte_sws_5055 rte_sws_6028 rte_sws_5056 rte_sws_5057 rte_sws_5058 rte_sws_5059
[RTE00083] Optimization for source-code components	rte_sws_1274 rte_sws_1152
[RTE00084] Support infrastructural errors	rte_sws_2593 rte_sws_1318

[RTE00087] Software Module Header File generation	rte_sws_1274 rte_sws_1000 rte_sws_3786 rte_sws_1004 rte_sws_1006 rte_sws_7131 rte_sws_7924 rte_sws_5078 rte_sws_7127 rte_sws_1132 rte_sws_6713 rte_sws_6703 rte_sws_6704 rte_sws_6705
[RTE00089] Independent access to interface elements	rte_sws_6008
[RTE00091] Inter-ECU Marshalling	rte_sws_4504 rte_sws_4505 rte_sws_4506 rte_sws_4507 rte_sws_4508 rte_sws_2557 rte_sws_5081 rte_sws_5173 rte_sws_4527
[RTE00092] Implementation of VFB model waitpoints	rte_sws_1358 rte_sws_7402 rte_sws_3010 rte_sws_3018 rte_sws_2740 rte_sws_2741 rte_sws_2743 rte_sws_2744 rte_sws_2745 rte_sws_2746
[RTE00094] Communication and Resource Errors	rte_sws_2524 rte_sws_2525 rte_sws_2721 rte_sws_1318 rte_sws_2571 rte_sws_1034 rte_sws_7820 rte_sws_7822 rte_sws_7821 rte_sws_7823 rte_sws_2674 rte_sws_1207 rte_sws_1339 rte_sws_1084 rte_sws_7636 rte_sws_3774 rte_sws_7637 rte_sws_1086 rte_sws_7658 rte_sws_7652 rte_sws_2727 rte_sws_2728 rte_sws_3853 rte_sws_2729 rte_sws_7659 rte_sws_1093 rte_sws_7690 rte_sws_7673 rte_sws_2598 rte_sws_1094 rte_sws_1095 rte_sws_2572 rte_sws_7665 rte_sws_1103 rte_sws_1104 rte_sws_1105 rte_sws_1106 rte_sws_1107 rte_sws_7656 rte_sws_1112 rte_sws_1113 rte_sws_8301 rte_sws_1114 rte_sws_3606 rte_sws_7657 rte_sws_8302 rte_sws_2578 rte_sws_3803 rte_sws_2602 rte_sws_7691 rte_sws_7374 rte_sws_7375 rte_sws_7650 rte_sws_7376 rte_sws_7660 rte_sws_7651 rte_sws_7392 rte_sws_7393 rte_sws_1261 rte_sws_1262 rte_sws_1259 rte_sws_1260 rte_sws_7258
[RTE00098] Explicit Sending	rte_sws_6011 rte_sws_6016 rte_sws_1071
[RTE00099] Decoupling of interrupts	rte_sws_3600 rte_sws_3594 rte_sws_3530 rte_sws_3531 rte_sws_3532
[RTE00100] Compiler independent API	rte_sws_1314
[RTE00107] Support for INFORMATION_TYPE attribute	rte_sws_6010 rte_sws_4500 rte_sws_2516 rte_sws_2518 rte_sws_2520 rte_sws_2521 rte_sws_2522 rte_sws_2523 rte_sws_2524 rte_sws_2525 rte_sws_2718 rte_sws_2719 rte_sws_2720 rte_sws_2721 rte_sws_2571 rte_sws_2572 rte_sws_7665 rte_sws_1135 rte_sws_1137 rte_sws_2758
[RTE00108] Support for INIT_VALUE attribute	rte_sws_4525 rte_sws_7642 rte_sws_7681 rte_sws_6009 rte_sws_4501 rte_sws_4502 rte_sws_2517 rte_sws_7668 rte_sws_1268 rte_sws_5078 rte_sws_7680

[RTE00109] Support for RECEIVE_MODE attribute	rte_sws_3018 rte_sws_6002 rte_sws_6012 rte_sws_2519
[RTE00110] Support for BUFFERING attribute	rte_sws_2515 rte_sws_2522 rte_sws_2523 rte_sws_2524 rte_sws_2525 rte_sws_2526 rte_sws_2719 rte_sws_2720 rte_sws_2721 rte_sws_2723 rte_sws_2527 rte_sws_2529 rte_sws_2530 rte_sws_7008 rte_sws_2571 rte_sws_2572 rte_sws_7665
[RTE00111] Support for CLIENT_MODE attribute	rte_sws_1293 rte_sws_1294 rte_sws_6639
[RTE00115] API for data consistency mechanism	rte_sws_1120 rte_sws_1307 rte_sws_1122 rte_sws_1308
[RTE00116] RTE Initialization and finalization	rte_sws_2538 rte_sws_2535 rte_sws_2536 rte_sws_7586 rte_sws_7046 rte_sws_3852 rte_sws_2544 rte_sws_2569 rte_sws_2585 rte_sws_2570 rte_sws_2584 rte_sws_7270
[RTE00121] Support for FILTER attribute	rte_sws_5503 rte_sws_5500 rte_sws_5501
[RTE00122] Support for Transmission Acknowledgement	rte_sws_5504 rte_sws_3754 rte_sws_3756 rte_sws_3757 rte_sws_3604 rte_sws_3758 rte_sws_8017 rte_sws_8043 rte_sws_8018 rte_sws_8020 rte_sws_8044 rte_sws_8021 rte_sws_8022 rte_sws_8045 rte_sws_8023 rte_sws_1080 rte_sws_2673 rte_sws_1083 rte_sws_1283 rte_sws_1284 rte_sws_1285 rte_sws_1286 rte_sws_1287 rte_sws_7634 rte_sws_7635 rte_sws_1084 rte_sws_7636 rte_sws_3774 rte_sws_7637 rte_sws_1086 rte_sws_7658 rte_sws_7652 rte_sws_2725 rte_sws_2676 rte_sws_2677 rte_sws_2678 rte_sws_2727 rte_sws_2729 rte_sws_7659 rte_sws_7367 rte_sws_7646 rte_sws_7647 rte_sws_7648 rte_sws_7649 rte_sws_7374 rte_sws_7375 rte_sws_7650 rte_sws_7376 rte_sws_7660 rte_sws_7651 rte_sws_3002 rte_sws_3775 rte_sws_2612 rte_sws_5084 rte_sws_3005 rte_sws_3776 rte_sws_5065 rte_sws_5085 rte_sws_1137 rte_sws_2758 rte_sws_7379 rte_sws_7286 rte_sws_7557 rte_sws_7558 rte_sws_7560 rte_sws_7561 rte_sws_7055
[RTE00123] Forwarding of application level server errors	rte_sws_2593 rte_sws_2576 rte_sws_1103 rte_sws_2577 rte_sws_2578
[RTE00124] APIs for application level server errors	rte_sws_2573 rte_sws_2575 rte_sws_1103 rte_sws_1130

[RTE00125] Rejection of '1:n' communication with the Transmission Acknowledgment	rte_sws_5506
[RTE00126] C support	rte_sws_1005 rte_sws_3709 rte_sws_3710 rte_sws_7124 rte_sws_7125 rte_sws_7126 rte_sws_7678 rte_sws_7923 rte_sws_3724 rte_sws_1169 rte_sws_1167 rte_sws_1162 rte_sws_7507 rte_sws_7508 rte_sws_7509 rte_sws_7297 rte_sws_7298 rte_sws_7299
[RTE00128] Implicit Reception	rte_sws_7007 rte_sws_3012 rte_sws_6000 rte_sws_6001 rte_sws_6004 rte_sws_6011 rte_sws_3954 rte_sws_3598 rte_sws_3955 rte_sws_3599 rte_sws_7062 rte_sws_3956 rte_sws_7687 rte_sws_7020 rte_sws_7063 rte_sws_7064 rte_sws_7652 rte_sws_3741 rte_sws_1268
[RTE00129] Implicit Sending	rte_sws_7007 rte_sws_6011 rte_sws_3570 rte_sws_3571 rte_sws_3572 rte_sws_3573 rte_sws_3574 rte_sws_3954 rte_sws_3598 rte_sws_3955 rte_sws_3953 rte_sws_7062 rte_sws_7041 rte_sws_3957 rte_sws_7688 rte_sws_7021 rte_sws_7065 rte_sws_7066 rte_sws_7067 rte_sws_7068 rte_sws_3744 rte_sws_3746 rte_sws_5509 rte_sws_7367 rte_sws_7646 rte_sws_7647 rte_sws_7648 rte_sws_7649 rte_sws_7374 rte_sws_7375 rte_sws_7650 rte_sws_7376 rte_sws_7660 rte_sws_7651
[RTE00131] n:1 Sender-receiver communication	rte_sws_2670 rte_sws_2724 rte_sws_3760 rte_sws_3761 rte_sws_3762 rte_sws_1071 rte_sws_7824 rte_sws_7826 rte_sws_2635 rte_sws_1072 rte_sws_7825 rte_sws_7827 rte_sws_2633 rte_sws_2631 rte_sws_1091 rte_sws_7394 rte_sws_1092 rte_sws_1135
[RTE00133] Concurrent invocation of Runnable Entities	rte_sws_7007 rte_sws_2697 rte_sws_3523
[RTE00134] Runnable Entity categories supported by the RTE	rte_sws_6003 rte_sws_6007 rte_sws_3574 rte_sws_3954 rte_sws_7062
[RTE00137] API for mismatched ports	rte_sws_1368 rte_sws_1369 rte_sws_1370
[RTE00138] C++ support	Only partially supported by: rte_sws_1005 rte_sws_3709 rte_sws_3710 rte_sws_7124 rte_sws_7125 rte_sws_7126 rte_sws_1011 rte_sws_7507 rte_sws_7508 rte_sws_7509 rte_sws_7297 rte_sws_7298 rte_sws_7299

[RTE00139] Support for unconnected ports	rte_sws_3019 rte_sws_2750 rte_sws_7669 rte_sws_7667 rte_sws_7668 rte_sws_3978 rte_sws_5101 rte_sws_3980 rte_sws_5102 rte_sws_5170 rte_sws_2749 rte_sws_7655 rte_sws_1329 rte_sws_1330 rte_sws_7663 rte_sws_1331 rte_sws_1344 rte_sws_1332 rte_sws_3783 rte_sws_7378 rte_sws_1346 rte_sws_1347 rte_sws_3784 rte_sws_3785 rte_sws_2638 rte_sws_2639 rte_sws_2640 rte_sws_2641 rte_sws_2642 rte_sws_1333 rte_sws_1334 rte_sws_7658 rte_sws_7659 rte_sws_7690 rte_sws_7665 rte_sws_7656 rte_sws_7657 rte_sws_7691 rte_sws_7660 rte_sws_5099
[RTE00140] Binary-code AUTOSAR software components	rte_sws_1315 rte_sws_1000 rte_sws_7120 rte_sws_1195
[RTE00141] Explicit Reception	rte_sws_6011 rte_sws_1072 rte_sws_1091 rte_sws_7394 rte_sws_1092 rte_sws_7673
[RTE00142] Inter-RunnableVariables	rte_sws_7007 rte_sws_3589 rte_sws_7187 rte_sws_3516 rte_sws_3517 rte_sws_3582 rte_sws_3583 rte_sws_3584 rte_sws_7022 rte_sws_3519 rte_sws_3580 rte_sws_3550 rte_sws_1303 rte_sws_3581 rte_sws_3553 rte_sws_1304 rte_sws_3555 rte_sws_3560 rte_sws_1305 rte_sws_3562 rte_sws_3565 rte_sws_1306 rte_sws_3567 rte_sws_3569 rte_sws_2636 rte_sws_1350 rte_sws_1351
[RTE00143] Mode switches	rte_sws_2706 rte_sws_2500 rte_sws_2662 rte_sws_2663 rte_sws_2664 rte_sws_7157 rte_sws_7155 rte_sws_2503 rte_sws_2504 rte_sws_7150 rte_sws_7151 rte_sws_7173 rte_sws_2667 rte_sws_2661 rte_sws_7152 rte_sws_2562 rte_sws_7153 rte_sws_2707 rte_sws_2708 rte_sws_2564 rte_sws_7154 rte_sws_2563 rte_sws_2587 rte_sws_2665 rte_sws_2668 rte_sws_2630 rte_sws_2669 rte_sws_7533 rte_sws_7535 rte_sws_7564 rte_sws_2544 rte_sws_2546 rte_sws_2679 rte_sws_7559 rte_sws_2730 rte_sws_7056 rte_sws_7060 rte_sws_7057 rte_sws_7058 rte_sws_7059 rte_sws_2634 rte_sws_2631 rte_sws_2675 rte_sws_2512 rte_sws_7259
[RTE00144] Mode switch notification via AUTOSAR interfaces	rte_sws_2738 rte_sws_7155 rte_sws_2544 rte_sws_2549 rte_sws_2508 rte_sws_2566 rte_sws_2624 rte_sws_2567 rte_sws_2546 rte_sws_7540 rte_sws_2627 rte_sws_2659 rte_sws_3858 rte_sws_7640 rte_sws_2568 rte_sws_3859 rte_sws_2628 rte_sws_2731 rte_sws_7666 rte_sws_2660 rte_sws_2732 rte_sws_8500 rte_sws_8503 rte_sws_8504 rte_sws_8505 rte_sws_8506 rte_sws_7262 rte_sws_8509 rte_sws_8510
[RTE00145] Compatibility mode	rte_sws_1257 rte_sws_3794 rte_sws_1279 rte_sws_1326 rte_sws_1277 rte_sws_1151 rte_sws_1216 rte_sws_1234

[RTE00146] Vendor mode	rte_sws_1234
[RTE00147] Support for communication infrastructure time-out notification	rte_sws_5020 rte_sws_5021 rte_sws_3759 rte_sws_5022 rte_sws_8004 rte_sws_2703 rte_sws_2599 rte_sws_2600 rte_sws_2604 rte_sws_2629 rte_sws_8501 rte_sws_2610 rte_sws_2611 rte_sws_2607 rte_sws_2666 rte_sws_2589 rte_sws_2590 rte_sws_2609
[RTE00148] Support 'Specification of Memory Mapping'	rte_sws_3788 rte_sws_5088 rte_sws_7589 rte_sws_7047 rte_sws_7048 rte_sws_7049 rte_sws_7050 rte_sws_7051 rte_sws_7052 rte_sws_7053 rte_sws_7592 rte_sws_7590 rte_sws_7591 rte_sws_5089 rte_sws_7194 rte_sws_7195 rte_sws_7593 rte_sws_7594 rte_sws_7595 rte_sws_7596
[RTE00149] Support 'Specification of Compiler Abstraction'	rte_sws_3787 rte_sws_7641 rte_sws_7194 rte_sws_7195 rte_sws_7593 rte_sws_7594 rte_sws_7595 rte_sws_7596
[RTE00150] Support 'Specification of Platform Types'	rte_sws_7641
[RTE00151] Support RTE relevant requirements of the 'General Requirements on Basic Software Modules'	see [BSW...] entries in this table
[RTE00152] Support for port-defined argument values	rte_sws_1360 rte_sws_1166
[RTE00153] Support for Measurement	rte_sws_3951 rte_sws_3900 rte_sws_3972 rte_sws_3973 rte_sws_3974 rte_sws_7344 rte_sws_3901 rte_sws_3975 rte_sws_3976 rte_sws_3977 rte_sws_7349 rte_sws_6700 rte_sws_6701 rte_sws_3902 rte_sws_7160 rte_sws_7174 rte_sws_7197 rte_sws_7198 rte_sws_3978 rte_sws_5101 rte_sws_3980 rte_sws_5102 rte_sws_5170 rte_sws_3979 rte_sws_3903 rte_sws_3904 rte_sws_3950 rte_sws_3981 rte_sws_3982 rte_sws_5120 rte_sws_5121 rte_sws_5172 rte_sws_6702 rte_sws_5122 rte_sws_5123 rte_sws_5124 rte_sws_5125 rte_sws_5136 rte_sws_5168 rte_sws_5176 rte_sws_5174 rte_sws_5169 rte_sws_5175 rte_sws_6726 rte_sws_5087

[RTE00154] Support for Calibration	rte_sws_3970 rte_sws_5145 rte_sws_3958 rte_sws_7186 rte_sws_7029 rte_sws_3959 rte_sws_5112 rte_sws_7096 rte_sws_7185 rte_sws_7030 rte_sws_7033 rte_sws_7034 rte_sws_7035 rte_sws_3905 rte_sws_3906 rte_sws_3907 rte_sws_3971 rte_sws_3909 rte_sws_3942 rte_sws_3910 rte_sws_3943 rte_sws_3911 rte_sws_3912 rte_sws_5194 rte_sws_3968 rte_sws_3913 rte_sws_3947 rte_sws_3936 rte_sws_3914 rte_sws_3948 rte_sws_3915 rte_sws_3935 rte_sws_3916 rte_sws_3908 rte_sws_3922 rte_sws_3960 rte_sws_3932 rte_sws_3933 rte_sws_3934 rte_sws_3961 rte_sws_3962 rte_sws_3963 rte_sws_3964 rte_sws_3965 rte_sws_7693 rte_sws_3835 rte_sws_3949
[RTE00155] API to access calibration parameters	rte_sws_1252 rte_sws_1300 rte_sws_3927 rte_sws_3952 rte_sws_3928 rte_sws_3929 rte_sws_3930 rte_sws_3835 rte_sws_3949 rte_sws_7093 rte_sws_7094 rte_sws_7095
[RTE00156] Support for different calibration data emulation methods	rte_sws_3970 rte_sws_5145 rte_sws_3905 rte_sws_3906 rte_sws_3971 rte_sws_3909 rte_sws_3942 rte_sws_3910 rte_sws_3943 rte_sws_3911 rte_sws_3968 rte_sws_3913 rte_sws_3947 rte_sws_3936 rte_sws_3914 rte_sws_3948 rte_sws_3915 rte_sws_3935 rte_sws_3916 rte_sws_3908 rte_sws_3922 rte_sws_3960 rte_sws_3932 rte_sws_3933 rte_sws_3934 rte_sws_3961 rte_sws_3962 rte_sws_3963 rte_sws_3964 rte_sws_3965
[RTE00157] Support for calibration parameters in NVRAM	rte_sws_3936
[RTE00158] Support separation of calibration parameters	rte_sws_5145 rte_sws_3959 rte_sws_7096 rte_sws_3907 rte_sws_3911 rte_sws_3912 rte_sws_5194 rte_sws_3908
[RTE00159] Sharing of calibration parameters	rte_sws_2750 rte_sws_3958 rte_sws_7186 rte_sws_5112 rte_sws_2749
[RTE00160] Debounced start of Runnable Entities	rte_sws_2697
[RTE00161] Activation Offset of Runnable Entities	rte_sws_7000
[RTE00162] 1:n External Trigger communication	rte_sws_7229 rte_sws_7212 rte_sws_7213 rte_sws_7214 rte_sws_7543 rte_sws_7215 rte_sws_7216 rte_sws_7218 rte_sws_7200 rte_sws_7201 rte_sws_7207
[RTE00163] Support for InterRunnableTriggering	rte_sws_7229 rte_sws_7220 rte_sws_7555 rte_sws_7221 rte_sws_7224 rte_sws_7223 rte_sws_7203 rte_sws_7204 rte_sws_7226 rte_sws_7227 rte_sws_7228 rte_sws_7208

[RTE00164] Ensure a unique naming of generated types visible in the global namespace	rte_sws_7110 rte_sws_7111 rte_sws_6706 rte_sws_6707 rte_sws_6708 rte_sws_7114 rte_sws_7144 rte_sws_7115 rte_sws_7116 rte_sws_7117 rte_sws_7118 rte_sws_7119 rte_sws_7145 rte_sws_7146
[RTE00165] Suppress identical 'C' type redefinitions	rte_sws_7143 rte_sws_7134 rte_sws_7105 rte_sws_7112 rte_sws_7113 rte_sws_7107 rte_sws_7167 rte_sws_7169
[RTE00166] Use the AUTOSAR Standard Types in the global namespace if the AUTOSAR data type is mapped to an AUTOSAR Standard Type	rte_sws_7104 rte_sws_7109 rte_sws_7148 rte_sws_7149 rte_sws_7166 rte_sws_7036 rte_sws_7037 rte_sws_7162 rte_sws_7163
[RTE00167] Encapsulate a Software Component local name space	rte_sws_2575 rte_sws_3809 rte_sws_3810 rte_sws_8401 rte_sws_5051 rte_sws_8402 rte_sws_5052 rte_sws_1004 rte_sws_1276 rte_sws_7122 rte_sws_7123 rte_sws_7132 rte_sws_6513 rte_sws_3854 rte_sws_6515 rte_sws_6518 rte_sws_6519 rte_sws_6520 rte_sws_6530 rte_sws_6541 rte_sws_6542 rte_sws_7140 rte_sws_6716 rte_sws_6717 rte_sws_6718
[RTE00168] Typing of RTE API	rte_sws_7104
[RTE00169] Map code and memory allocated by the RTE to memory sections	rte_sws_5088 rte_sws_7589 rte_sws_7047 rte_sws_7048 rte_sws_7049 rte_sws_7050 rte_sws_7051 rte_sws_7052 rte_sws_7053 rte_sws_7592 rte_sws_7590 rte_sws_7591 rte_sws_5089
[RTE00170] Provide used memory sections description	rte_sws_6725 rte_sws_5086 rte_sws_5089
[RTE00171] Support for fixed and constant data	rte_sws_3930
[RTE00176] Sharing of NVRAM data	rte_sws_7301
[RTE00177] Support of NvBlockComponentType	rte_sws_7353 rte_sws_7303 rte_sws_7632 rte_sws_7355 rte_sws_7633 rte_sws_7343 rte_sws_7398 rte_sws_7399 rte_sws_7312 rte_sws_7317
[RTE00178] Data consistency of NvBlockComponentType	rte_sws_7310 rte_sws_7311 rte_sws_7319 rte_sws_7602 rte_sws_7613 rte_sws_7315 rte_sws_7316 rte_sws_7350 rte_sws_7601 rte_sws_7614

[RTE00179] Support of Update Flag for Data Reception	rte_sws_7654 rte_sws_7385 rte_sws_7386 rte_sws_7387 rte_sws_7689 rte_sws_7390 rte_sws_7391 rte_sws_7392 rte_sws_7393
[RTE00180] DataSemantics range check during runtime	rte_sws_8024 rte_sws_3861 rte_sws_8026 rte_sws_8039 rte_sws_3839 rte_sws_3840 rte_sws_3841 rte_sws_3842 rte_sws_3843 rte_sws_8027 rte_sws_8040 rte_sws_8030 rte_sws_8031 rte_sws_8032 rte_sws_8033 rte_sws_8028 rte_sws_8041 rte_sws_3845 rte_sws_3846 rte_sws_3847 rte_sws_3848 rte_sws_8016 rte_sws_3849 rte_sws_8025 rte_sws_8029 rte_sws_8042 rte_sws_8034 rte_sws_8035 rte_sws_8036 rte_sws_8037 rte_sws_8038 rte_sws_7038
[RTE00181] Conversion between internal and network data types	rte_sws_3827 rte_sws_3828
[RTE00182] Self Scaling Signals at Port Interfaces	rte_sws_3815 rte_sws_3816 rte_sws_7091 rte_sws_7092 rte_sws_7099 rte_sws_3817 rte_sws_3818 rte_sws_3819 rte_sws_3820 rte_sws_3821 rte_sws_3822 rte_sws_3823 rte_sws_3829 rte_sws_3830 rte_sws_3855 rte_sws_3856 rte_sws_3857 rte_sws_3860 rte_sws_3831 rte_sws_3832 rte_sws_3833 rte_sws_7038
[RTE00183] RTE Read API returning the dataElement's value	rte_sws_7396 rte_sws_7394 rte_sws_7395
[RTE00184] RTE Status 'Never Received'	rte_sws_8008 rte_sws_8009 rte_sws_7381 rte_sws_7382 rte_sws_7383 rte_sws_7645 rte_sws_7384 rte_sws_7643 rte_sws_7644
[RTE00185] RTE API with Rte_IFeedback	rte_sws_7378 rte_sws_7652 rte_sws_7367 rte_sws_7646 rte_sws_7647 rte_sws_7648 rte_sws_7649 rte_sws_7374 rte_sws_7375 rte_sws_7650 rte_sws_7376 rte_sws_7660 rte_sws_7651 rte_sws_2608 rte_sws_2666 rte_sws_2589 rte_sws_2590 rte_sws_3836 rte_sws_7379
[RTE00189] A2L Generation Support	rte_sws_5118 rte_sws_5130 rte_sws_5131 rte_sws_5132 rte_sws_5133 rte_sws_5119 rte_sws_5129 rte_sws_5135 rte_sws_5134 rte_sws_5120 rte_sws_5121 rte_sws_6702 rte_sws_5122 rte_sws_5123 rte_sws_5124 rte_sws_5125 rte_sws_5126 rte_sws_5127 rte_sws_5128 rte_sws_7097 rte_sws_5136 rte_sws_5137 rte_sws_5138 rte_sws_5139 rte_sws_5140 rte_sws_5141 rte_sws_5142 rte_sws_5143 rte_sws_5144 rte_sws_5152 rte_sws_5153 rte_sws_5154 rte_sws_5155 rte_sws_5156 rte_sws_5157 rte_sws_5158 rte_sws_5159 rte_sws_5160 rte_sws_5161 rte_sws_5162 rte_sws_6726 rte_sws_5087

[RTE00190] Support for variable-length Data Types	rte_sws_7813 rte_sws_7814
[RTE00191] Support for Variant Handling	rte_sws_5168 rte_sws_5176 rte_sws_5174 rte_sws_5169 rte_sws_5175 rte_sws_6543 rte_sws_6500 rte_sws_6546 rte_sws_6501 rte_sws_6507 rte_sws_6547 rte_sws_6509 rte_sws_6510 rte_sws_6512 rte_sws_6549 rte_sws_6550 rte_sws_6611 rte_sws_6612
[RTE00192] Support multiple trace clients	rte_sws_6725 rte_sws_5086 rte_sws_5093 rte_sws_5091 rte_sws_5092 rte_sws_5106
[RTE00193] Support for Runnable Entity execution chaining	rte_sws_7802 rte_sws_7800
[RTE00195] No activation of Runnable Entities in terminated or restarting partitions	rte_sws_7606 rte_sws_7604
[RTE00196] Inter-partitions communication consistency	rte_sws_2761 rte_sws_7610 rte_sws_5147
[RTE00200] Support of unconnected R-Ports	rte_sws_7655 rte_sws_1330 rte_sws_7663 rte_sws_1331 rte_sws_3785 rte_sws_1333 rte_sws_1334 rte_sws_7690 rte_sws_7665 rte_sws_7656 rte_sws_7657 rte_sws_7691
[RTE00201] Contract Phase with Variant Handling support	rte_sws_5104 rte_sws_6543 rte_sws_6500 rte_sws_6546 rte_sws_6502 rte_sws_6505 rte_sws_6516 rte_sws_6529 rte_sws_6527 rte_sws_6528 rte_sws_6521 rte_sws_6522 rte_sws_6523 rte_sws_6524 rte_sws_6525 rte_sws_6526 rte_sws_6514 rte_sws_6515 rte_sws_6518 rte_sws_6519 rte_sws_6520 rte_sws_6530 rte_sws_6541 rte_sws_6542 rte_sws_6620 rte_sws_6638 rte_sws_6539 rte_sws_6540 rte_sws_6531
[RTE00202] Support for array size variants	rte_sws_6543 rte_sws_6500 rte_sws_6546 rte_sws_6505
[RTE00203] API to read system constant	rte_sws_6517 rte_sws_6514 rte_sws_6513 rte_sws_3854
[RTE00204] Support the selection / deselection of SWC prototypes	rte_sws_5104 rte_sws_6601 rte_sws_6544 rte_sws_6545
[RTE00206] Support the selection of a signal provider	rte_sws_5104 rte_sws_6601 rte_sws_6602 rte_sws_6603 rte_sws_6604 rte_sws_6605 rte_sws_6606 rte_sws_6544 rte_sws_6545

[RTE00207] Support N to M communication patterns while unresolved variations are affecting these communications	rte_sws_5104 rte_sws_6601 rte_sws_6602 rte_sws_6603 rte_sws_6604 rte_sws_6605 rte_sws_6606 rte_sws_6544 rte_sws_6545
[RTE00210] Support for inter OS application communication	rte_sws_7606 rte_sws_2752 rte_sws_2753 rte_sws_8400 rte_sws_2756 rte_sws_2754 rte_sws_2728 rte_sws_3853 rte_sws_2755 rte_sws_2731 rte_sws_2732 rte_sws_8503 rte_sws_8504 rte_sws_8506
[RTE00211] Cyclic time based scheduling of BSW Schedulable Entities	rte_sws_7514 rte_sws_7574 rte_sws_7584 rte_sws_2697 rte_sws_7282 rte_sws_7283
[RTE00212] Activation Offset of BSW Schedulable Entities	rte_sws_7520
[RTE00213] Mode Switches for BSW modules shall be supported	rte_sws_2500 rte_sws_2662 rte_sws_2663 rte_sws_2664 rte_sws_7157 rte_sws_7514 rte_sws_7530 rte_sws_7531 rte_sws_7150 rte_sws_7151 rte_sws_7173 rte_sws_2667 rte_sws_2661 rte_sws_7152 rte_sws_2562 rte_sws_7153 rte_sws_2707 rte_sws_2708 rte_sws_2564 rte_sws_7154 rte_sws_2563 rte_sws_2587 rte_sws_2665 rte_sws_2668 rte_sws_2630 rte_sws_2669 rte_sws_7534 rte_sws_7535 rte_sws_7564 rte_sws_7532 rte_sws_7538 rte_sws_7539 rte_sws_7541 rte_sws_7540 rte_sws_7559 rte_sws_7292 rte_sws_7293 rte_sws_7294 rte_sws_7258 rte_sws_7259 rte_sws_7286 rte_sws_7260 rte_sws_7694 rte_sws_7556 rte_sws_7557 rte_sws_7558 rte_sws_7560 rte_sws_7561 rte_sws_7055 rte_sws_7282 rte_sws_7283
[RTE00214] Common Mode handling for Basic SW and Application SW	rte_sws_2697 rte_sws_7535 rte_sws_7582 rte_sws_7583 rte_sws_7564 rte_sws_7258 rte_sws_7259 rte_sws_7286
[RTE00215] API for Mode switch notification to the SchM	rte_sws_7255 rte_sws_7256 rte_sws_7261 rte_sws_8507
[RTE00216] Triggering of BSW Schedulable Entities by occurrence of External Trigger	rte_sws_7514 rte_sws_7542 rte_sws_7213 rte_sws_7214 rte_sws_7544 rte_sws_7545 rte_sws_7548 rte_sws_7546 rte_sws_7216 rte_sws_7218 rte_sws_7549 rte_sws_7282 rte_sws_7283

[RTE00217] Synchronized activation of Runnable Entities and BSW Schedulable Entities	rte_sws_7218 rte_sws_7549 rte_sws_2697
[RTE00218] API for Triggering BSW modules by Triggered Events	rte_sws_7263 rte_sws_7264 rte_sws_7266 rte_sws_7267
[RTE00219] Support for interlaced execution sequences of Runnable Entities and BSW Schedulable Entities	rte_sws_7517 rte_sws_7518 rte_sws_2697
[RTE00220] ECU life cycle dependent scheduling	rte_sws_7580 rte_sws_2538
[RTE00221] Support for 'BSW integration' builds	rte_sws_7569 rte_sws_7585
[RTE00222] Support shared exclusive areas in BSW Service Modules and the corresponding Service Component	rte_sws_7522 rte_sws_7523 rte_sws_7524 rte_sws_7250 rte_sws_7251 rte_sws_7252 rte_sws_7578 rte_sws_7579 rte_sws_7253 rte_sws_7254
[RTE00223] Callout for partition termination notification	rte_sws_7330 rte_sws_7331 rte_sws_7334 rte_sws_7335 rte_sws_7620 rte_sws_7619 rte_sws_7617 rte_sws_7622
[RTE00224] Callout for partition restart request	rte_sws_7645 rte_sws_7643 rte_sws_7644 rte_sws_7188 rte_sws_7336 rte_sws_7338 rte_sws_7339 rte_sws_7340 rte_sws_7341 rte_sws_7342
[RTE00228] Fan-out NvBlock callback function	rte_sws_7623 rte_sws_7624 rte_sws_7625 rte_sws_7671 rte_sws_7626 rte_sws_7627 rte_sws_7628 rte_sws_7629 rte_sws_7630 rte_sws_7672 rte_sws_7631
[RTE00229] Support for Variant Handling of BSW Modules	rte_sws_5104 rte_sws_6543 rte_sws_6500 rte_sws_6546 rte_sws_6503 rte_sws_6504 rte_sws_6507 rte_sws_6548 rte_sws_6508 rte_sws_6537 rte_sws_6535 rte_sws_6536 rte_sws_6532 rte_sws_6544 rte_sws_6545 rte_sws_6533 rte_sws_6534

[RTE00230] Triggering of BSW Schedulable Entities by occurrence of Internal Trigger	rte_sws_7229 rte_sws_7551 rte_sws_7552 rte_sws_7553 rte_sws_7554
[RTE00231] Support native interface between Rte and Com for Strings and uint8 arrays	rte_sws_7408 rte_sws_7817
[RTE00232] Synchronization of runnable entities	rte_sws_7806 rte_sws_7807 rte_sws_7804 rte_sws_7805
[RTE00233] Generation of the Basic Software Module Description	rte_sws_5184 rte_sws_5185 rte_sws_6725 rte_sws_5086 rte_sws_8305 rte_sws_5165 rte_sws_8404 rte_sws_5177 rte_sws_5179 rte_sws_5180 rte_sws_7085 rte_sws_5166 rte_sws_5181 rte_sws_5182 rte_sws_5183 rte_sws_5167 rte_sws_5187 rte_sws_5186 rte_sws_5190 rte_sws_5188 rte_sws_5189 rte_sws_5191 rte_sws_5192
[RTE00234] Support for Record Type subsetting	rte_sws_7091 rte_sws_7092 rte_sws_7099
[RTE00235] Support queued triggers	rte_sws_7087 rte_sws_7088 rte_sws_7089 rte_sws_7090 rte_sws_6720 rte_sws_6721 rte_sws_6722 rte_sws_6723

2 RTE Overview

2.1 The RTE in the Context of AUTOSAR

The Run-Time Environment (RTE) is at the heart of the AUTOSAR ECU architecture. The RTE is the realization (for a particular ECU) of the interfaces of the AUTOSAR Virtual Function Bus (VFB). The RTE provides the infrastructure services that enable communication to occur between AUTOSAR software-components as well as acting as the means by which AUTOSAR software-components access basic software modules including the OS and communication service.

The RTE encompasses both the variable elements of the system infrastructure that arise from the different mappings of components to ECUs as well as standardized RTE services.

In principle the RTE can be logically divided into two sub-parts realizing:

- the communication between software components
- the scheduling of the software components

To fully describe the concept of the RTE, the Basic Software Scheduler has to be considered as well. The Basic Software Scheduler schedules the schedulable entities of the basic software modules. In some documents the schedulable entities are also called main processing functions.

Due to the situation that the same OS Task might be used for the scheduling of software components and basic software modules the scheduling part of the RTE is strongly linked with the Basic Software Scheduler and can not be clearly separated.

The RTE and the Basic Software Scheduler is generated[1] for each ECU to ensure that the RTE and Basic Software Scheduler is optimal for the ECU [RTE00023].

2.2 AUTOSAR Concepts

This section introduces some important AUTOSAR concepts and how they are implemented within the context of the RTE.

2.2.1 AUTOSAR Software-components

In AUTOSAR, “application” software is conceptually located above the AUTOSAR RTE and consists of “AUTOSAR application software-components” that are ECU and loca-

[1] An implementation is free to *configure* rather than *generate* the RTE and Basic Software Scheduler. The remainder of this specification refers to generation for reasons of simplicity only and these references should not be interpreted as ruling out either a wholly configured, or partially generated and partially configured, RTE and Basic Software Scheduler implementation.

tion independent and “AUTOSAR sensor-actuator components” that are dependent on ECU hardware and thus not readily relocatable for reasons of performance/efficiency. This means that, subject to constraints imposed by the system designer, an AUTOSAR software-component can be deployed to any available ECU during system configuration. The RTE is then responsible for ensuring that components can communicate and that the system continues to function as expected wherever the components are deployed. Considering sensor/actuator software components, they may only directly address the local ECU abstraction. Therefore, access to remote ECU abstraction shall be done through an intermediate sensor/actuator software component which broadcasts the information on the remote ECU. Hence, moving the sensor/actuator software components on different ECUs, may then imply to also move connected devices (sensor/actuator) to the same ECU (provided that efficient access is needed).

An AUTOSAR software-component is defined by a *type* definition that defines the component’s interfaces. A component type is instantiated when the component is deployed to an ECU. A component type can be instantiated more than once on the same ECU in which case the component type is said to be “multiple instantiated”. The RTE supports per-instance memory sections that enable each component instance to have private states.

The RTE supports both AUTOSAR software-components where the source is available (“source-code software-components”) [RTE00024] and AUTOSAR software-components where only the object code (“object-code software components”) is available [RTE00140].

Details of AUTOSAR software-components in relation to the RTE are presented in Section 4.1.3.

2.2.2 Basic Software Modules

As well as “AUTOSAR software-components” an AUTOSAR ECU includes basic software modules. Basic software modules can access the ECU abstraction layer as well as other basic software modules directly and are thus neither ECU nor location independent [2].

An “AUTOSAR software-component” *cannot* directly access basic software modules – all communication is via AUTOSAR interfaces and therefore under the control of the RTE. The requirement to not have direct access applies to all *Basic Software Modules* including the operating system [RTE00020] and the communication service.

[2] The functionality provided by a basic software module cannot be relocated in another ECU. However, the source of some basic software modules can be reused on other ECUs.

2.2.3 Communication

The communication interface of an AUTOSAR software-component consists of several ports (which are characterized by port-interfaces). An AUTOSAR software-component can communicate through its interfaces with other AUTOSAR software-components (whether that component is located on the same ECU or on a different ECU) or with basic software modules that have ports and runnables (i.e `ServiceSwComponents`, `EcuAbstractionSwComponents` and `ComplexDeviceDriverSwComponents`) and are located on the same ECU. This communication can *only* occur via the component's ports. A port can be categorized by either a sender-receiver or client-server port-interface. A sender-receiver interface provides a message passing facility whereas a client-server interface provides function invocation.

2.2.3.1 Communication Paradigms

The RTE provides different paradigms for the communication between software-component instances: sender-receiver (signal passing), client-server (function invocation), mode switch, and `NvBlockSwComponentType` interaction.

Each communication paradigm can be applied to intra-partition software-component distribution (which includes both intra-task and inter-task distribution, within the same Partition), inter-Partition software-component distribution, and inter-ECU software-component distribution. Intra-task communication occurs between runnable entities that are mapped to the same OS task whereas inter-task communication occurs between runnable entities mapped to different tasks of the same Partition and can therefore involve a context switch. Inter-Partition communication occurs between runnable entities in components mapped to different partitions of the same ECU and therefore involve a context switch and crossing a protection boundary (memory protection, timing protection, isolation on a core). Inter-ECU communication occurs between runnable entities in components that have been mapped to different ECUs and so is inherently concurrent and involves potentially unreliable communication.

Details of the communication paradigms that are supported by the RTE are contained in Section 4.3.

2.2.3.2 Communication Modes

The RTE supports two modes for sender-receiver communication:

- **Explicit** — A component uses explicit RTE API calls to send and receive data elements [RTE00098].
- **Implicit** — The RTE automatically reads a specified set of data elements before a runnable is invoked and automatically writes (a different) set of data elements after the runnable entity has terminated [RTE00128] [RTE00129]. The term "im-

plicit" is used here since the runnable does not actively initiate the reception or transmission of data.

Implicit and explicit communication is considered in greater detail in Section 4.3.1.5.

2.2.3.3 Static Communication

[rte_sws_6026]⌈ The RTE shall support static communication only. ⌋*(RTE00025)*

Static communication includes only those communication connections where the source(s) and destination(s) of all communication is known at the point the RTE is generated. [RTE00025]. This includes also connections which are subject to variability because the variant handling concept of AUTOSAR does only support the selection of connectors from a superset of possible connectors to define a particular variant.
Dynamic reconfiguration of communication is not supported due to the run-time and code overhead which would therefore limit the range of devices for which the RTE is suitable.

2.2.3.4 Multiplicity

As well as point to point communication (i.e. "1:1") the RTE supports communication connections with multiple providers or requirers:

- When using sender-receiver communication, the RTE supports both "1:n" (single sender with multiple receivers) [RTE00028] and "n:1" (multiple senders and a single receiver) [RTE00131] communication with the restriction that multiple senders are not allowed for `mode switch notifications`, see metamodel restrictions rte_sws_2670.

  The execution of the multiple senders or receivers is not coordinated by the RTE. This means that the actions of different software-components are independent – the RTE does not ensure that different senders transmit data simultaneously and does not ensure that all receivers read data or receive events simultaneously.

- When using client-server communication, the RTE supports "n:1" (multiple clients and a single server) [RTE00029] communication. The RTE does *not* support "1:n" (single client with multiple servers) client-server communication.

Irrespective of whether "1:1", "n:1" or "1:n" communication is used, the RTE is responsible for implementing the communication connections and therefore the AUTOSAR software-component is unaware of the configuration. This permits an AUTOSAR software-component to be redeployed in a different configuration without modification.

2.2.4 Concurrency

AUTOSAR software-components have no direct access to the OS and hence there are no “tasks” in an AUTOSAR application. Instead, concurrent activity within AUTOSAR is based around *runnable entities* within components that are invoked by the RTE.

The AUTOSAR VFB specification [1] defines a runnable entity as a “sequence of instructions that can be started by the Run-Time Environment”. A component provides one[3] or more runnable entities [RTE00031] and each runnable entity has exactly one entry point. An entry point defines the *symbol* within the software-component’s code that provides the implementation of a runnable entity.

The RTE is responsible for invoking runnable entities – AUTOSAR software-components are not able to (dynamically) create private threads of control. Hence, all activity within an AUTOSAR application is initiated by the triggering of runnable entities by the RTE as a result of `RTEEvent`s.

An *RTEEvent* encompasses all possible situations that can trigger execution of a runnable entity by the RTE. The different classes of RTEEvent are defined in Section 5.7.5.

The RTE supports runnable entities in any component that has an AUTOSAR interface - this includes AUTOSAR software-components and basic software modules.[4]

Runnable entities are divided into multiple categories with each category supporting different facilities. The categories supported by the RTE are described in Section 4.2.2.3.

2.3 The RTE Generator

The RTE generator is one of a set of tools[5] that create the realization of the AUTOSAR virtual function bus for an ECU based on information in the *ECU Configuration Description*. The RTE Generator is responsible for creating the AUTOSAR software-component API functions that link AUTOSAR software-components to the OS and manage communication between AUTOSAR software-components and between AUTOSAR software-components and basic software modules.

Additionally the RTE Generator creates both the *Basic Software Scheduler* and the *Basic Software Scheduler* API functions for each particular instance of a *Basic Software Module*.

The RTE generation process for SWCs has two main phases:

[3] The VFB specification does not permit zero runnable entities.

[4] The OS and COM are basic software modules but present a *standardized interface* to the RTE and have no AUTOSAR interface. The OS and COM therefore do not have runnable entities.

[5] The RTE generator works in conjunction with other tools, for example, the OS and COM generators, to fully realize the AUTOSAR VFB.

- **RTE Contract phase** – a limited set of information about a component, principally the AUTOSAR interface definitions, is used to create an application header file for a component type. The application header file defines the “contract” between component and RTE.
- **RTE Generation phase** - all relevant information about components, their deployment to ECUs and communication connections is used to generate the RTE and optionally the Ioc configuration [12]. One RTE is generated for each ECU in the system.

The two-phase development model ensures that the RTE generated application header files are available for use for source-code AUTOSAR software-components as well as object-code AUTOSAR software-components with both types of component having access to all definitions created as part of the RTE generation process.

The RTE generation process, and the necessary inputs in each phase, are considered in more detail in chapter 3.

2.4 Design Decisions

This section details decisions that affect both the general direction that has been taken as well as the actual content of this document.

1. The role of this document is to specify RTE behavior, not RTE implementation. Implementation details should not be considered to be part of the RTE software specification unless they are explicitly marked as RTE requirements.
2. An AUTOSAR system consists of multiple ECUs each of which contains an RTE that may have been generated by different RTE generators. Consequently, the specification of how RTEs from multiple vendors interoperate is considered to be within the scope of this document.
3. The RTE does not have sufficient information to be able to derive a mapping from runnable entity to OS task. The decision was therefore taken to require that the mapping be specified as part of the RTE input.
4. Support for C++ is provided by making the C RTE API available for C++ components rather than specifying a completely separate object-oriented API. This decision was taken for two reasons; firstly the same interface for the C and C++ simplifies the learning curve and secondly a single interface greatly simplifies both the specification and any subsequent implementations.
5. There is no support within the specification for Java.
6. The AUTOSAR meta-model is a highly expressive language for defining systems however for reasons of practicality certain restrictions and constraints have been placed on the use of the meta-model. The restrictions are described in Appendix A.

3 RTE Generation Process

This chapter describes the methodology of the RTE and Basic Software Scheduler generation. For a detailed description of the overall AUTOSAR methodology refer to methodology document [6].

[rte_sws_2514]⌈ The RTE generator shall produce the same RTE API, RTE code, SchM API and SchM code when the input information is the same. ⌋*(RTE00065)*

The RTE Generator gets involved in the AUTOSAR Methodology several times in different roles. Technically the RTE Generator can be implemented as one tool which is invoked with options to switch between the different roles. Or the RTE Generator could be a set of separate tools. In the following section the individual applications of the RTE Generator are described based on the roles that are take, not necessarily the actual tools.

The RTE Generator is used in different roles for the following phases:

- RTE Contract Phase
- Basic Software Scheduler Contract Phase
- PreBuild Data Set Contract Phase
- Basic Software Scheduler Generation Phase
- RTE Generation Phase
- PreBuild Data Set Generation Phase
- PostBuild Data Set Generation Phase

RTE Generator for Software-Components

In Figure 3.1 the overall AUTOSAR Methodology wrt. Application SW-Components and the RTE Generator.

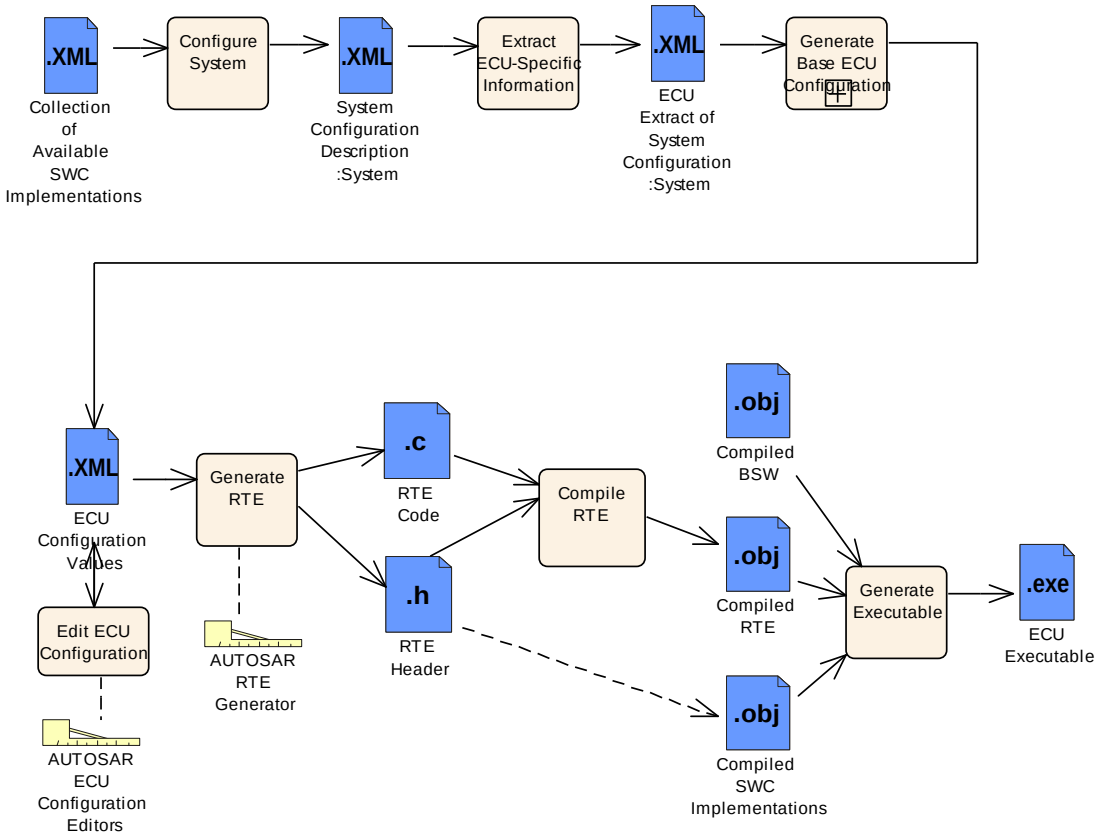


Figure 3.1: System Build Methodology

The whole vehicle functionality is described with means of `CompositionSwComponents`, `SwComponentPrototypes` and `AtomicSwComponents` [2]. In the `CompositionSwComponent` descriptions the connections between the software-components' ports are also defined. Such a collection of software-components connected to each other, without the mapping on actual ECUs, is called the VFB view.

During the 'Configure System' step the needed software-components, the available ECUs and the System Constraints are resolved into a System Configuration Description. Now the `SwComponentPrototypes` and thus the associated `AtomicSwComponents` are mapped on the available ECUs.

Since in the VFB view the communication relationships between the `AtomicSwComponents` have been described and the mapping of each `SwComponentPrototypes` and `AtomicSwComponents` to a specific ECU has been fixed, the communication matrix can be generated. In the `SwComponentType Description` (using the format of the AUTOSAR Software Component Template [2]) the data that is exchanged through ports is defined in an abstract way. Now the 'System Configuration Generator' needs to define system signals (including the actual signal length and the frames in which they will be transmitted) to be able to transmit the application data over the network. COM signals that correspond to the system signals will be later used by the 'RTE Generator' to actually transmit the application data.

In the next step the 'System Configuration Description' is split into descriptions for each individual ECU. During the generation of the Ecu Extract also the hierarchical

structure of the CompositionSwComponents of the VFB view is flattened and the SwComponentPrototypes of the ECU Extract represent actual instances. The Ecu Extract only contains information necessary to configure one ECU individually and it is fed into the ECU Configuration for each ECU.

[rte_sws_5000]⌈ The RTE is configured and generated for each ECU instance individually. ⌋*(RTE00021)*

The 'ECU Configuration Editors' (see also Section 3.3) are working iteratively on the 'ECU Configuration Values' until all configuration issues are resolved. There will be the need for several configuration editors, each specialized on a specific part of ECU Configuration. So one editor might be configuring the COM stack (not the communication matrix but the interaction of the individual modules) while another editor is used to configure the RTE.

Since the configuration of a specific Basic-SW module is not entirely independent from other modules there is the need to apply the editors several times to the 'ECU Configuration Values' to ensure all configuration parameters are consistent.

Only when the configuration issues are resolved the 'RTE Generator' will be used to generate the actual RTE code (see also Section 3.4.2) which will then be compiled and linked together with the other Basic-SW modules and the software-components code.

The 'RTE Generator' needs to cope with many sources of information since the necessary information for the RTE Generator is based on the 'ECU Configuration Values' which might be distributed over several files and itself references to multiple other AUTOSAR descriptions.

[rte_sws_5001]⌈ The RTE Generation tools needs to support input according to the Interoperability of AUTOSAR Authoring Tools document [13]. ⌋*(RTE00048)*

This is just a rough sketch of the main steps necessary to build an ECU with AUTOSAR and how the RTE is involved in this methodology. For a more detailed description of the AUTOSAR Methodology please refer to the methodology document [6]. In the next sections the steps with RTE interaction are explained in more detail.

RTE Generator for Basic Software Scheduler

In Figure 3.2 the overall AUTOSAR Methodology wrt. Basis Software Scheduler and the RTE Generator interaction.

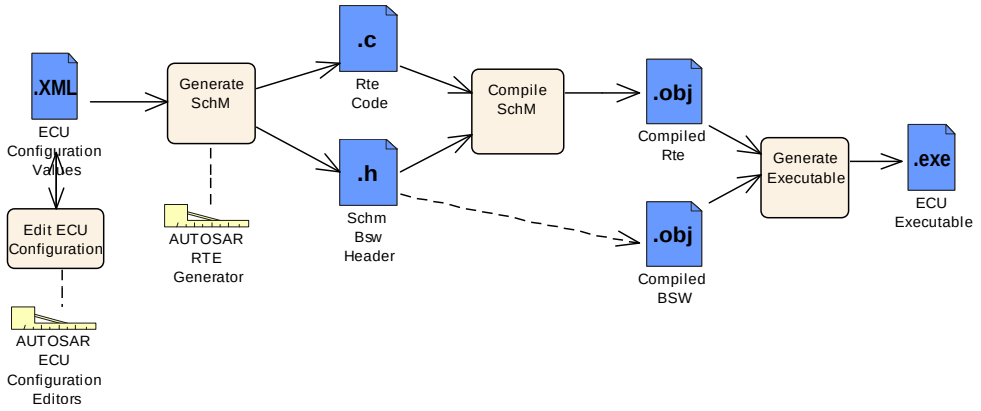


Figure 3.2: Basic Software Scheduler Methodology

The ECU Configuration phase is the start of the Basic Software Scheduler configuration where all the requirements of the different Basic Software Modules are collected. The Input information is provided in the Basic Software Module Descriptions [9] of the individual Basic Software Modules.

The Basic Software Scheduler configuration is then generated into the Basic Software Scheduler code which is compiled and built into the Ecu executable.

3.1 Contract Phase

3.1.1 RTE Contract Phase

To be able to support the AUTOSAR software-component development with RTE-specific APIs the 'Component API' (application header file) is generated from the 'software-component Internal Behavior Description' (see Figure 3.1) by the RTE Generator in the so called 'RTE Contract Phase' (see Figure 3.3).

In the software-component Interface description – which is using the AUTOSAR Software Component Template – at least the AUTOSAR Interfaces of the particular software-component have to be described. This means the software-component Types with Ports and their Interfaces. In the software-component Internal Behavior description additionally the Runnable Entities and the RTE Events are defined. From this information the RTE Generator can generate specific APIs to access the Ports and send and receive data.

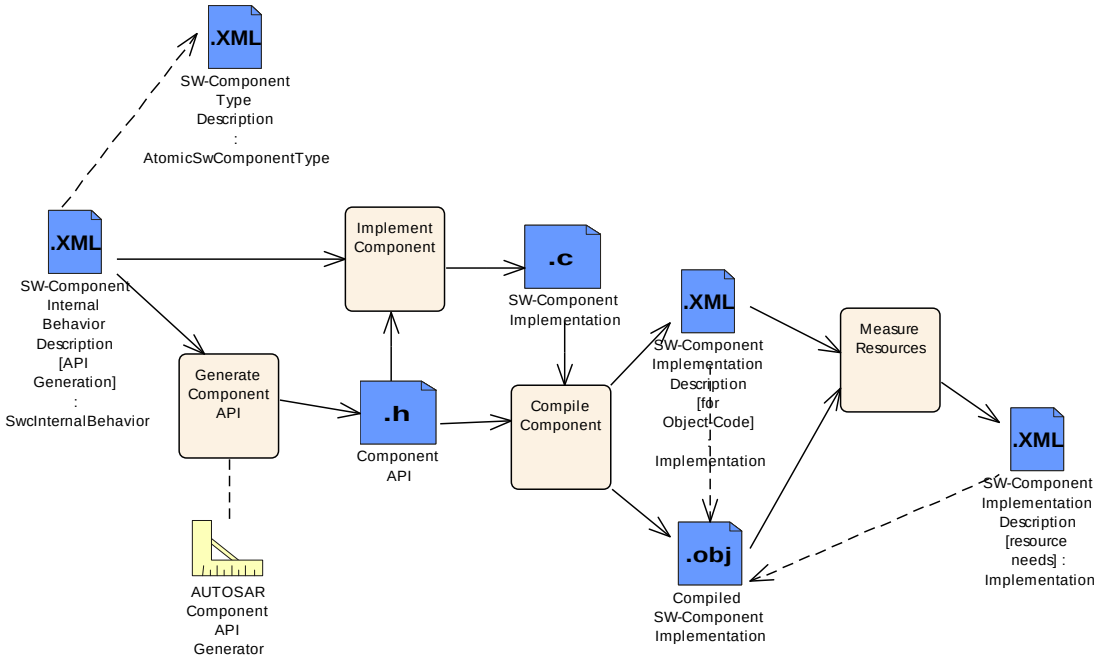


Figure 3.3: RTE Contract Phase

With the generated 'Component API' (application header file) the Software Component developer can provide the Software Component's source code without being concerned as to whether the communication will later be local or using some network(s).

It has to be considered that the AUTOSAR software-component development process is iterative and that the AUTOSAR software-component description might be changed during the development of the AUTOSAR software-component. This requires the application header file to be regenerated to reflect the changes done in the software-component description.

When the software-component has been compiled successfully the 'Component Implementation Description Generation' tool will analyze the resulting object files and enhance the software-component description with the information from the specific implementation. This includes information about the actual memory needs for ROM as well as for RAM and goes into the 'Component Implementation Description' section of the AUTOSAR Software Component Template.

Please note that in case of implemented `PreCompileTime Variability` additionally the *PreBuild Data Set Contract Phase* is required 3.2 to be able to compile the software component.

So when a software-component is delivered it will consist of the following parts:

- SW-Component Type Description
- SW-Component Internal Behavior Description
- The actual SW-Component implementation and/or compiled SW-Component
- SW-Component Implementation Description

The above listed information will be needed to provide enough information for the System Generation steps when the whole system is assembled.

3.1.2 Basic Software Scheduler Contract Phase

To be able to support the *Basic Software Module* development with *Basic Software Scheduler* specific APIs the *Module Interlink Header* (6.3.2) and *Module Interlink Types Header* (6.3.1) containing the definitions and declaration for the *Basic Software Scheduler* API related to the single *Basic Software Module* instance is generated by the RTE Generator in the so called '*Basic Software Scheduler Contract Phase*'.

The required input is

- *Basic Software Module Description* and
- *Basic Software Module Internal Behavior* and
- *Basic Software Module Implementation*

Please note that in case of implemented `PreCompileTime Variability` additionally the *PreBuild Data Set Contract Phase* is required 3.2 to be able to compile the *Basic Software Module*.

3.2 PreBuild Data Set Contract Phase

In the *RTE PreBuild Data Set Contract Phase* are the *Condition Value Macros* (see 5.3.8.2.2) generated which are required to resolve the implemented `PreBuild Variability` of a particular software component or *Basic Software Module*.

The particular values are defined via `PredefinedVariant`s. These `PredefinedVariant` elements containing definition of `SwSystemconstValue`s for `SwSystemconst`s which shall be applied when resolving the variability during ECU Configuration.

The output of this phase is the *RTE Configuration Header File* 5.3.8. This file is required to compile a particular variant of a software component using `PreCompileTime Variability`. The *Condition Value Macros* are used for the implementation of `PreCompileTime Variability` with preprocessor statements and therefore are needed to run the C preprocessor resolving the implemented variability.

3.3 Edit ECU Configuration of the RTE

During the configuration of an ECU the RTE also needs to be configured. This is divided into several steps which have to be performed iteratively: The configuration of the RTE and the configuration of other modules.

So first the 'RTE Configuration Editor' needs to collect all the information needed to establish an operational RTE. This gathering includes information on the software-component instances and their communication relationships, the Runnable Entities and the involved RTE-Events and so on. The main source for all this information is the 'ECU Configuration Values', which might provide references to further descriptions like the software-component description or the System Configuration description.

An additional input source is the Specification of Timing Extensions [14]. This template can be used to specify the execution order of runnable entities (see section 'Execution order constraint'). An 'RTE Configuration Editor' can use the information to create and check the configuration of the Rte Event to Os task mapping (see section 7.6.1).

The usage of 'ECU Configuration Editors' covering different parts of the 'ECU Configuration Values' will – if there are no cyclic dependencies which do not converge – converge to a stable configuration and then the ECU Configuration process is finished. A detailed description of the ECU Configuration can be found in [15]. The next phase is the generation of the actual RTE code.

3.4 Generation Phase

After the ECU has been entirely configured the generation of the actual RTE inclusive the *Basic Software Scheduler* part can be performed. Since all the relationships to and from the other Basic-SW modules have been already resolved during the ECU Configuration phase, the generation can be performed in parallel for all modules (see Figure 3.4).

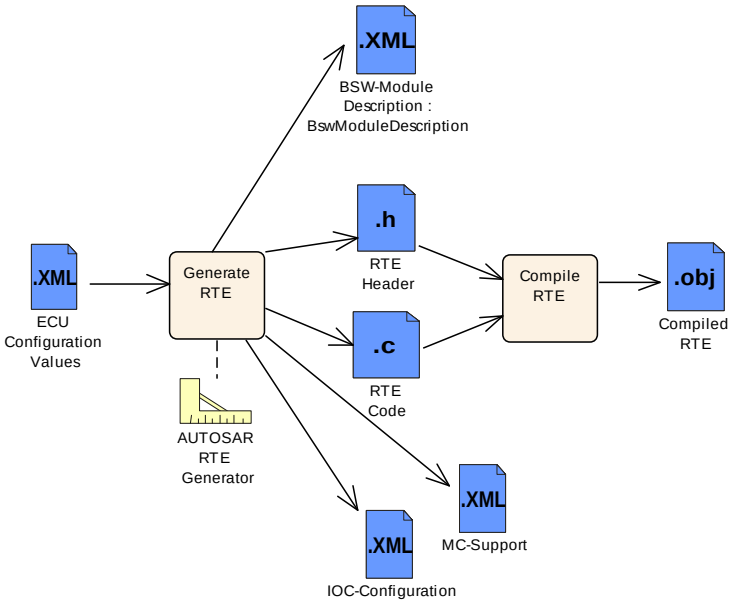


Figure 3.4: RTE Generation Phase

The *Basic Software Scheduler* is a part of the *Rte* and therefore not explicitly shown in figure 3.4.

3.4.1 Basic Software Scheduler Generation Phase

Depending on the complexity of the ECU and the cooperation model of the different software vendors it might be required to integrate the *Basic Software* stand alone without software components.

Therefore the RTE Generator has to support the generation of the *Basic Software Scheduler* without software component related RTE fragments. The *Basic Software Scheduler Generation Phase* is only applicable for software builds which are not containing any kind of software components.

[rte_sws_7569]⌈ In the *Basic Software Scheduler Generation Phase* the RTE Generator shall generate the *Basic Software Scheduler* without the RTE functionality. ⌋*(RTE00221)*

In this case the RTE Generator generates the API for *Basic Software Modules* and the *Basic Software Scheduling* code only. When the input contains software component related information this information raises an error.

For instance:

- *Application Header File*s are not generated for the software components contained in the ECU extract.
- Mapped `RTEEvent`s are not permitted and the runnable calls are not generated into the OS task bodies. Nevertheless all OS task bodies related to the *Basic Software Scheduler* configuration are generated.
- Mode machine instances mapped to the RTE are not supported.

[rte_sws_7585]⌈ In the *Basic Software Scheduler Generation Phase* the RTE Generator shall reject input configuration containing software component related information. ⌋*(RTE00221)*

The RTE Generator in the *Basic Software Scheduler Generation Phase* is also responsible to generate additional artifacts which contribute to the further build, deployment and calibration of the ECU's software.

[rte_sws_6725]⌈ The RTE Generator in *Basic Software Scheduler Generation Phase* shall provide its *Basic Software Module Description* in order to capture the generated RTE's / Basic Software Scheduler attributes. ⌋*(RTE00170, RTE00192, RTE00233)*

Details about the Basic Software Module Description generation can can be found in section 3.4.3.

[rte_sws_6726]⌈ The RTE Generator in *Basic Software Scheduler Generation Phase* shall provide an *MC-Support* (Measurement and Calibration) description as part of the *Basic Software Module Description*. ⌋*(RTE00153, RTE00189)*

Details about the *MC-Support* can be found in section 4.2.8.4.

For software builds which are containing software components the *RTE Generation Phase* 3.4.2 is applicable where the *Basic Software Scheduler* part of the RTE is generated as well.

3.4.2 RTE Generation Phase

The actual AUTOSAR software-components and Basic-SW modules code will be linked together with the RTE and *Basic Software Scheduler* code to build the entire ECU software.

Please note that in case of implemented `PreCompileTime Variability` additionally the *PreBuild Data Set Generation Phase* is required (see section 3.5) to be able to compile the ECU software. Further on in case of implemented `PostBuild Variability` *PostBuild Data Set Generation Phase* is required (see section 3.6) to be able to link the full ECU software.

The RTE Generator in the *Generation Phase* is also responsible to generate additional artifacts which contribute to the further build, deployment and calibration of the ECU's software.

[rte_sws_5086]⌈ The RTE Generator in Generation Phase shall provide its *Basic Software Module Description* in order to capture the generated RTE's attributes. ⌋*(RTE00170, RTE00192, RTE00233)*

Details about the Basic Software Module Description generation can can be found in section 3.4.3.

[rte_sws_5087]⌈ The RTE Generator in Generation Phase shall provide an *MC-Support* (Measurement and Calibration) description as part of the *Basic Software Module Description*. ⌋*(RTE00153, RTE00189)*

Details about the *MC-Support* can be found in section 4.2.8.4.

[rte_sws_5147]⌈ The RTE Generator in Generation Phase shall provide the configuration for the Ioc module [12] if the Ioc module is used. ⌋*(RTE00196)*

The RTE generates the IOC configurations and uses an implementation specific deterministic generation scheme. This generation scheme can be used by implementations to reuse these IOC configurations (e.g. if the configuration switch `strictConfigurationCheck` is used).

[rte_sws_8400]⌈ The RTE Generator in Generation Phase shall generate internal ImplementationDataTypes types used for IOC configuration. ⌋*(RTE00210)*

The corresponding C data types shall be generated into the *Rte_Type.h*. This *Rte_Type.h* header file shall be used by the IOC to get the types for the IOC API.

Changing the RTE generator will require a new IOC configuration generation.

Details about the Ioc module can be found in section 4.3.4.1.

[rte_sws_8305]⌈ The RTE Generator in Generation Phase shall ignore XML-Content categorized as ICS. ⌋*(RTE00233)*

3.4.3 Basic Software Module Description generation

The Basic Software Module Description [9] generated by the RTE Generator in generation phase describes features of the actual RTE code. The following requirements specify which elements of the Basic Software Module Description are mandatory to be generated by the RTE Generator.

3.4.3.1 Bsw Module Description

[rte_sws_5165]⌈ The RTE Generator in Generation Phase shall provide the `BswModuleDescription` element of the Basic Software Module Description for the generated RTE. ⌋*(RTE00233)*

[rte_sws_8404]⌈ The RTE `BswModuleDescription` shall be provided in `ARPackage` AUTOSAR_Rte according to AUTOSAR Generic Structure Template [10] (chapter "Identifying M1 elements in packages"). ⌋*(RTE00233)*

[rte_sws_5177]⌈ The RTE Generator in Generation Phase shall provide the `BswModuleEntry` and a reference to it from the `BswModuleDescription` in the role `providedEntry` for each *Standardized Interface* provided by the RTE (see Layered Software Architecture [16] page *tz76a* and page *94ju5*). The provided *Standardized Interfaces* are the Rte Lifecycle API (section 5.8) and the SchM Lifecycle API (section 6.7). ⌋*(RTE00233)*

[rte_sws_5179]⌈ The RTE Generator in Generation Phase shall provide the `BswModuleDependency` in the `BswModuleDescription` with the role `bswModuleDependency` for each callback API provided by the RTE and called by the respective Basic Software Module. The reference from the `BswModuleDependency` to the `BswModuleEntry` shall be in the role `expectedCallback`. The calling Basic Software Module is specified in the attribute `targetModuleId` of the `BswModuleDependency`. ⌋*(RTE00233)*

For all the APIs the RTE code is invoking in other Basic Software Modules the dependencies are described via requirement rte_sws_5180.

[rte_sws_5180]⌈ The RTE Generator in Generation Phase shall provide the `BswModuleDependency` in the `BswModuleDescription` with the role `bswModuleDependency` for each API called by the RTE in another Basic Software Module. The reference from the `BswModuleDependency` to the `BswModuleEntry` shall be in the role `requiredEntry`. The called Basic Software Module is specified in the attribute `targetModuleId` of the `BswModuleDependency`. ⌋*(RTE00233)*

[rte_sws_7085]⌈ If the Basic Software Module Description for the generated RTE depends from elements in Basic Software Module Descriptions of other Basic Software Modules the RTE Generator shall use the full qualified path name to this elements according the rules in "Identifying M1 elements in packages" of the document AUTOSAR Generic Structure Template [10]. ⌋*(RTE00233)*

For instance the description of the the hook function

```
1 void Rte_Dlt_Task_Activate(TaskType task)
```

for the Dlt needs the `ImplementationDataType` "TaskType" from the OS in order to describe the data type of the `SwServiceArg` "task" in the description of the related `BswModuleEntry`.

In this case the full qualified path name to the `ImplementationDataType` "TaskType" shall be

1 AUTOSAR_OS/ImplementationDataTypes/TaskType

The full example about the description is given below:

```
<AR-PACKAGE>
  <SHORT-NAME>AUTOSAR_RTE</SHORT-NAME>
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>BswModuleEntrys</SHORT-NAME>
      <ELEMENTS>
        <BSW-MODULE-ENTRY>
          <SHORT-NAME>Rte_Dlt_Task_Activate</SHORT-NAME>
          <ARGUMENTS>
            <SW-SERVICE-ARG>
              <SHORT-NAME>task</SHORT-NAME>
              <CATEGORY>TYPE_REFERENCE</CATEGORY>
              <SW-DATA-DEF-PROPS>
                <SW-DATA-DEF-PROPS-VARIANTS>
                  <SW-DATA-DEF-PROPS-CONDITIONAL>
                    <IMPLEMENTATION-DATA-TYPE-REF DEST="IMPLEMENTATION-
                      DATA-TYPE">AUTOSAR_OS/ImplementationDataTypes/
                      TaskType</IMPLEMENTATION-DATA-TYPE-REF>
                  </SW-DATA-DEF-PROPS-CONDITIONAL>
                </SW-DATA-DEF-PROPS-VARIANTS>
              </SW-DATA-DEF-PROPS>
            </SW-SERVICE-ARG>
          </ARGUMENTS>
        </BSW-MODULE-ENTRY>
      </ELEMENTS>
    </AR-PACKAGE>
```

3.4.3.2 Bsw Internal Behavior

[rte_sws_5166]⌈ The RTE Generator in Generation Phase shall provide the BswInternalBehavior element in the BswModuleDescription of the Basic Software Module Description for the generated RTE. ⌋*(RTE00233)*

[rte_sws_5181]⌈ The RTE Generator in Generation Phase shall provide the BswCalledEntity element in the BswInternalBehavior for each C-function implementing the lifecycle APIs (section 5.8) and the SchM Lifecycle API (section 6.7). The BswCalledEntity shall have a reference to the respective BswModuleEntry (rte_sws_5177) in the role implementedEntry. ⌋*(RTE00233)*

[rte_sws_5182]⌈ The RTE Generator in Generation Phase shall provide the VariableDataPrototype element in the BswInternalBehavior in the role staticMemory for each variable memory object the RTE allocates. ⌋*(RTE00233)*

[rte_sws_5183]⌈ The RTE Generator in Generation Phase shall provide the ParameterDataPrototype element in the BswInternalBehavior in the role constantMemory for each constant memory object the RTE allocates. ⌋*(RTE00233)*

3.4.3.3 Bsw Implementation

[rte_sws_5167]⌈ The RTE Generator in Generation Phase shall provide the `BswImplementation` element and a reference to the `BswInternalBehavior` of the Basic Software Module Description in the role `behavior`. ⌋*(RTE00233)*

[rte_sws_5187]⌈ The RTE Generator in Generation Phase shall provide the `programmingLanguage` element in the `BswImplementation` element according to the actual RTE implementation. ⌋*(RTE00233)*

[rte_sws_5186]⌈ The RTE Generator in Generation Phase shall provide the `swVersion` element in the `BswImplementation` element according to the input information from the RTE Ecu configuration (rte_sws_5184, rte_sws_5185). ⌋*(RTE00233)*

[rte_sws_5190]⌈ The RTE Generator in Generation Phase shall provide the `arReleaseVersion` element in the `BswImplementation` element according to AUTOSAR release version the RTE Generator is based on. ⌋*(RTE00233)*

[rte_sws_5188]⌈ The RTE Generator in Generation Phase shall provide the `usedCodeGenerator` element in the `BswImplementation` element according to the actual RTE implementation. ⌋*(RTE00233)*

[rte_sws_5189]⌈ The RTE Generator in Generation Phase shall provide the `vendorId` element in the `BswImplementation` element according to the input information from the RTE Ecu configuration (`RteCodeVedorId`). ⌋*(RTE00233)*

The `RteCodeVedorId` specifies the vendor id of the actual user of the RTE Generator, not the id of the RTE Vendor itself.

[rte_sws_5191]⌈ If the generated RTE code is hardware specific (due to vendor specific optimizations of the RTE Generator) then the reference to the applicable `HwElements` from the ECU Resource Description [17] shall be provided in the `BswImplementation` element with the role `hwElement`. ⌋*(RTE00233)*

[rte_sws_5192]⌈ The RTE Generator in Generation Phase shall provide the `DependencyOnArtifact` element in the `BswImplementation` with the role `generatedArtifact` for all c- and header-files which are required to compile the Rte code. This does not include other Basic Software modules or Application Software. ⌋*(RTE00233)*

Note: The use case is the support of the build-environment (automatic or manual).

Attributes shall be used in this context as follow:

- `category` shall be used as defined in Generic Structure Template [10] (e.g. SWSRC, SWOBJ, SWHDR)
- `domain` is optional and can be chosen freely
- `revisionLabel` shall contain the revision label out of RTE Configuration
- `shortLabel` is the name of artifact

Details on the description of DependencyOnArtifact can be found in the Generic Structure Template [10].

Additional elements of the *Basic Software Module Description* shall be exported are specified in later requirements e.g. in section 4.2.8.4 and section 5.1.2.4.

3.5 PreBuild Data Set Generation Phase

During the *PreBuild Data Set Generation Phase* are the *Condition Value Macros* (see 5.3.8.2.2) generated which are required to resolve the implemented PreBuild Variability of the software components, generated RTE and *Basic Software Scheduler*.

The particular values are defined via the EcucVariationResolver configuration selecting PredefinedVariants. These PredefinedVariant elements containing definition of SwSystemconstValues for SwSystemconsts which shall be applied when resolving the variability during ECU Configuration.

The values of the *Condition Value Macros* are the results of evaluated ConditionByFormulas of the related VariationPoints. These ConditionByFormulas referencing SwSystemconsts in the formula expressions. It is supported that the assigned SwSystemconstValue might contain again a formula expressions referencing SwSystemconsts. Therefore the input might be a tree of formula expressions and SwSystemconstValues but the leaf SwSystemconstValues are required to be values which are not dependent from other SwSystemconsts to ensure that the evaluation of the tree results in a unique number.

[rte_sws_6610]⌈ The RTE generator shall validate the resolved pre-build variants and check the integrity with regards to the meta model. Any meta model violation shall result in the rejection of the input configuration. ⌋*(RTE00018)*

The output of this phase is the *RTE Configuration Header File* 5.3.8.This file is required to compile a particular variant of ECU software including software component code and RTE code using PreCompileTime Variability. The *Condition Value Macros* are used for the implementation of PreCompileTime Variability with preprocessor statements and therefore are needed to run the C preprocessor resolving the implemented variability.

3.6 PostBuild Data Set Generation Phase

In the *PostBuild Data Set Generation Phase* the PredefinedVariant values are generated which are required to resolve the implemented PostBuild Variability of the software components and generated RTE.

The output of this phase are the *RTE Post Build Variant Sets* 5.3.10. This file is required to link the ECU software and to select a particular PostBuild variant in the generated RTE code during start up when the *Basic Software Scheduler* is initialized.

[rte_sws_6611]⌈ If the DET is enabled then the RTE shall generate validation code which at runtime (i.e. during initialization) validates the resolved post-build variants and check the integrity with regards to the active variants. If a violation is detected the RTE shall report a development error to the DET. To execute this validation RTE initialization will get a pointer to the `RtePostBuildVariantConfiguration` instance to allow it to validate the selected variant. ⌋*(RTE00191)*

[rte_sws_6612]⌈ The RTE generator shall create an RTE Post Build Data Set configuration (i.e. Rte_PBCfg.c) representing the collection of `PredefinedVariant` definitions (typically for each subsystem and/or system configuration) providing and defining the post build variants of the RTE. ⌋*(RTE00191)*

Note that the Rte_PBCfg.h is generated during the Rte Generation phase. An Rte_PBCfg.c may also have to be generated at that time to reserve memory (with default values).

Additional details about these configuration files are described in section 5.3.10.

An RTE variant can consist of a collection of `PredefinedVariant`s. Each `PredefinedVariant` contains a collection of `PostBuildVariantCriterionValue`s which assigns a value to a specific `PostBuildVariantCriterion` which in turn is used to resolve the variability at runtime by evaluating a `PostBuildVariantCondition`. Different `PredefinedVariant`s could assign different values to the same `PostBuildVariantCriterion` and as such create conflicts for a specific `PostBuildVariantSet`. It is allowed to have different assignments if these assignment assign the same value.

[rte_sws_6613]⌈ The RTE Generator shall reject configurations where different `PredefinedVariant`s assign different values to the same `PostBuildVariantCriterion` for the same `RtePostBuildVariantConfiguration`. ⌋*(RTE00018)*

3.7 RTE Configuration interaction with other BSW Modules

The generated RTE interacts heavily with other AUTOSAR Basic Software Modules like Com and Os. The configuration values for the different BSW Modules are stored in individual structures of ECU Configuration it is however essential that the common used values are synchronized between the different BSW Module's configurations. AUTOSAR does not provide a standardized way how the individual configurations can be synchronized, it is assumed that during the generation of the BSW Modules the input information provided to the individual BSW Module is in sync with the input information provided to other (dependent) BSW Modules.

The AUTOSAR BSW Module code-generation methodology is heavily relying on the logical distinction between Configuration editors and configuration generators. These

tools do not necessarily have to be implemented as two separate tools, it just shall be possible to distinguish the different roles the tools take during a certain step in the methodology.

For the RTE it is assumed that tool support for the resolution of interactions between the Rte and other BSW Modules is needed to allow an efficient configuration of the Rte. It is however not specified how and in which tools this support shall be implemented.

The RTE Generator in Generation Phase needs information about other BSW Module's configurations based on the configuration input of the Rte itself (there are references in the configuration of the Rte which point to configuration values of other BSW Modules). If during RTE Generation Phase the provided input information is inconsistent wrt. the Rte input the Rte Generator will have to consider the input as invalid configuration.

[rte_sws_5149]⌈ The RTE Generator in Generation Phase shall consider errors in the Rte configuration input information as invalid configuration. ⌋*(RTE00018)*

Due to implementation freedom of the RTE Generator it should be possible to correct / update provided input configurations of other BSW Modules based on the RTE configuration requirements. But to allow a stable build process it shall be possible to disallow such an update behavior.

[rte_sws_5150]⌈ If the external configuration switch `strictConfigurationCheck` is set to *true* the Rte Generator shall not create or modify any configuration input. ⌋*(RTE00065)*

If the external configuration switch `strictConfigurationCheck` (see rte_sws_5148) is set to *false* the Rte Generator may update the input configuration information of the Rte and other BSW Modules.

Example: If the Rte configuration is referencing an `OsTask` which is not configured in the provided Os configuration, the RTE Generator would behave like:

- In case rte_sws_5150 applies: Only show an error message.
- Otherwise: Possible behavior: Show a warning message and modify the Os configuration to contain the `OsTask` which is referred to by the Rte configuration (Of course the Os configuration of this new `OsTask` needs to be refined afterwards).

4 RTE Functional Specification

4.1 Architectural concepts

4.1.1 Scope

In this section the concept of an AUTOSAR software-component and its usage within the RTE is introduced.

The AUTOSAR Software Component Template [2] defines the kinds of software-components within the AUTOSAR context. These are shown in Figure 4.1. The abstract `SwComponentType` can not be instantiated, so there can only be either a `CompositionSwComponentType`, a `ParameterSwComponentType`, or a specialized class `ApplicationSwComponentType`, `ServiceProxySwComponentType`, `SensorActuatorSwComponentType`, `NvBlockSwComponentType`, `ServiceSwComponentType`, `ComplexDeviceDriverSwComponentType`, or `EcuAbstractionSwComponentType` of the abstract class `AtomicSwComponentType`.

In the following document the term `AtomicSwComponentType` is used as collective term for all the mentioned non-abstract derived meta-classes.

The `SwComponentType` is defining the type of an AUTOSAR software-component which is independent of any usage and can be potentially re-used several times in different scenarios. In a composition the types are occurring in specific roles which are called `SwComponentPrototype`s. The prototype is the utilization of a type within a certain scenario. In AUTOSAR any `SwComponentType` can be used as a type for a prototype.

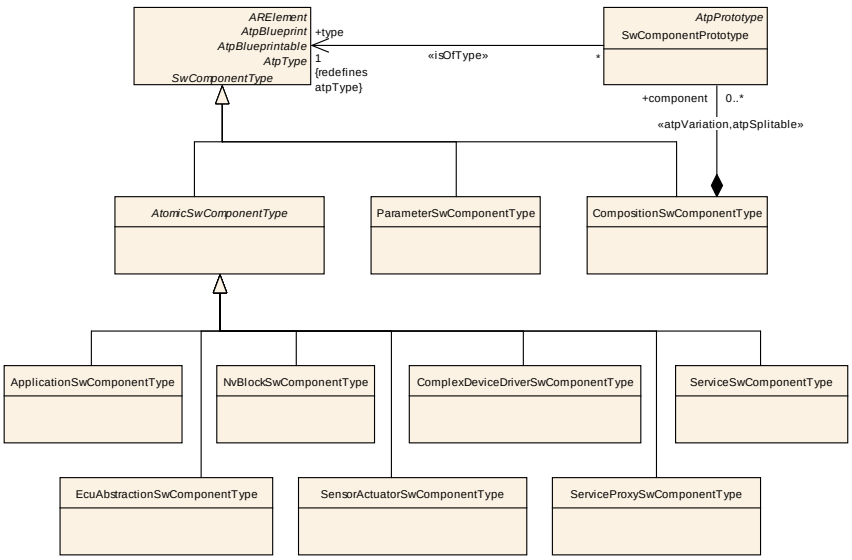


Figure 4.1: AUTOSAR software-component classification

The AUTOSAR software-components shown in Figure 4.1 are located above and below the RTE in the architectural Figure 4.2.

Below the RTE there are also software entities that have an AUTOSAR Interface. These are the AUTOSAR services, the ECU Abstraction and the Complex Device Drivers. For these software not only the AUTOSAR Interface will be described but also information about their internal structure will be available in the Basic Software Module Description.

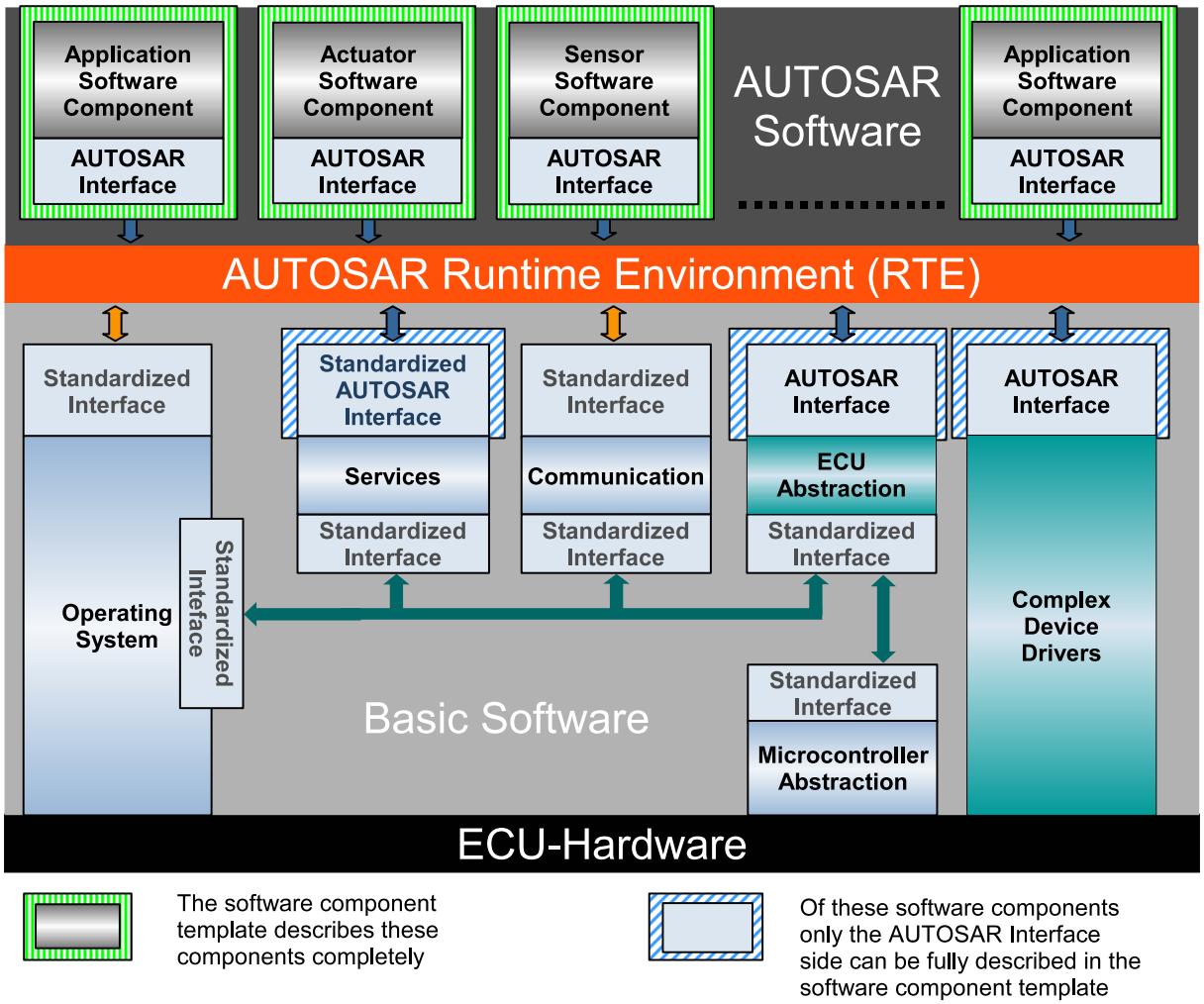


Figure 4.2: AUTOSAR ECU architecture diagram

In the next sections the different AUTOSAR software-components kinds will be described in detail with respect to their influence on the RTE.

4.1.2 RTE and Data Types

The AUTOSAR Meta Model defines `ApplicationDataTypes` and `ImplementationDataTypes`. A `AutosarDataPrototype` can be typed by an `ApplicationDataType` or an `ImplementationDataType`. But the RTE Generator only implements `ImplementationDataTypes` as C data types and uses these C data types to type the RTE API which is related to `DataPrototypes`. Therefore it is required in the input configuration that every `ApplicationDataType` used for the typing of a

DataPrototype which is relevant for RTE generation is mapped to an ImplementationDataType with a DataTypeMapping. Which DataTypeMapping is applicable for an particular software component respectively Basic Software Module is defined by the DataTypeMappingSets referenced by the InternalBehavior.

[rte_sws_7028]⌈ The RTE Generator shall reject input configurations containing a AutosarDataPrototype which influences the generated RTE and which is typed by an ApplicationDataType not mapped to an ImplementationDataType. ⌋*(RTE00018)*

Nevertheless a subset of the attributes given by the ApplicationDataTypes are relevant for the RTE generator for instance

- to create the *McSupportData* (see section 4.2.8.4) information
- to calculate the conversion formula in case of *Data Conversion* (see section 4.3.5 and 4.3.5.3)
- to calculate numerical representation of values required for the RTE code but defined in the physical representation (e.g. initialValues and invalidValues).

[rte_sws_7038]⌈ The RTE Generator shall calculate the numerical representation of values provided in the physical representation and required for the RTE code according the conversion defined by an compuMethod for instances of VariableDataPrototypes, ArgumentDataPrototypes and ParameterDataPrototypes typed by an ApplicationDataType of category VALUE, VAL_BLK, STRUCTURE, ARRAY, BOOLEAN. If there is no CompuMethod provided the conversion is treated like an CompuMethod of category IDENTICAL. ⌋*(RTE00180, RTE00182)*

4.1.3 RTE and AUTOSAR Software-Components

The description of an AUTOSAR software-component is divided into the sections

- hierarchical structure
- ports and interfaces
- internal behavior
- implementation

which will be addressed separately in the following sections.

[rte_sws_7196]⌈ The RTE Generator shall respect the precedence of data properties defined via SwDataDefProps as defined in the *Software Component Template* [2]. ⌋*()*

Requirement rte_sws_7196 means that:

1. SwDataDefProps defined on ApplicationDataType which may be overwritten by
2. SwDataDefProps defined on ImplementationDataType which may be overwritten by
3. SwDataDefProps defined on AutosarDataPrototype which may be overwritten by
4. SwDataDefProps defined on InstantiationDataDefProps which may be overwritten by
5. SwDataDefProps defined on AccessPoint respectively Argument which may be overwritten by
6. SwDataDefProps defined on FlatInstanceDescriptor which may be overwritten by
7. SwDataDefProps defined on McDataInstance

The SwDataDefProps defined on McDataInstance are not relevant for the RTE generation but rather the documentation of the generated RTE.

Especially the attributes swAddrMethod, swCalibrationAccess, swImplPolicy and DataConstr do have an impact on the generated RTE. In the following document only the attribute names are mentioned with the semantic that this refers to the most significant one.

4.1.3.1 Hierarchical Structure of Software-Components

In AUTOSAR the structure of an E/E-system is described using the AUTOSAR Software Component Template and especially the mechanism of compositions. Such a Top Level Composition assembles subsystems and connects their ports.

Of course such a composition utilizes a lot of hierarchical levels where compositions instantiate other composition types and so on. But at some low hierarchical level each composition only consists of AtomicSwComponentType instances. And those instances of AtomicSwComponentTypes are what the RTE is going to be working with.

4.1.3.2 Ports, Interfaces and Connections

Each AUTOSAR software-component (SwComponentType) can have ports (PortPrototype). An AUTOSAR software-component has provide ports (PPortPrototype) and/or has require ports (RPortPrototype) to communicate with other AUTOSAR software-components. The requiredInterface or providedInterface (PortInterface) determines if the port is a sender/receiver or a client/server port. The attribute isService is used with AUTOSAR Services (see section 4.1.5).

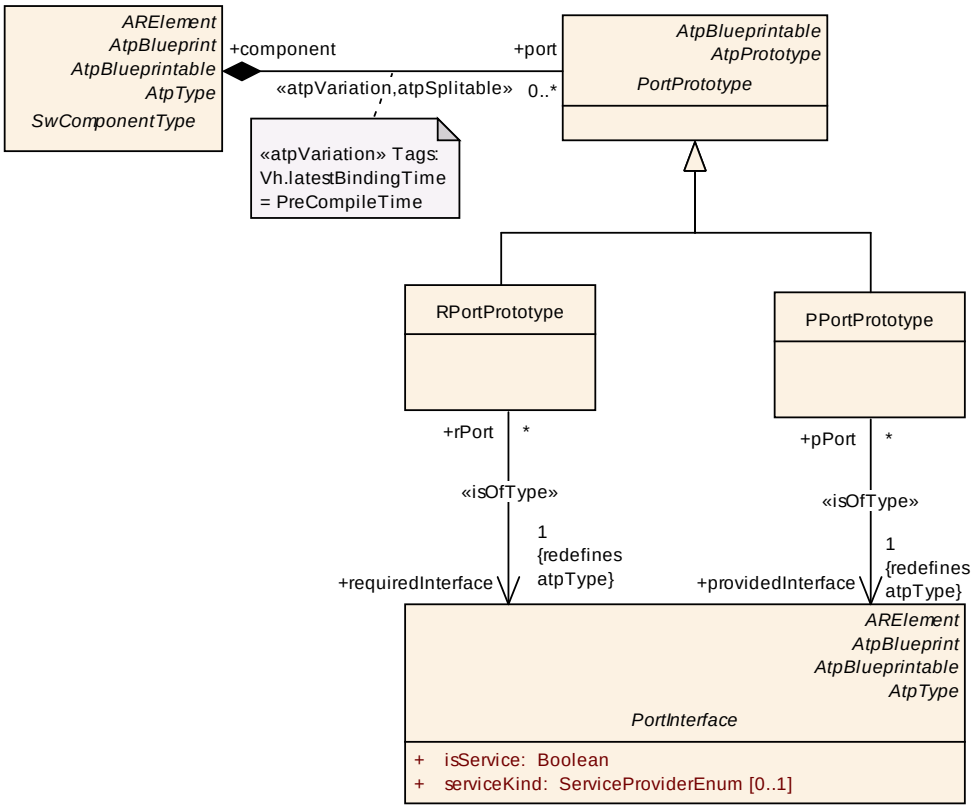


Figure 4.3: Software-Components and Ports

When compositions are built of instances the ports can be connected either within the composition or made accessible to the outside of the composition. For the connections inside a composition the `AssemblySwConnector` is used, while the `Delegation-SwConnector` is used to connect ports from the inside of a composition to the outside. Ports not connected will be handled according to the requirement [RTE00139].

The next step is to map the SW-C instances on ECUs and to establish the communication relationships. From this step the actual communication is derived, so it is now fixed if a connection between two instance's ports is going to be over a communication bus or locally within one ECU.

[rte_sws_2200]⌈ The RTE shall implement the communication paths specified by the ECU Configuration description. ⌋*(RTE00027)*

[rte_sws_2201]⌈ The RTE shall implement the semantic of the communication attributes given by the AUTOSAR software-component description. The semantic of the given communication mechanism shall not change regardless of whether the communication partner is located on the same partition, on another partition of the same ECU or on a remote ECU, or whether the communication is done by the RTE itself or by the RTE calling COM or IOC. ⌋*(RTE00027)*

E.g., according to rte_sws_2200 and rte_sws_2201 the RTE is not permitted to change the semantic of an asynchronous client to synchronous because both client and server are mapped to the very same ECU.

4.1.3.3 Internal Behavior

Only for AtomicSwComponentTypes the internal structure is exposed in the SwcInternalBehavior description. Here the definition of the RunnableEntitys and used RTEEvents is done (see Figure 4.4).

The AUTOSAR MetaModel enforces that there is at most one SwcInternalBehavior per AtomicSwComponentType

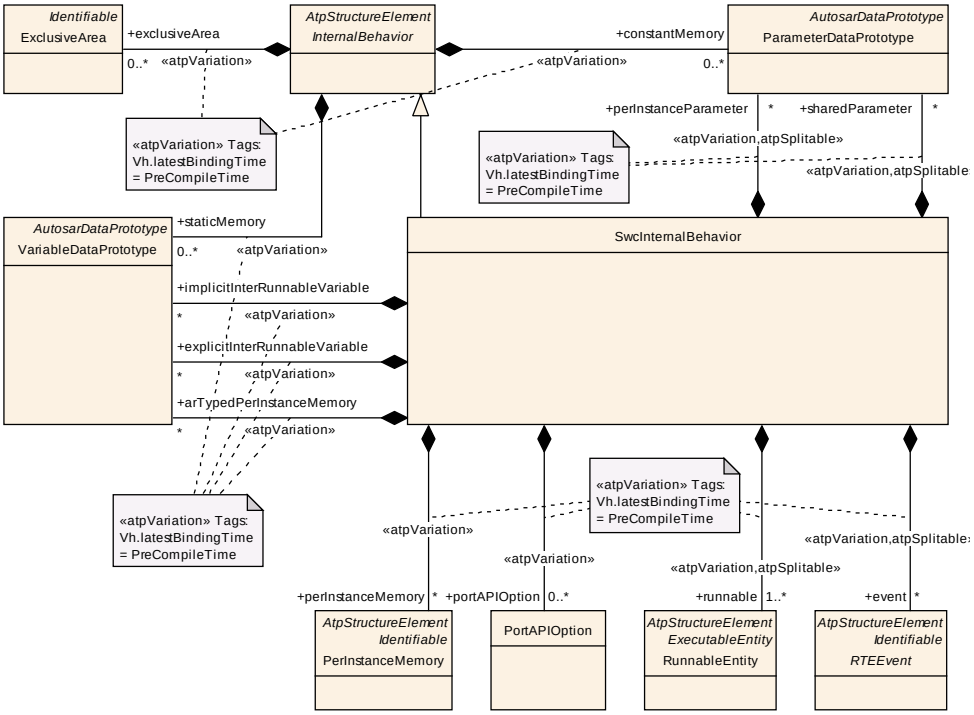


Figure 4.4: Software-component internal behavior

RunnableEntitys (also abbreviated simply as Runnable) are the smallest code fragments that are provided by AUTOSAR software-components and those basic software modules that implement *AUTOSAR Interfaces*. They are represented by the meta-class RunnableEntity, see Figure 4.5.

In general, software components are composed of multiple RunnableEntitys in order to accomplish servers, receivers, feedback, etc.

[rte_sws_2202]⌈ The RTE shall support multiple RunnableEntitys in AUTOSAR software-components. ⌋*(RTE00031)*

RunnableEntitys are executed in the context of an OS task, their execution is triggered by RTEEvents. Section 4.2.2.3 gives a more detailed description of the concept of RunnableEntitys, Section 4.2.2.6 discusses the problem of mapping

RunnableEntitys to OS tasks. RTEEvents and the activation of RunnableEntitys by RTEEvents is treated in Section 4.2.2.4.

[rte_sws_2203]⌈ The RTE shall trigger the execution of RunnableEntitys in accordance with the connected RTEEvent. ⌋*(RTE00072)*

[rte_sws_2204]⌈ The RTE Generator shall reject configurations where an RTEEvent instance which can start a RunnableEntity is not mapped to an OS task. The only exceptions are RunnableEntitys that are invoked by a direct function call. ⌋*(RTE00049, RTE00018)*

[rte_sws_7347]⌈ The RTE Generator shall reject configurations where RunnableEntitys of a SW-C are mapped to tasks of different partitions. ⌋*(RTE00036, RTE00018)*

[rte_sws_2207]⌈ The RTE shall respect the configured execution order of RunnableEntitys within one OS task. ⌋*(RTE00070)*

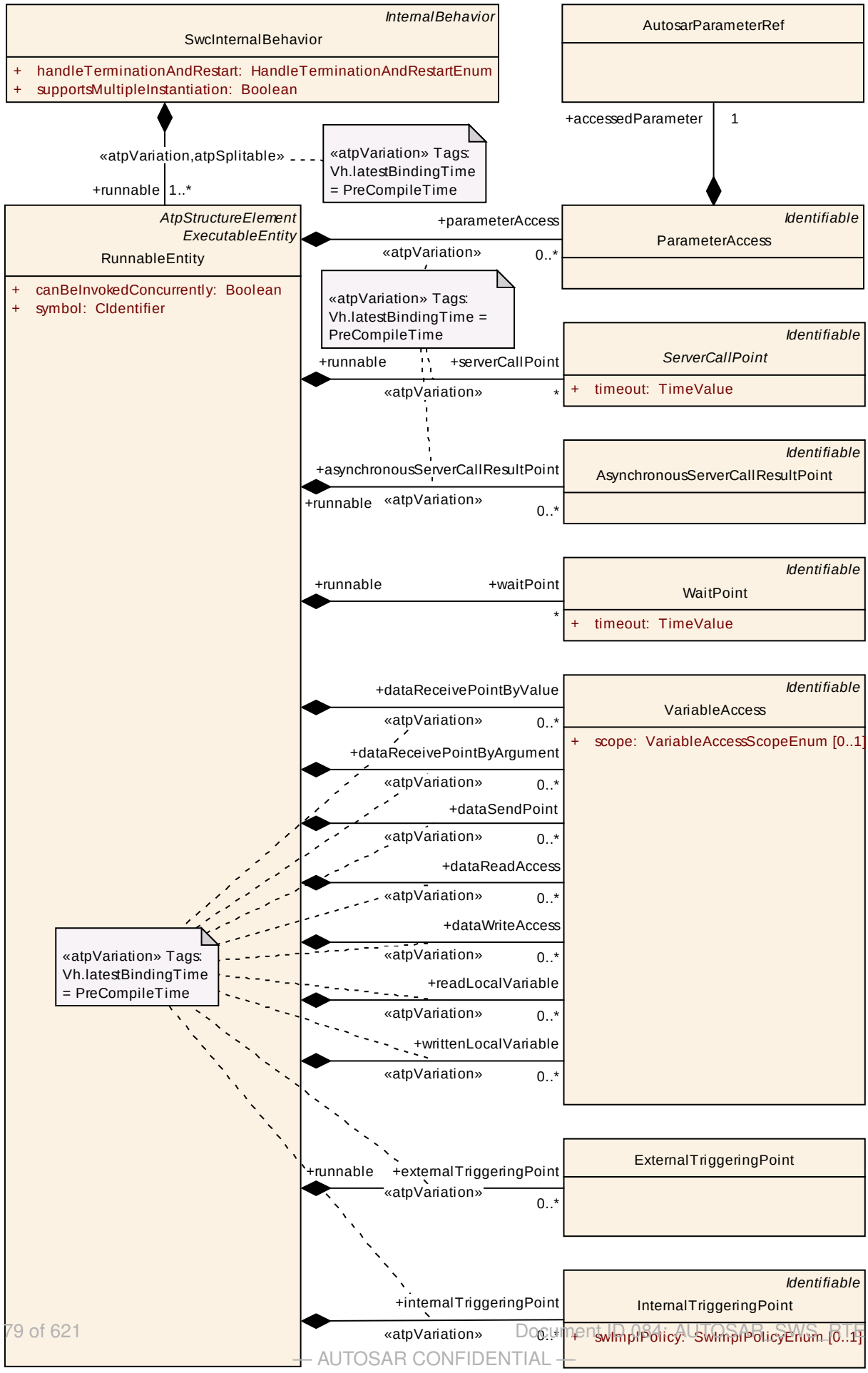

InternalBehavior
SwcInternalBehavior
+ handleTerminationAndRestart: HandleTerminationAndRestartEnum
+ supportsMultipleInstantiation: Boolean
AutosarParameterRef
+accessedParameter
1
«atpVariation,atpSplitable»
«atpVariation» Tags: Vh.latestBindingTime = PreCompileTime
+runnable
1..*
AtpStructureElement
ExecutableEntity
RunnableEntity
+ canBeInvokedConcurrently: Boolean
+ symbol: CIdentifier
+parameterAccess
«atpVariation»
0..*
Identifiable
ParameterAccess
«atpVariation» Tags: Vh.latestBindingTime = PreCompileTime
+runnable
+serverCallPoint
«atpVariation»
*
Identifiable
ServerCallPoint
+ timeout: TimeValue
+asynchronousServerCallResultPoint
+runnable
«atpVariation»
0..*
Identifiable
AsynchronousServerCallResultPoint
+runnable
+waitPoint
*
Identifiable
WaitPoint
+ timeout: TimeValue
+dataReceivePointByValue
«atpVariation»
0..*
Identifiable
VariableAccess
+ scope: VariableAccessScopeEnum [0..1]
+dataReceivePointByArgument
«atpVariation»
0..*
+dataSendPoint
«atpVariation»
0..*
+dataReadAccess
«atpVariation»
0..*
+dataWriteAccess
«atpVariation»
0..*
«atpVariation» Tags: Vh.latestBindingTime = PreCompileTime
+readLocalVariable
«atpVariation»
0..*
+writtenLocalVariable
«atpVariation»
0..*
+runnable
+externalTriggeringPoint
«atpVariation»
0..*
ExternalTriggeringPoint
+internalTriggeringPoint
«atpVariation»
0..*
Identifiable
InternalTriggeringPoint
+ swImplPolicy: SwImplPolicyEnum [0..1]

Figure 4.5: Software-component runnable entity

With the information from `SwcInternalBehavior` a part of the setup of the AUTOSAR software-component within the RTE and the OS can already be configured. Furthermore, the information (description) of the structure (ports, interfaces) and the internal behavior of an AUTOSAR software component are sufficient for the *RTE Contract Phase*.

However, some detailed information is still missing and this is part of the Implementation description.

4.1.3.4 Implementation

In the Implementation description an actual implementation of an AUTOSAR software-component is described including the memory consumption (see Figure 4.6).

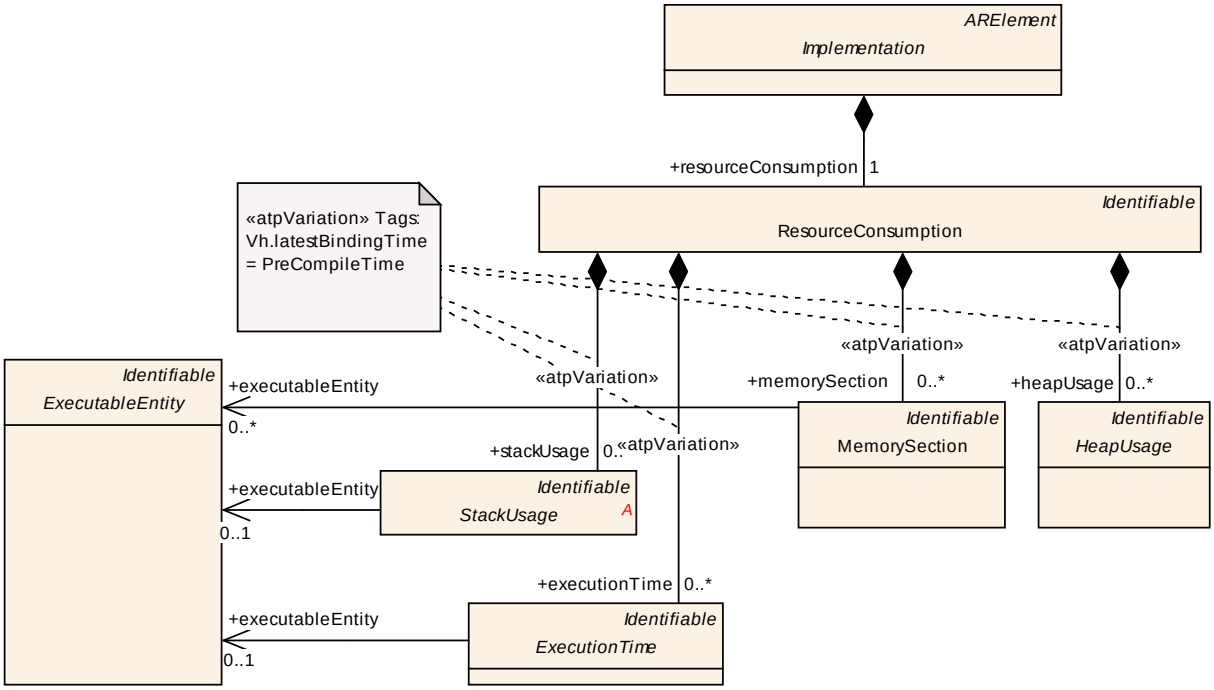


Figure 4.6: Software-component resource consumption

Note that the information from the Implementation part are only required for the *RTE Generation Phase*, if at all.

4.1.4 Instantiation

4.1.4.1 Scope and background

Generally spoken, the term *instantiation* refers to the process of deriving specific instances from a model or template. But, this process can be accomplished on different levels of abstraction. Therefore, the instance of the one level can be the model for the next.

With respect to AUTOSAR four modeling levels are distinguished. They are referred to as the levels $M3$ to $M0$.

The level $M3$ describes the concepts used to derive an AUTOSAR meta model of level $M2$. This meta model at level $M2$ defines a language in order to be able to describe specific attributes of a model at level $M1$, e.g., to be able to describe an specific type of an AUTOSAR software component. E.g., one part of the AUTOSAR meta model is called *Software Component Template* or *SW-C-T* for short and specified in [2]. It is discussed more detailed in section 4.1.3.

At level $M1$ engineers will use the defined language in order to design components or interfaces or compositions, say to describe an specific *type* of a *LightManager*. Hereby, e.g., the descriptions of the (atomic) software components will also contain an internal behavior as well as an implementation part as mentioned in section 4.1.3.

Those descriptions are input for the RTE Generator in the so-called 'Contract Phase' (see section 3.1.1). Out of this information specific APIs (in a programming language) to access ports and interfaces will be generated.

Software components generally consist of a set of Runnable Entities. They can now specifically be described in a programming language which can be refered to as "implementation". As one can see in section 4.1.3 this "implementation" then corresponds exactly to one implementation description as well as to one internal behavior description.

$M0$ refers to a specific running instance on a specific car.

Objects derived from those specified component types can only be executed in a specific run time environment (on a specific target). The objects embody the real and running implementation and shall therefore be referred to as software component instances (on modeling level $M0$). E.g., there could be two component instances derived from the same component type *LightManager* on a specific *light controller* ECU each responsible for different lights. Making instances means that it should be possible to distinguish them even though the objects are descended from the same model.

With respect to this more narrative description the *RTE* as the *run time environment* shall enable the process of instantiation. Thereby the term *instantiation* throughout the document shall refer to the process of deriving and providing explicit particular descriptions of all occuring instances of all types. Therefore, this section will address the problems which can arise out of the instantiation process and will specify the needs for AUTOSAR components and the AUTOSAR RTE respectively.

4.1.4.2 Concepts of instantiation

Regardless of the fact that the (aforementioned) instantiation of AUTOSAR software components can be generally achieved on a per-system basis, the RTE Generator restricts its view to a per-ECU customization (see rte_sws_5000).

Generally, there are two different kinds of instantiations possible:

- single instantiation – which refers to the case where only *one* object or AUTOSAR software component instance will be derived out of the AUTOSAR software component description
- multiple instantiation – which refers to the case where *multiple* objects or AUTOSAR software component instances will be derived out of the AUTOSAR software component description

[rte_sws_2001]⌈ The RTE Generator shall be able to instantiate one or more AUTOSAR software component instances out of a single AUTOSAR software component description. ⌋*(RTE00011)*

[rte_sws_2008]⌈ The RTE Generator shall evaluate the attribute *supportsMultipleInstantiation* of the *SwcInternalBehavior* of an AUTOSAR software component description. ⌋*(RTE00011)*

[rte_sws_2009]⌈ The RTE Generator shall reject configurations where multiple instantiation is required, but the value of the attribute *supportsMultipleInstantiation* of the *SwcInternalBehavior* of an AUTOSAR software component description is set to *FALSE*. ⌋*(RTE00011, RTE00018)*

4.1.4.3 Single instantiation

Single instantiation refers to the easiest case of instantiation.

To be instantiated merely means that the code and the corresponding data of a particular RunnableEntity are embedded in a runtime context. In general, this is achieved by the context of an OS task (see example 4.1).

Example 4.1

Runnable entity `R1` called out of a task context:

```
1   TASK(Task1){
2       ...
3       R1();
4       ...
5   }
```

Since the single instance of the software component is unambigous per se no additional concepts have to be added.

4.1.4.4 Multiple instantiation

[rte_sws_2002]⌈ Multiple objects instantiated from a single AUTOSAR software component (type) shall be identifiable without ambiguity. ⌋*(RTE00011)*

There are two *principle* ways to achieve this goal –

- by code duplication (of runnable entities)
- by code sharing (of reentrant runnable entities)

For now it was decided to solely concentrate on code sharing and not to support code duplication.

[rte_sws_3015]⌈ The RTE only supports multiple objects instantiated from a single AUTOSAR software component by code sharing, the RTE doesn't support code duplication. ⌋*(RTE00011, RTE00012)*

Multiple instances can share the same code, if the code is reentrant. For a multi core controller, the possibility to share code between the cores depends on the hardware.

Example 4.2 is similar to the example 4.1, but for a software-component that support multiple instantiations, and where two instances have their `R1 RunnableEntity` mapped to the same task.

Example 4.2

Runnable entity `R1` called for two instances out of the same task context:

```
1       TASK(Task1){
2           ...
3           R1(instance1);
4           R1(instance2);
5           ...
6       }
```

The same code for `R1` is shared by the different instances.

4.1.4.4.1 Reentrant code

In general, side effects can appear if the same code entity is invoked by different threads of execution running, namely tasks. This holds particularly true, if the invoked code entity inherits a state or memory by the means of static variables which are visible to all instances. That would mean that all instances are coupled by those static variables.

Thus, they affect each other. This would lead to data consistency problems on one hand. On the other – and that is even more important – it would introduce a new

communication mechanism to AUTOSAR and this is forbidden. AUTOSAR software components can only communicate via ports.

To be complete, it shall be noted that a calling code entity also inherits the reentrancy problems of its callee. This holds especially true in case of recursive calls.

4.1.4.4.2 Unambiguous object identification

[rte_sws_2015]⌈ The instantiated AUTOSAR software component objects shall be unambiguously identifiable by an *instance handle*, if multiple instantiation by sharing code is required. ⌋*(RTE00011, RTE00012)*

4.1.4.4.3 Multiple instantiation and Per-instance memory

An AUTOSAR SW-C can define internal memory only accessible by a SW-C instance itself. This concept is called PerInstanceMemory. The memory can only be accessed by the runnable entities of this particular instance. That means in turn, other instances don't have the possibility to access this memory.

PerInstanceMemory API principles are explained in Section 5.2.5.

The API for PerInstanceMemory is specified in Section 5.6.15.

4.1.5 RTE and AUTOSAR Services

According to the AUTOSAR glossary [11] "an AUTOSAR service is a logical entity of the Basic Software offering general functionality to be used by various AUTOSAR software components. The functionality is accessed via standardized AUTOSAR interfaces".

Therefore, AUTOSAR services provide standardized AUTOSAR Interfaces: ports typed by standardized *PortInterfaces*.

When connecting AUTOSAR service ports to ports of AUTOSAR software components the RTE maps standard RTE API calls to the symbols defined in the RTE input (i.e. XML) for the AUTOSAR service runnables of the BSW. The key technique to distinguish ECU dependent identifiers for the AUTOSAR services is called "port-defined argument values", which is described in Section 4.3.2.4. Currently "port-defined argument values" are only supported for client-server communication. It is not possible to use a pre-defined symbol for sending or receiving data.

The RTE does not pass an instance handle to the *C*-based API of AUTOSAR services since the latter are single-instantiatable (see rte_sws_3806).

As displayed on figure 4.2, there can be direct interactions between the RTE and some Basic Software Modules. This is the case of the Operating System, the AUTOSAR Communication, and the NVRAM Manager.

4.1.6 RTE and ECU Abstraction

The *ECU Abstraction* provides an interface to physical values for AUTOSAR software components. It abstracts the physical origin of signals (their pathes to the ECU hardware ports) and normalizes the signals with respect to their physical appearance (like specific values of current or voltage).

See the AUTOSAR ECU architecture in figure 4.2. From an architectural point of view the ECU Abstraction is part of the *Basic Software* layer and offers AUTOSAR interfaces to AUTOSAR software components.

Seen from the perspective of an RTE, regular AUTOSAR ports are connected. Without any restrictions all communication paradigms specified by the AUTOSAR Virtual Functional Bus (VFB) shall be applicable to the ports, interfaces and connections – sender-receiver just as well as client-server mechanisms.

However, ports of the ECU Abstraction shall always only be connected to ports of specific AUTOSAR software components: sensor or actuator software components. In this sense they are tightly coupled to a particular ECU Abstraction.

Furthermore, it must not be possible (by an RTE) to connect AUTOSAR ports of the ECU Abstraction to AUTOSAR ports of any AUTOSAR component located on a remote ECU (see rte_sws_2051 and [RTE00136]).

This means, e.g., that sensor-related signals coming from the ECU Abstraction are always received by an AUTOSAR sensor component located on the same ECU. The AUTOSAR sensor component will then process the received signal and deploy it to other AUTOSAR components regardless of whether they are located on the same or any remote ECU. This applies to actuator-related signals accordingly, however, the opposite way around.

[rte_sws_2050]⌈ The RTE Generator shall generate a communication path between connected ports of AUTOSAR sensor or actuator software components and the ECU Abstraction in the exact same manner like for connected ports of AUTOSAR software components. ⌋*()*

[rte_sws_2051]⌈ The RTE Generator shall reject configurations which require a communication path from a AUTOSAR software component to an ECU Abstraction located on a remote ECU. ⌋*(RTE00062, RTE00018)*

Further information about the ECU Abstraction can be found in the corresponding specification document [18].

4.1.7 RTE and Complex Device Driver

A Complex Device Driver has an AUTOSAR Interface, therefore the RTE can deal with the communication on the Complex Device Drivers ports. The Complex Device Driver is allowed to have code entities that are not under control of the RTE but yet still may use the RTE API (e.g. ISR2, BSW main processing functions).

4.1.8 Basic Software Scheduler and Basic Software Modules

4.1.8.1 Description of a Basic Software Module

The description of a Basic Software Module is divided into the sections

- interfaces
- internal behavior
- implementation

For further details see document [9].

4.1.8.2 Basic Software Interfaces

The interface of a *Basic Software Module* is described with *Basic Software Module Entries* (*BswModuleEntry*). For the functionality of the *Basic Software Scheduler* only *BswModuleEntry*s from *BswCallType SCHEDULED* are relevant. Nevertheless for optimization purpose the analysis of the full call tree might be required which requires the consideration of all *BswModuleEntry*'s

4.1.8.3 Basic Software Internal Behavior

The *Basic Software Internal Behavior* specifies the behavior of a BSW module or a BSW cluster w.r.t. the code entities visible by the BSW Scheduler. For the *Basic Software Scheduler* mainly *Basic Software Schedulable Entities* (*BswSchedulableEntity*'s) are relevant. These are *Basic Software Module Entities*, which are designed for control by the *Basic Software Scheduler*. *Basic Software Schedulable Entities* are implementing main processing functions. Furthermore all *Basic Software Schedulable Entities*

are allowed to use exclusive areas and for call tree analysis all *Basic Software Module Entities* are relevant.

[rte_sws_7514]⌈ The *Basic Software Scheduler* shall support multiple *Basic Software Module Entities* in *AUTOSAR Basic Software Modules*. ⌋*(RTE00211, RTE00213, RTE00216)*

[rte_sws_7515]⌈ The *Basic Software Scheduler* shall trigger the execution of *Schedulable Entity*'s in accordance with the connected *BswEvent*. ⌋*(RTE00072)*

[rte_sws_7516]⌈ The RTE Generator shall reject configurations where an *BswEvent* which can start a *Schedulable Entity* is not mapped to an OS task. The exceptions are *BswEvent* that are implemented by a direct function call. ⌋*(RTE00049, RTE00018)*

[rte_sws_7517]⌈ The RTE Generator shall respect the configured execution order of *Schedulable Entities* within one OS task. ⌋*(RTE00219)*

[rte_sws_7518]⌈ The RTE shall support the execution sequences of *Runnable Entities* and *Schedulable Entities* within the same OS task in an arbitrarily configurable order. ⌋*(RTE00219)*

4.1.8.4 Basic Software Implementation

The implementation defines further details of the implantation of the *Basic Software Module*. The *vendorApiInfix* attribute is of particular interest, because it defines the name space extension for multiple instances of the same basic software module. Further on the category of the `codeDescriptor` specifies if the *Basic Software Module* is delivered as source code or as object.

4.1.8.5 Multiple Instances of Basic Software Modules

In difference to the multiple instantiation concept of software components, where the same component code is used for all component instances, basic software modules are multiple instantiated by creation of own code per instance in a different name space. The attribute *vendorApiInfix* allows to define name expansions required for global symbols.

4.1.8.6 AUTOSAR Services / ECU Abstraction / Complex Device Drivers

AUTOSAR Services, *ECU Abstraction* and *Complex Device Drivers* are hybrid of AUTOSAR software-component and *Basic Software Module*. These kinds of modules might use *AUTOSAR Interfaces* to communicate via RTE as well as C-API to directly access other *Basic Software Modules*. Caused by the structure of the *AUTOSAR Meta Model* some entities of the 'C' implementation have to be described twice; on the one hand by the means of the *Software Component Template* [2] and on the other hand by

the means of the *Basic Software Module Description Template* [9]. Further on the dualism of port based communication between software component and non-port based communication between *Basic Software Modules* requires in some cases the coordination and synchronization between both principles. The information about elements belonging together is provided by the so called *SwcBswMapping*.

4.1.8.6.1 RunnableEntity / BswModuleEntity mapping

A *Runnable Entity* which is mapped to a *Basic Software Module Entity* has to be treated as one common entity. This means it describes an entity which can use the features of a *Runnable Entity* and a *Basic Software Module Entity* as well. For instance it supports to use the port based API as well as *Basic Software Scheduler* API in one C function.

4.1.8.6.2 Synchronized ModeDeclarationGroupPrototype

Two synchronized *ModeDeclarationGroupPrototype* are resulting in the implementation of one common `mode machine instance`. Consequently the call of the belonging `Rte_Switch` API and the `SchM_Switch` API are having the same effect. For optimization purpose the `Rte_Switch` API might just refer to the `SchM_Switch` API.

4.1.8.6.3 Synchronized Trigger

Two synchronized *Trigger* are behaving like one common *Trigger*. Consequently the call of the belonging `Rte_Trigger` API and the `SchM_Trigger` API are having the same effect. For optimization purpose the `Rte_Trigger` API might just refer to the `SchM_Trigger` API.

4.2 RTE and Basic Software Scheduler Implementation Aspects

4.2.1 Scope

This section describes some specific implementation aspects of an AUTOSAR RTE and the Basic Software Scheduler. It will mainly address

- the mapping of logical concepts (e.g., Runnable Entities, BSW Schedulable Entities) to technical architectures (namely, the AUTOSAR OS)
- the decoupling of pending interrupts (in the Basic Software) and the notification of AUTOSAR software components
- data consistency problems to be solved by the RTE

Therefore this section will also refer to aspects of the interaction of the AUTOSAR RTE and Basic Software Scheduler and the two modules of the AUTOSAR Basic Software with standardized interfaces (see Figure 4.7):

- the module *AUTOSAR Operating System* [19, 4, 20, 12]
- the module *AUTOSAR COM* [21, 3]

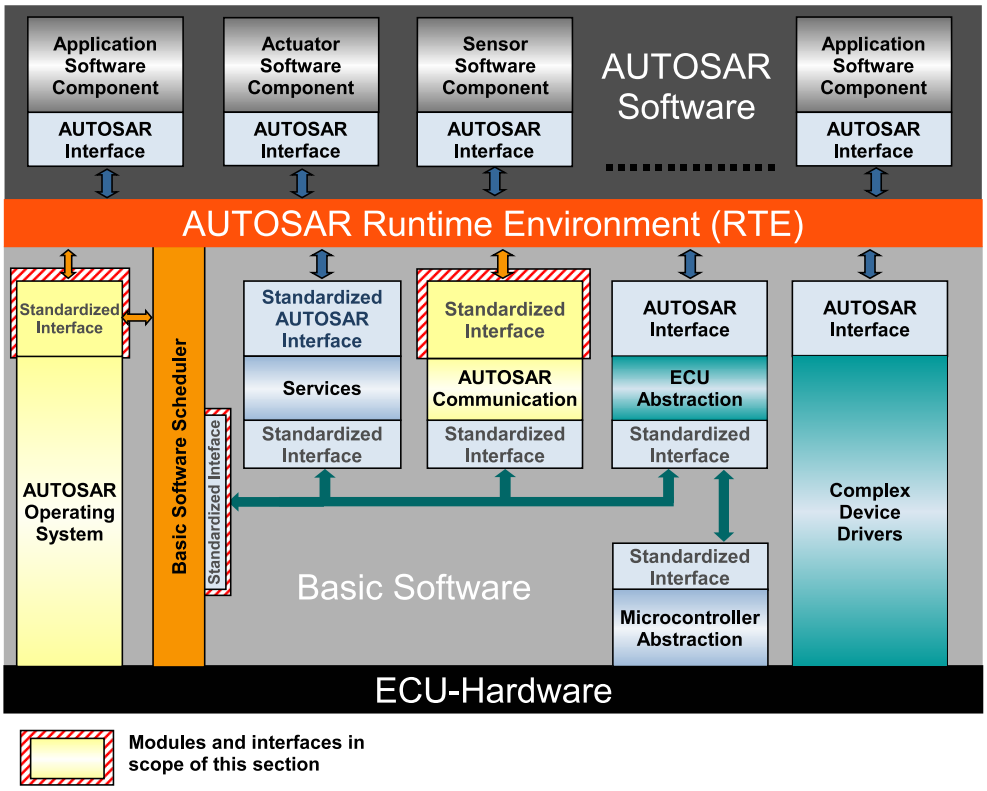


Figure 4.7: Scope of the section on Basic Software modules

Having a standardized interface means *first* that the modules do not provide or request services for/of the *AUTOSAR software components* located above the RTE. They do not have ports and therefore cannot be connected to the aforementioned AUTOSAR software components. AUTOSAR OS as well as AUTOSAR COM are simply invisible for them.

Secondly AUTOSAR OS and AUTOSAR COM are used by the RTE in order to achieve the functionality requested by the AUTOSAR software components. The AUTOSAR COM module is used by the RTE to route a signal over ECU boundaries, but this mechanism is hidden to the sending as well as to the receiving AUTOSAR software component. The AUTOSAR OS module is used for two main purposes. First, OS is used by the RTE to route a signal over core and partition boundaries. Secondly, the AUTOSAR OS module is used by the RTE in order to properly schedule the single Runnables in the sense that the RTE Generator generates Task-bodies which contain then the calls to appropriate Runnables.

In this sense the RTE shall also *use* the available means to convert interrupts to notifications in a task context or to guarantee data consistency.

With respect to this view, the RTE is *thirdly* **not** a generic abstraction layer for AUTOSAR OS and AUTOSAR COM. It is generated for a specific ECU and offers the same *interface* to the AUTOSAR Software Components as the VFB. It implements the functionality of the VFB using modules of the Basic Software, including a specific implementation of AUTOSAR OS and AUTOSAR COM.

The *Basic Software Scheduler* offers services to integrate *Basic Software Modules* for all modules of all layers. Hence, the *Basic Software Scheduler* provides the following functions:

- embed *Basic Software Modules* implementations into the *AUTOSAR OS* context
- trigger `BswSchedulableEntity`s of the *Basic Software Modules*
- apply data consistency mechanisms for the *Basic Software Modules*

The integrator's task is to apply given means (of the AUTOSAR OS) in order to assemble BSW modules in a well-defined and efficient manner in a project specific context.

This also means that the BSW Scheduler only uses the AUTOSAR OS. It is not in the least a competing entity for the AUTOSAR OS scheduler.

[rte_sws_2250]⌈ The RTE shall only use the AUTOSAR OS and AUTOSAR COM in order to provide the RTE functionality to the AUTOSAR components. ⌋*(RTE00020)*

[rte_sws_7519]⌈ The *Basic Software Scheduler* shall only use the *AUTOSAR OS* in order to provide the *Basic Software Scheduler* functionality to the *Basic Software Modules*. ⌋*()*

[rte_sws_2251]⌈ The RTE Generator shall construct task bodies for those tasks which contain `RunnableEntity`s and *Basic Software Schedulable Entities*. ⌋*(RTE00049)*

The information for the construction of task bodies has to be given by the ECU Configuration description. The mapping of *Runnable Entities* to tasks is given as an input by the ECU Configuration description. The RTE Generator does not decide on the mapping of `RunnableEntity`s to tasks.

[rte_sws_2254]⌈ The RTE Generator shall reject configurations where input information is missing regarding the mapping of *Runnable Entities* and *Basic Software Schedulable Entities* to OS tasks or the construction of tasks bodies. ⌋*(RTE00049, RTE00018)*

4.2.2 OS

This section describes the interaction between the RTE + Basic Software Scheduler and the AUTOSAR OS. The interaction is realized via the standardized interface of the OS - the AUTOSAR OS API. See Figure 4.7.

The OS is statically configured by the ECU Configuration. The RTE generator however may be allowed to create tasks and other OS objects, which are necessary for the runtime environment (see rte_sws_5150). The mapping of `RunnableEntity`s and *BSW Schedulable Entities* to OS tasks is not the job of the RTE generator. This mapping has to be done in a configuration step before, in the RTE-Configuration phase. The RTE generator is responsible for the generation of OS task bodies, which contain the calls for the `RunnableEntity`s and *BSW Schedulable Entities*. The `RunnableEntity`s and *BSW Schedulable Entities* themselves are OS independent and are not allowed to use OS service calls. The RTE and *Basic Software Scheduler* have to encapsulate such calls via the standardized RTE API respectively *Basic Software Scheduler* API.

4.2.2.1 OS Objects

Tasks

- The RTE generator has to create the task bodies, which contain the calls of the `RunnableEntity`s and `BswSchedulableEntity`s. Note that the term *task body* is used here to describe a piece of code, while the term *task* describes a configuration object of the OS.
- The RTE and *Basic Software Scheduler* controls the task activation/resumption either directly by calling OS services like `SetEvent()` or `ActivateTask()` or indirectly by initializing OS alarms or starting Schedule-Tables for time-based activation of `RunnableEntity`s. If the task terminates, the generated taskbody also contains the calls of `TerminateTask()` or `ChainTask()`.
- The RTE generator does **not** create tasks. The mapping of `RunnableEntity`s and `BswSchedulableEntity`s to tasks is the input to the RTE generator and is therefore part of the RTE Configuration.
- The RTE configurator has to allocate the necessary tasks in the OS configuration.

OS applications

- AUTOSAR OS has in R4.0 a new feature called Inter-OS-Application Communication (IOC). IOC is generated by the OS based on the configuration partially generated by the RTE. The appropriate objects (OS-Applications) are generated by the OS, and are used by RTE to for task/runnable mapping.

Events

- The RTE and *Basic Software Scheduler* may use OS Events for the implementation of the abstract `RTEEvent`s and `BswEvent`s.
- The RTE and *Basic Software Scheduler* therefore may call the OS service functions `SetEvent()`, `WaitEvent()`, `GetEvent()` and `ClearEvent()`.
- The used OS Events are part of the input information of the RTE generator.

- The RTE configurator has to allocate the necessary events in the OS configuration.

Resources

- The RTE and *Basic Software Scheduler* may use OS Resources (standard or internal) e.g. to implement data consistency mechanisms.
- The RTE and *Basic Software Scheduler* may call the OS services `GetResource()` and `ReleaseResource()`.
- The used Resources are part of the input information of the RTE generator.
- The RTE configurator has to allocate the necessary resources (all types of resources) in the OS configuration.

Interrupt Processing

- An alternative mechanism to get consistent data access is disabling/enabling of interrupts. The AUTOSAR OS provides different service functions to handle interrupt enabling/disabling. The RTE may use these functions and must **not** use compiler/processor dependent functions for the same purpose.

Alarms

- The RTE may use Alarms for timeout monitoring of asynchronous client/server calls. The RTE is responsible for Timeout handling.
- The RTE and *Basic Software Scheduler* may setup cyclic alarms for periodic triggering of `RunnableEntity`s and `BswSchedulableEntity`s (`RunnableEntity` activation via `RTEEvent TimingEvent` respectively `BswSchedulableEntity` activation via `BswEvent BswTimingEvent`)
- The RTE and *Basic Software Scheduler* therefore may call the OS service functions `GetAlarmBase()`, `GetAlarm()`, `SetRelAlarm()`, `SetAbsAlarm()` and `CancelAlarm()`.
- The used Alarms are part of the input information of the RTE generator.
- The RTE configurator has to allocate the necessary alarms in the OS configuration.

Schedule Tables

- The RTE and *Basic Software Scheduler* may setup schedule tables for cyclic task activation (e.g. `RunnableEntity` activation via `RTEEvent TimingEvent`)
- The used schedule tables are part of the input information of the RTE generator.
- The RTE configurator has to allocate the necessary schedule tables in the OS configuration.

Common OS features

Depending on the global scheduling strategy of the OS, the RTE can make decisions about the necessary data consistency mechanisms. E.g. in an ECU, where all tasks are non-preemptive - and as the result also the global scheduling strategy of the complete ECU is non-preemptive - the RTE may optimize the generated code regarding the mechanisms for data consistency.

Hook functions

The AUTOSAR OS Specification defines hook functions as follows:

A Hook function is implemented by the user and invoked by the operating system in the case of certain incidents. In order to react to these on system or application level, there are two kinds of hook functions.

- **application-specific:** Hook functions within the scope of an individual OS Application.
- **system-specific:** Hook functions within the scope of the complete ECU (in general provided by the integrator).

If no memory protection is used (scalability classes SCC1 and SCC2) only the system-specific hook functions are available.

In the SRS the requirements to implement the system-specific hook functions were rejected [RTE00001], [RTE00101], [RTE00102] and [RTE00105], as well as the application-specific hook functions [RTE00198]. The reason for the rejection is the system (ECU) global scope of those functions. The RTE is not the only user of those functions. Other BSW modules might have requirements to use hook functions as well. This is the reason why the RTE is not able to generate these functions without the necessary information of the BSW configuration.

It is intended that the implementation of the hook functions is done by the system integrator and NOT by the RTE generator.

4.2.2.2 Basic Software Schedulable Entities

`BswSchedulableEntity`s are *Basic Software Module Entities*, which are designed for control by the BSW Scheduler. `BswSchedulableEntity`s are implementing main processing functions. The configuration of the *Basic Software Scheduler* allows mapping of `BswSchedulableEntity`s to both types; basic tasks and extended tasks.

`BswSchedulableEntity`s not mapped to a `RunnableEntity` are not allowed to enter a wait state. Therefore such `BswSchedulableEntity`s are comparable to `RunnableEntity`s of category 1. `BswSchedulableEntity`s mapped to a `RunnableEntity` can enter wait states by usage of the RTE API and such `BswSchedulableEntity`s have to be treated according the classification of the mapped `RunnableEntity`. The mapping of `BswSchedulableEntity`s to a `RunnableEntity`s is typically used for *AUTOSAR Services*, *ECU Abstraction* and *Complex Device Drivers*. See sections 4.1.8.6.

4.2.2.3 Runnable Entities

The following section describes the RunnableEntitys, their categories and their task-mapping aspects. The prototypes of the functions implementing RunnableEntitys are described in section 5.7

Runnable entities are the schedulable parts of SW-Cs. With the exception of reentrant server runnables that are invoked via direct function calls, they have to be mapped to tasks. The mapping must be described in the ECU Configuration Description. This configuration - or just the RTE relevant parts of it - is the input of the RTE generator.

All RunnableEntitys are activated by the RTE as a result of an RTEEvent. Possible activation events are described in the meta-model by using RTEEvents (see section 4.2.2.4). If no RTEEvent is specified in the role startOnEvent for the RunnableEntity, the RunnableEntity is never activated by the RTE.

The categories of RunnableEntitys are described in [2].

RunnableEntities and SchedulableEntities are generalized by ExecutableEntities.

4.2.2.4 RTE Events

The meta model describes the following RTE events:

Abbreviation	Name
T	TimingEvent
BG	BackgroundEvent
DR	DataReceivedEvent (S/R Communication only)
DRE	DataReceiveErrorEvent (S/R Communication only)
DSC	DataSendCompletedEvent (explicit S/R Communication only)
DWC	DataWriteCompletedEvent (implicit S/R Communication only)
OI	OperationInvokedEvent (C/S Communication only)
ASCR	AsynchronousServerCallReturnsEvent (C/S communication only)
MS	SwcModeSwitchEvent
MSA	ModeSwitchedAckEvent
ETO	ExternalTriggerOccurredEvent
ITO	InternalTriggerOccurredEvent

According to the meta model each kind of RTEEvent can either

ACT activate a RunnableEntity, or

WUP wakeup a RunnableEntity at its WaitPoints

The meta model makes no restrictions which kind of RTEEvents are referred by WaitPoints. As a consequence RTE API functions would be necessary to set up the WaitPoints for each kind of RTEEvent.

Nevertheless in some cases it seems to make no sense to implement all possible combinations of the general meta model. E.g. setting up a WaitPoint, which should be resolved by a cyclic TimingEvent. Therefore the RTE SWS defines some restrictions, which are also described in section A.

The meta model also allows, that the same RunnableEntity can be triggered by several RTEEvents. For the current approach of the RTE and restrictions see section 4.2.6.

	T	BG	DR	DRE	DSC	DWC	OI	ASCR	MS	MSA	ETO	ITO
ACT	x	x	x	x	x	x	x	x	x	x	x	x
WUP			x		x			x		x		

The table shows, that *activation of RunnableEntity* is possible for each kind of RTEEvent. For RunnableEntity activation, no explicit RTE API in the to be activated RunnableEntity is necessary. The RTE itself is responsible for the activation of the RunnableEntity depending on the configuration in the SW-C Description.

If the RunnableEntity contains a WaitPoint, it can be resolved by the assigned RTEEvent(s). Entering the WaitPoint requires an explicit call of a RTE API function. The RTE (together with the OS) has to implement the WaitPoint inside this RTE API.

The following list shows which RTE API function has to be called to set up WaitPoints.

- DataReceivedEvent: Rte_Receive()
- DataSendCompletedEvent: Rte_Feedback()
- ModeSwitchedAckEvent: Rte_SwitchAck()
- AsynchronousServerCallReturnsEvent: Rte_Result()

[rte_sws_1292]⌈ When a DataReceivedEvent references a RunnableEntity and a required VariableDataPrototype and no WaitPoint references the DataReceivedEvent, the RunnableEntity shall be activated when the data is received. rte_sws_1135. ⌋*(RTE00072)*

Requirement rte_sws_1292 merely affects when the runnable is activated – an API call should still be created, according to requirement rte_sws_1288, rte_sws_1289, and rte_sws_7395 as appropriate, to actually read the data.

4.2.2.5 BswEvents

The meta model describes the following BswEvents.

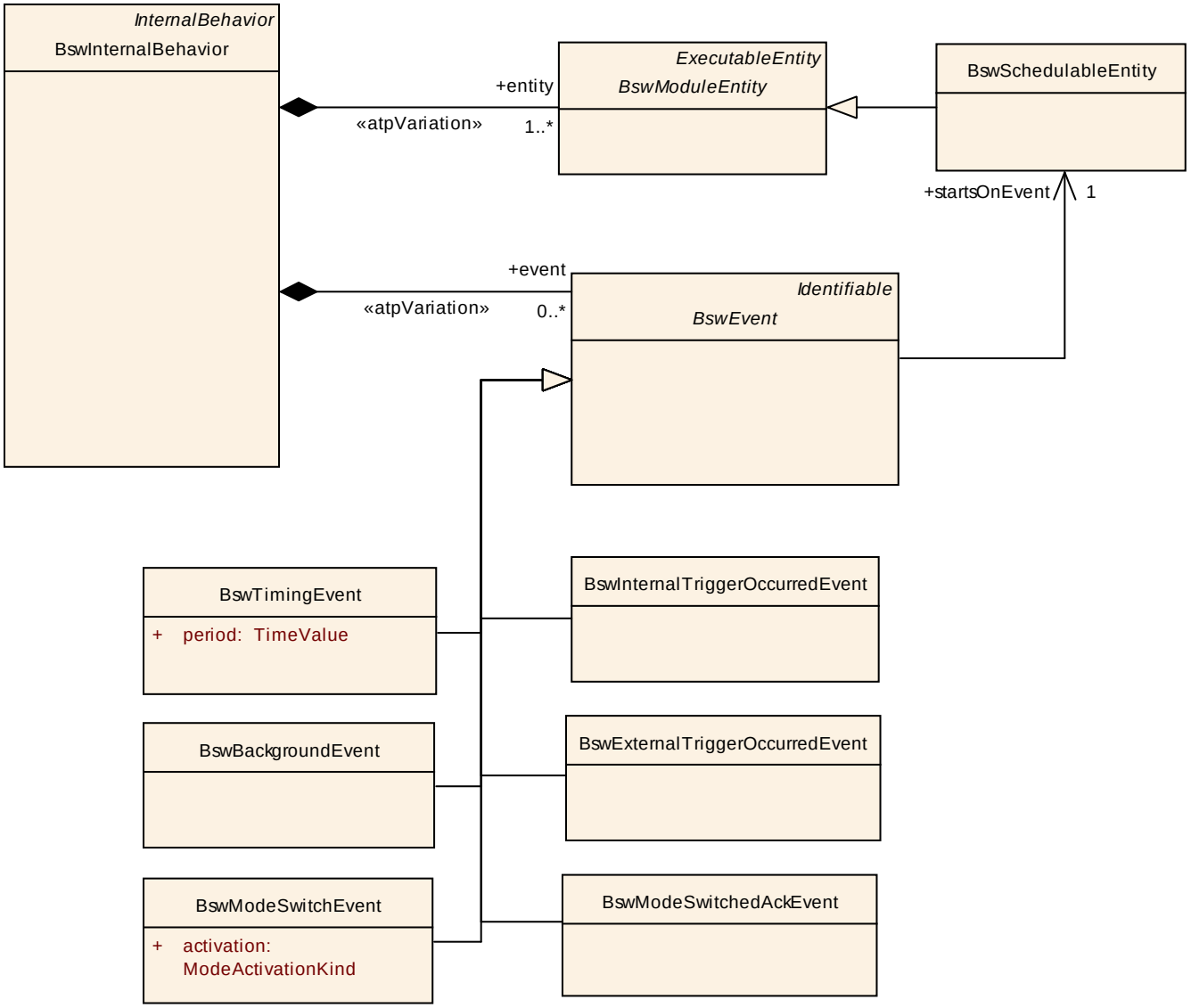


Figure 4.8: Different kinds of `BswEvents`

Similar to `RTEEvent`s the *activation of Basic Software Schedulable Entities* is possible for each kind of `BswEvent`. For of `BswSchedulableEntity`s activation, no explicit *Basic Software Scheduler* API in the to be activated `BswSchedulableEntity` is necessary. The *Basic Software Scheduler* itself is responsible for the activation of the `BswSchedulableEntity` depending on the configuration in the *Basic Software Module Description*. In difference to `RTEEvent`s, none of the `BswEvent`s support `WaitPoint`s. For more details see document [9].

4.2.2.6 Mapping of Runnable Entities and Basic Software Schedulable Entities to tasks (informative)

One of the main requirements of the RTE generator is "Construction of task bodies" [RTE00049]. The necessary input information e.g. the mapping of `RunnableEntity`s and `BswSchedulableEntity` to tasks must be provided by the ECU configuration description.

The ECU configuration description (or an extract of it) is the input for the RTE Generator (see Figure 3.4). It is also the purpose of this document to define the necessary input information. Therefore the following scenarios may help to derive requirements for the ECU Configuration Template as well as for the RTE-generator itself.
Note: The scenarios do not cover all possible combinations.

The RTE-Configurator uses parts of the ECU Configuration of other BSW Modules, e.g. the mapping of `RunnableEntity`s to `OsTask`s. In this configuration process the RTE-Configurator expects OS objects (e.g. Tasks, Events, Alarms...) which are used in the generated RTE and *Basic Software Scheduler*.

Some figures for better understanding use the following conventions:

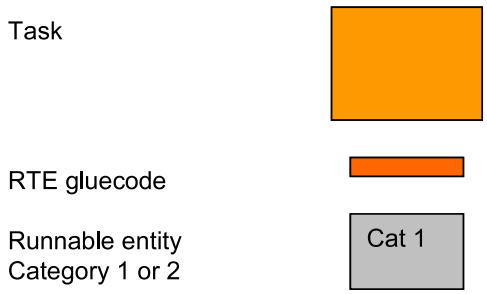


Figure 4.9: Element description

Note: The following examples are only showing `RunnableEntity`s. But taking the categorization of `BswSchedulableEntity`s defined in section 4.2.2.2 into account, the scenarios are applicable for `BswSchedulableEntity`s as well.

4.2.2.6.1 Scenario for mapping of `RunnableEntity`s to tasks

The different properties of `RunnableEntity`s with respect to data access and termination have to be taken into account when discussing possible scenarios of mapping `RunnableEntity`s to tasks.

- `RunnableEntity`s using `VariableAccess`es in the `dataReadAccess` or `dataWriteAccess` roles (implicit read and send) have to terminate.
- `RunnableEntity`s of category 1 can be mapped either to basic or extended tasks. (see next subsection).

- RunnableEntitys using at least one WaitPoint are of category 2.
- RunnableEntitys of category 2 that contain WaitPoints will be typically mapped to extended tasks.
- RunnableEntitys that contain a SynchronousServerCallPoint generally have to be mapped to extended tasks.
- RunnableEntitys that contain a SynchronousServerCallPoint can be mapped to basic tasks if no timeout monitoring is required and the server runnable is on the same partition.
- RunnableEntitys that contain a SynchronousServerCallPoint can be mapped to basic tasks if the server runnable is invoked directly and is itself of category 1.

Note that the runnable to task mapping scenarios supported by a particular RTE implementation might be restricted.

4.2.2.6.1.1 Scenario 1

Runnable entity category 1A: "runnable1"

- Ports: only S/R with VariableAccesses in the dataReadAccess or dataWriteAccess role
- RTEEvents: TimingEvent
- no sequence of RunnableEntitys specified
- no VariableAccess in the dataSendPoint role
- no WaitPoint

Possible mappings of "runnable1" to tasks:

Basic Task
If only one of those kinds of RunnableEntitys is mapped to a task (task contains only one RunnableEntity), or if multiple RunnableEntitys with the same activation period are mapped to the same task, a basic task can be used. In this case, the execution order of the RunnableEntitys within the task is necessary. In case the RunnableEntitys have different activation periods, the RTE has to provide the glue-code to guarantee the correct call cycle of each RunnableEntity.

The ECU Configuration-Template has to provide the sequence of RunnableEntitys mapped to the same task, see PositionInTask.

Figure 4.10 shows the possible mappings of RunnableEntitys into a basic task. If and only if a sequence order is specified, more than one RunnableEntity can be mapped into a basic task.

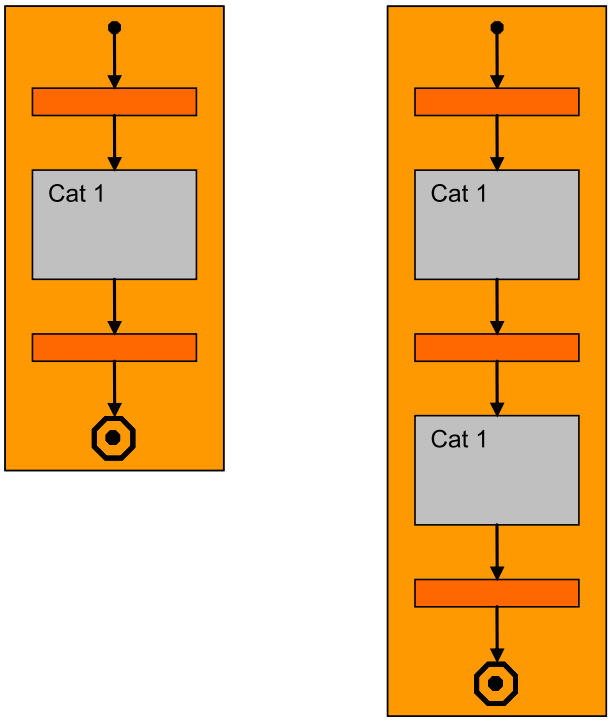


Figure 4.10: Mapping of Category 1 `RunnableEntity`s to Basic Tasks

Extended Task

If more than one `RunnableEntity` is mapped to the same task and the special condition (same activation period) does not fit, an extended task is used.

If an extended task is used, the entry points to the different `RunnableEntity`s might be distinguished by evaluation of different OS events. In the scenario above, the different activation periods may be provided by different OS alarms. The corresponding OS events have to be handled inside the task body. Therefore the RTE-generator needs for each task the number of assigned OS Events and their names.

The ECU Configuration has to provide the OS events assigned to the `RTEEvent`s triggering the `RunnableEntity`s that are mapped to an extended task, see `UsedOsEventRef`.

Figure 4.11 shows the possible mapping of the multiple `RunnableEntity`s of category 1 into an Extended Task. Note: The Task does not terminate.

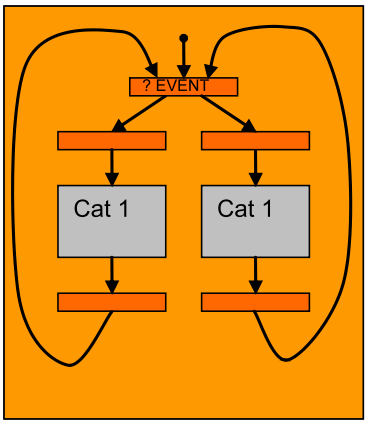


Figure 4.11: Mapping of Category 1 `RunnableEntity`s to Extended Tasks

For both, basic tasks and extended tasks, the ECU Configuration must provide the name of the task.

The ECU Configuration has to provide the name of the task, see `OsTask`.

The ECU Configuration has to provide the task type (BASIC or EXTENDED), which can be determined from the presence or absence of OS Events associated with that task, see `OsTask`.

4.2.2.6.1.2 Scenario 2

Runnable entity category 1B: "runnable2"

- Ports: S/R with `VariableAccess`es in the `dataSendPoint` role.
- `RTEEvent`s: TimingEvent
- no `WaitPoint`

Possible mappings of "runnable2" to tasks:

The following figure shows the different mappings:

- One category 1B runnable
- More than one category 1B runnable mapped to the same basic task with a specified sequence order
- More than one category 1B runnable mapped into an extended task

The gluecode to realize the `VariableAccess`in the `dataReadAccess` and `dataWriteAccess` roles respectively before entering the runnable and after exiting is not necessary.

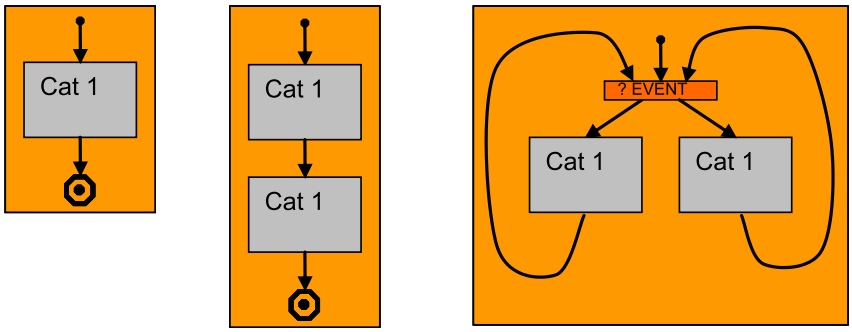


Figure 4.12: Mapping of Category 1 `RunnableEntity`s using no `VariableAccesses` in the `dataReadAccess` or `dataWriteAccess` role

4.2.2.6.1.3 Scenario 3

Runnable entity category 1A: "runnable3"

- Ports: S/R with `VariableAccesses` in the `dataReadAccess` or `dataWriteAccess` role
- `RTEEvents`: Runnable is activated by a `DataReceivedEvent`
- no `VariableAccess` in the `dataSendPoint` role
- no `WaitPoint`

There is no difference between Scenario 1 and 3. Only the RTEEvent that activates the `RunnableEntity` is different.

4.2.2.6.1.4 Scenario 4

Runnable entity category 2: "runnable4"

- Ports: S/R with `VariableAccesses` in the `dataReceivePointByValue` or `dataReceivePointByArgument` role and `WaitPoint` (blocking read)
- `RTEEvents`: `WaitPoint` referencing a `DataReceivedEvent`

Runnable is activated by an arbitrary `RTEEvent` (e.g. by a `TimingEvent`). When the `RunnableEntity` has entered the `WaitPoint` and the `DataReceivedEvent` occurs, the `RunnableEntity` resumes execution.

The runnable has to be mapped to an extended task. Normally each category 2 runnable has to be mapped to its own task. Nevertheless it is not forbidden to map multiple category 2 `RunnableEntity`s to the same task, though this might be restricted by an RTE generator. Mapping multiple category 2 `RunnableEntity`s to the same task can lead to big delay times if e.g. a `WaitPoint` is resolved by the incoming `RTEEvent`, but the task is still waiting at a different `WaitPoint`.

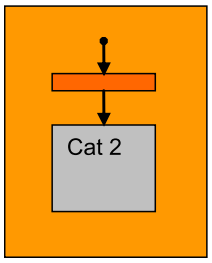


Figure 4.13: Mapping of Category 2 `RunnableEntity`s to Extended Tasks

4.2.2.6.1.5 Scenario 5

There are two `RunnableEntity`s implementing a client (category 2) and a server for synchronous C/S communication and the timeout attribute of the ServerCallPoint is 0.

On a single core, there are two ways to invoke a server synchronously:

- Simple function call for intra-partition C/S communication if the `canBeInvokedConcurrently` attribute of the server runnable is set and if the server runnable is of category 1. In that case the server runnable is executed in the same task context (same stack) as the client runnable that has invoked the server. The client runnable can be mapped to a basic task.
- The server runnable is mapped to its own task. If the `canBeInvokedConcurrently` attribute is not set, the server runnable must be mapped to a task.

  If the implementation of the synchronous server invocation does not use OS events, the client runnable can be mapped to a basic task and the task of the server runnable must have higher priority than the task of the client runnable. Furthermore, the task to which the client runnable is mapped must be preemptable. This has to be checked by the RTE generator. Activation of the server runnable can be done by ActivateTask() for a basic task or by SetEvent() for an extended task. In both cases, the task to be activated must have higher priority than the task of the client runnable to enforce a task switch (necessary, because the server invocation is synchronous).

4.2.2.6.1.6 Scenario 6

There are two `RunnableEntity`s implementing a client (category 2) and a server for synchronous C/S communication and the timeout attribute of the ServerCallPoint is greater than 0.

There are again two ways to invoke a server synchronously:

- Simple function call for intra-partition C/S communication if the `canBeInvokedConcurrently` attribute of the server runnable is set and the server is of category 1. In that case the server runnable is executed in the same task context

(same stack) as the client runnable that has invoked the server and no timeout monitoring is performed (see rte_sws_3768). In this case the client runnable can be mapped to a basic task.

- The server runnable is mapped to its own task. If the `canBeInvokedConcurrently` attribute is not set, the server runnable must be mapped to a task.

  If the implementation of the timeout monitoring uses OS events, the task of the server runnable must have lower priority than the task of the client runnable and the client runnable must be mapped to an extended task. Furthermore, both tasks must be preemptable[1]. This has to be checked by the RTE generator. The notification that a timeout occurred is then notified to the client runnable by using an OS Event. In order for the client runnable to immediately react to the timeout, a task switch to the client task must be possible when the timeout occurs.

4.2.2.6.1.7 Scenario 7

Runnable entity category 2: "runnable7"

- Ports: only C/S with AsynchronousServerCallPoint and `WaitPoint`
- `RTEEvent`s: AsynchronousServerCallReturnsEvent (C/S communication only)

The mapping scenario for "runnable7", the client runnable that collects the result of the asynchronous server invocation, is similar to Scenario 4.

[1] Strictly speaking, this restriction is not necessary for the task to which the client runnable is mapped. If OS events are used to implement the timeout monitoring and the notification that the server is finished, the RTE API implementation generally uses the OS service WaitEvent, which is a point of rescheduling.

4.2.2.7 Monitoring of runnable execution time

This section describes how the monitoring of RunnableEntity execution time can be done.

The RTE doesn't directly support monitoring of RunnableEntities execution time but the AUTOSAR OS support for monitoring of OsTasks execution time can be used for this purpose.

If execution time monitoring of a RunnableEntity is required a possible solution is to map the RunnableEntity alone to an OsTask and to configure the OS to monitor the execution time of the OsTask.

This solution can lead to dispatch to individual OsTasks RunnableEntities that should be initially mapped to the same OsTask because of for example:

- requirements on execution order of the RunnableEntities and/or
- requirements on evaluation order of the RTEEvents that activate the RunnableEntities and
- constraints to have no preemption between the RunnableEntities

In order to keep the control on the execution order of the RunnableEntities, the evaluation order of the RTEEvents and the non-preemption between the RunnableEntities when then RunnableEntities are individually mapped to several OsTasks for the purpose of monitoring, a possible solution is to replace the calls to the C-functions of the RunnableEntities by activations of the OsTasks to which the monitored RunnableEntities are mapped.

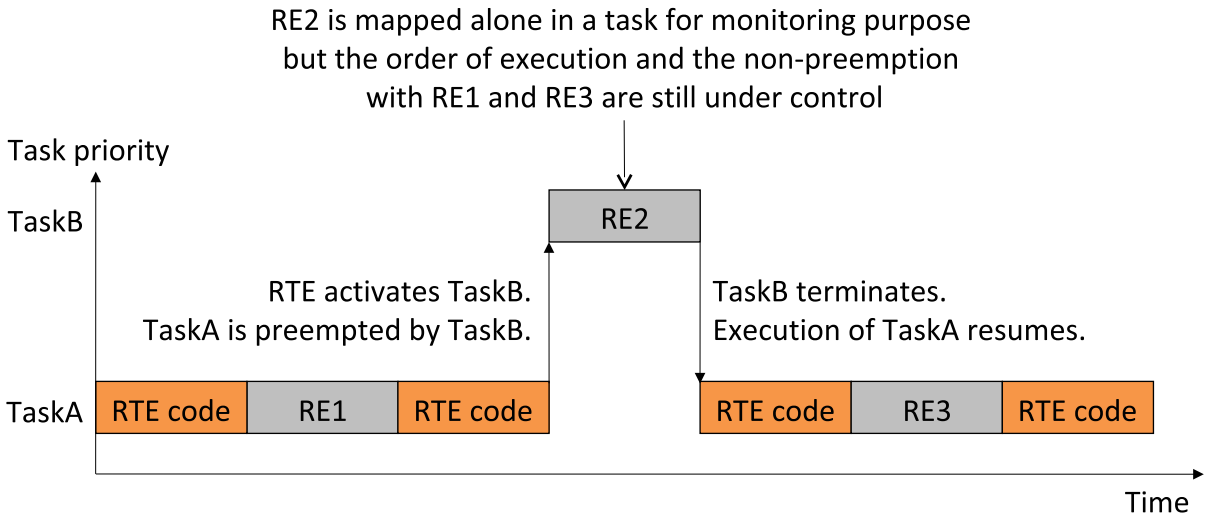


Figure 4.14: Inter task activation and mapping of runnable to individual task for monitoring purpose

This behavior of the RTE can be configured with the attributes RteVirtuallyMappedToTaskRef of the RteRunnableEventToTaskMapping. RteVirtuallyMappedToTaskRef references the OsTask in which the execution order of

the RunnableEntities and/or the evaluation order of the RTEEvents are controlled. RteMappedToTaskRef references the individual OsTasks to which the RunnableEntities are mapped for the purpose of monitoring.

[rte_sws_7800]⌈ The RTE Generator shall respect the configured virtual runnable to task mapping (RteVirtuallyMappedToTaskRef) in the RTE configuration. ⌋*(RTE00193)*

Of course this solution requires that the task priorities and scheduling properties are well configured in the OS to allow immediate preemption by the OsTasks to which the monitored RunnableEntities are mapped. A possible solution is:

- Priority of the OsTask to which the RunnableEntity is mapped is higher than the priority of the OsTask to which the RunnableEntity is virtually mapped and
- the OsTask to which the RunnableEntity is virtually mapped have a full preemptive scheduling or
- the RTE call the OS service Schedule() just after activation of the OsTask to which the RunnableEntity is mapped

Example 1: Without OsEvent

Description of the example:
RunnableEntity RE1 is activated by TimingEvent 100ms T1.
RunnableEntity RE2 is activated by TimingEvent 100ms T2.
RunnableEntity RE3 is activated by TimingEvent 100ms T3.
Execution order of the RunnableEntities shall be R1, R2 then R3.
RE2 shall be monitored.

Possible RTE configuration:
RE1/T1 is mapped to OsTask TaskA with RtePositionInTask equal to 1.
RE2/T2 is mapped to OsTask TaskB but virtually mapped to TaskA with RtePositionInTask equal to 2.
RE3/T3 is mapped to OsTask TaskA with RtePositionInTask equal to 3.

Possible RTE implementation:
RTE starts cyclic OsAlarm with 100ms period.
This OsAlarm is configured to activate TaskA.
Non preemptive scheduling is configured for Task A.
TaskB priority = TaskA priority + 1

```
1 void TaskA(void)
2 {
3     RE1();
4     ActivateTask(TaskB);
5     Schedule();
6     RE3();
7     TerminateTask();
8 }
```

```
9
10 void TaskB(void)
11 {
12    RE2();
13    TerminateTask();
14 }
```

Example 2: With `OsEvent`

Description of the example:
`RunnableEntity` RE1 is activated by `DataReceivedEvent` DR1.
`RunnableEntity` RE2 is activated by `DataReceivedEvent` DR2.
`RunnableEntity` RE3 is activated by `DataReceivedEvent` DR3.
Evaluation order of the `RTEEvent`s shall be DR1, DR2 then DR3.
All the runnables shall be monitored.

Possible RTE configuration:
RE1 is mapped to `OsTask` TaskB but virtually mapped to TaskA with a reference to `OsEvent` EvtA and `RtePositionInTask` equal to 1.
RE2 is mapped to `OsTask` TaskC but virtually mapped to TaskA with a reference to `OsEvent` EvtB and `RtePositionInTask` equal to 2.
RE3 is mapped to `OsTask` TaskD but virtually mapped to TaskA with a reference to `OsEvent` EvtC and `RtePositionInTask` equal to 3.

Possible RTE implementation:
RTE set EvtA, EvtB and EvtC according to the callbacks from COM.
Full preemptive scheduling is configured for Task A.
TaskB priority = TaskC priority = TaskD priority = TaskA priority + 1

```
1  void TaskA(void)
2  {
3     EventMaskType Event;
4
5     while(1)
6     {
7        WaitEvent(EvtA | EvtB | EvtC);
8        GetEvent(TaskA, &Event);
9        if (Event & EvtA)
10       {
11          ClearEvent(EvtA);
12          ActivateTask(TaskB);
13       }
14       else if (Event & EvtB)
15       {
16          ClearEvent(EvtB);
17          ActivateTask(TaskC);
18       }
19       else if (Event & EvtC)
20       {
21          ClearEvent(EvtC);
```

```
22            ActivateTask(TaskD);
23        }
24    }
25  }
26
27  void TaskB(void)
28  {
29     RE1();
30     TerminateTask();
31  }
32
33  void TaskC(void)
34  {
35     RE2();
36     TerminateTask();
37  }
38
39  void TaskD(void)
40  {
41     RE3();
42     TerminateTask();
43  }
```

It is also possible to configure the RTE for the monitoring of group of runnable = monitoring of the sum of the runnable execution times.

Example 3: Monitoring of group of runnables

Description of the example:
`RunnableEntity` RE1 is activated by `TimingEvent` 100ms T1.
`RunnableEntity` RE2 is activated by `TimingEvent` 100ms T2.
`RunnableEntity` RE3 is activated by `TimingEvent` 100ms T3.
`RunnableEntity` RE4 is activated by `DataReceivedEvent` DR1.
`RunnableEntity` RE5 is activated by `DataReceivedEvent` DR2.
`RunnableEntity` RE6 is activated by `DataReceivedEvent` DR3.
`RunnableEntity` RE7 is activated by `DataReceivedEvent` DR4.
`DataReceivedEvent` DR2, DR3 and DR4 references the same `DataElement`. Evaluation order of the `RTEEvent`s shall be T1, T2, T3, DR1, DR2, DR3 then DR4.
RE2 and RE3 shall be monitored as a group.
RE6 and RE7 shall be monitored as a group.

Possible RTE configuration:
RE1 is mapped to `OsTask` TaskA with a reference to `OsEvent` EvtA and `RtePositionInTask` equal to 1.
RE2 is mapped to `OsTask` TaskB but virtually mapped to TaskA with a reference to `OsEvent` EvtA and `RtePositionInTask` equal to 2.
RE3 is mapped to `OsTask` TaskB but virtually mapped to TaskA with a reference to `OsEvent` EvtA and `RtePositionInTask` equal to 3.
RE4 is mapped to `OsTask` TaskA with a reference to `OsEvent` EvtB and `RtePosi-`

tionInTask equal to 4.
RE5 is mapped to OsTask TaskA with a reference to OsEvent EvtC and RtePositionInTask equal to 5.
RE6 is mapped to OsTask TaskC but virtually mapped to TaskA with a reference to OsEvent EvtC and RtePositionInTask equal to 6.
RE7 is mapped to OsTask TaskC but virtually mapped to TaskA with a reference to OsEvent EvtC and RtePositionInTask equal to 7.

Possible RTE implementation:
RTE starts cyclic OsAlarm with 100ms period.
This OsAlarm is configured to set EvtA.
RTE set EvtB and EvtC according to the callbacks from COM.
Full preemptive scheduling is configured for Task A.
TaskB priority = TaskC priority = TaskA priority + 1

```
void TaskA(void)
{
   EventMaskType Event;

   while(1)
   {
      WaitEvent(EvtA | EvtB | EvtC);
      GetEvent(TaskA, &Event);
      if (Event & EvtA)
      {
         ClearEvent(EvtA);
         RE1();
         ActivateTask(TaskB);
      }
      else if (Event & EvtB)
      {
         ClearEvent(EvtB);
         RE4();
      }
      else if (Event & EvtC)
      {
         ClearEvent(EvtC);
         RE5();
         ActivateTask(TaskC);
      }
   }
}

void TaskB(void)
{
   RE2();
   RE3();
   TerminateTask();
}
```

```
35
36 void TaskC(void)
37 {
38    RE6();
39    RE7():
40    TerminateTask();
41 }
```

4.2.2.8 Synchronization of TimingEvent activated runnables

This section describes how the synchronization of TimingEvent activated RunnableEntities can be done.

The following cases have to be distinguished:

- the RunnableEntities are mapped to the same OsTask
- the RunnableEntities are mapped to different OsTasks in the same OsApplication
- the RunnableEntities are mapped to different OsTasks in different OsApplications on the same core
- the RunnableEntities are mapped to different OsTasks in different OsApplications on different cores on the same microcontroler
- the RunnableEntities are mapped to different OsTasks in different OsApplications on different microcontrolers within the same ECU
- the RunnableEntities are mapped to different OsTasks in different OsApplications on different microcontrolers within different ECUs

As OsAlarms and OsScheduleTableExpiryPoints are used to implement TimingEvents the following different possible solutions exist to synchronize the RunnableEntities according to the different cases:

- use the same OsAlarm or OsScheduleTableExpiryPoint to implement all the TimingEvents
- use different OsAlarms or OsScheduleTableExpiryPoints in different OsScheduleTables based on the same OsCounter and start them with absolute start offset to control the synchronization between them
- use different OsScheduleTableExpiryPoints in different explicitely synchronized OsScheduleTables based on different OsCounters but with same period and max value

The choice of the OsAlarms or OsScheduleTableExpiryPoints used to implement the TimingEvents can be configured in the RTE with RteUsedOsAlarmRef or RteUsedOsSchTblExpiryPointRef in the RteEventToTaskMapping.

[rte_sws_7804]⌈ The RTE Generator shall respect the configured OsAlarms (RteUsedOsAlarmRef) and OsScheduleTableExpiryPoints (RteUsedOsSchTblExpiryPointRef) for the implementation of the TimingEvents. ⌋*(RTE00232)*

The choice of the absolute start offset of the OsAlarms and OsScheduleTables can be configured in the RTE with RteExpectedActivationOffset in the RteUsedOsActivation.

[rte_sws_7805]⌈ The RTE Generator shall respect the configured absolute start offset (RteExpectedActivationOffset) when it starts the OsAlarms and OsScheduleTables used for the implementation of the TimingEvents. ⌋*(RTE00232)*

The RTE / *Basic Software Scheduler* is not responsible to synchronize/desynchronize the explicitly synchronized OsScheduleTables. The RTE / *Basic Software Scheduler* is only responsible to start the explicitly synchronized OsScheduleTables. In this case no RteExpectedActivationOffset has to be configured.

4.2.2.9 BackgroundEvent activated Runnable Entities and BasicSoftware Scheduleable Entities

A BackgroundEvent is a recurring RTEEvent / BswEvent which is used to perform background activities in *RunnableEntities* or BswSchedulableEntitys. It is similar to a TimingEvent but has no fixed time period and is typically activated only with lowest priority.

A BackgroundEvent triggering can be implemented in two principle ways by the RTE Generator. Either the background activation is done by a real background OS task; or the BackgroundEvents are activated like TimingEvents on a fixed recurrence which is defined by the ECU integrator (see rte_sws_7179 and rte_sws_7180). The second way might be required to overcome the limitation of a single real background OS task if BackgroundEvents are used in several partitions.

If the background activation is done by a real background OS task, the OS Task has to have the lowest priority on the CPU core (see rte_sws_7181). If a implementation is used where the OS Task terminates (*BasicTask*) the background OS Task is immediately reactivated after its termination, e.g. by usage of *ChainTask* call of the OS.

4.2.3 Activation and Start of ExecutableEntitys

This section defines the activation of ExecutableEntity execution-instances by using a state machine (Fig. 4.15).

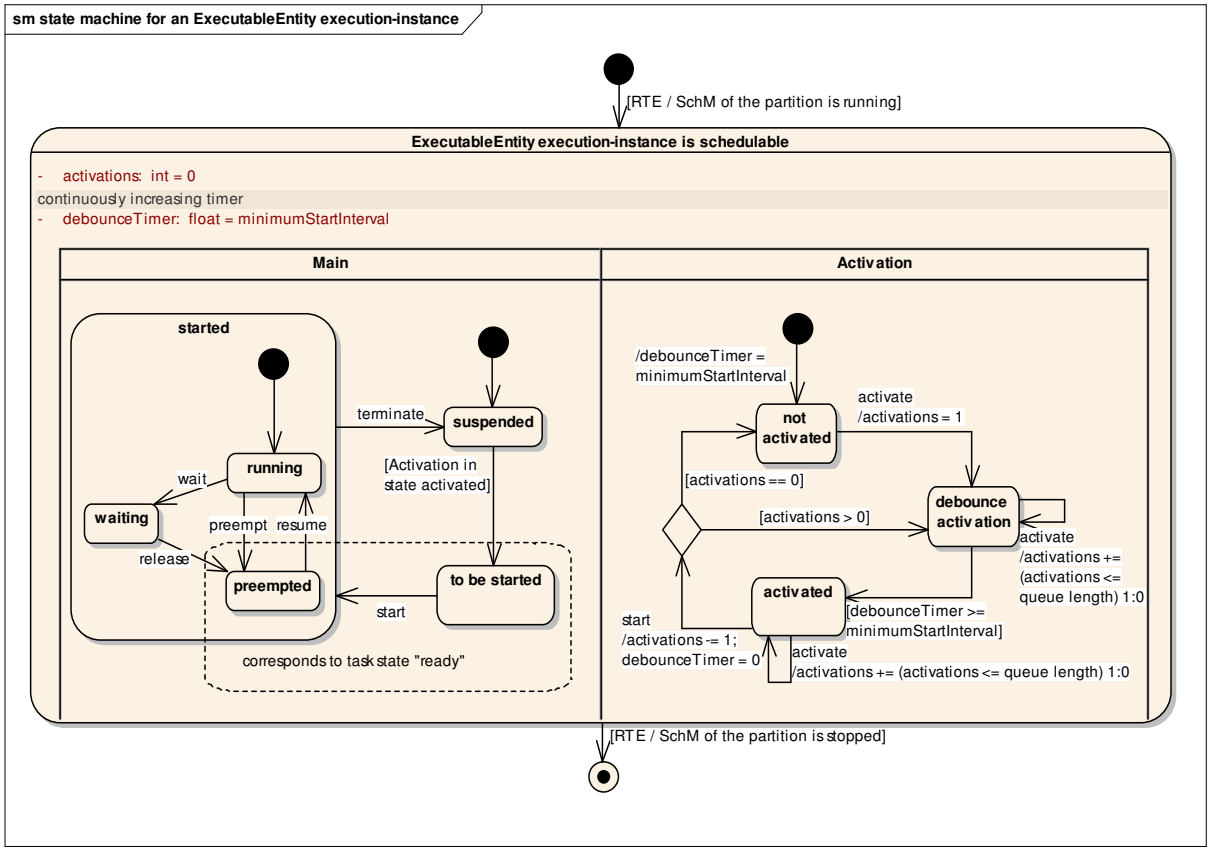


Figure 4.15: General state machine of an ExecutableEntity execution-instance.

An ExecutableEntity execution-instance is one execution-instance of an ExecutableEntity (RunnableEntity or BswSchedulableEntity) with respect to concurrent execution.

For a RunnableEntity with canBeInvokedConcurrently = false or for a BswSchedulableEntity whose referenced BswModuleEntry in the role implementedEntry has a isReentrant attribute set to false, there is only one execution-instance. For a RunnableEntity with canBeInvokedConcurrently = true or for a BswSchedulableEntity whose referenced BswModuleEntry in the role implementedEntry has its isReentrant attribute set to true, there is a well defined number of execution-instances.

E.g., for a server runnable that is executed as direct function call, each ServerCallPoint relates to exactly one ExecutableEntity execution-instance.

The main principles for the activation of runnables are:

- RunnableEntitys are activated by RTEEvents

- BswSchedulableEntitys are activated by BswEvents
- only server runnables (RunnableEntitys activated by an OperationInvokedEvent) are queued. All other ExecutableEntities are unqueued.

  If a RunnableEntity is activated due to several DataReceivedEvents of dataElements with swImplPolicy = queued, it is the responsibility of the RunnableEntity to dequeue all queued data.
- A minimumStartInterval will delay the activation of RunnableEntitys and BswSchedulableEntitys to prevent that a RunnableEntity or a BswSchedulableEntity is started more than once within the minimumStartInterval.

Each ExecutableEntity execution-instance has its own state machine. The full state machine is shown in Fig. 4.15.

Note on Figure 4.15: the debounce timer debounceTimer is an increasing timer. It is local to the ExecutableEntity execution-instance. The activation counter activations is a local integer to count the pending activations. The runnable debounce timer and the activation counter are like the whole state machine just concepts for the specification of the behavior, not for the implementation.

The pending activations are only counted for `server runnables` when RTE implements a serialization of their invocation. In all other cases, RTE does not queue activations and the state machine for the activation of `ExecutableEntity execution-instances` simplifies as shown in Figure 4.16.

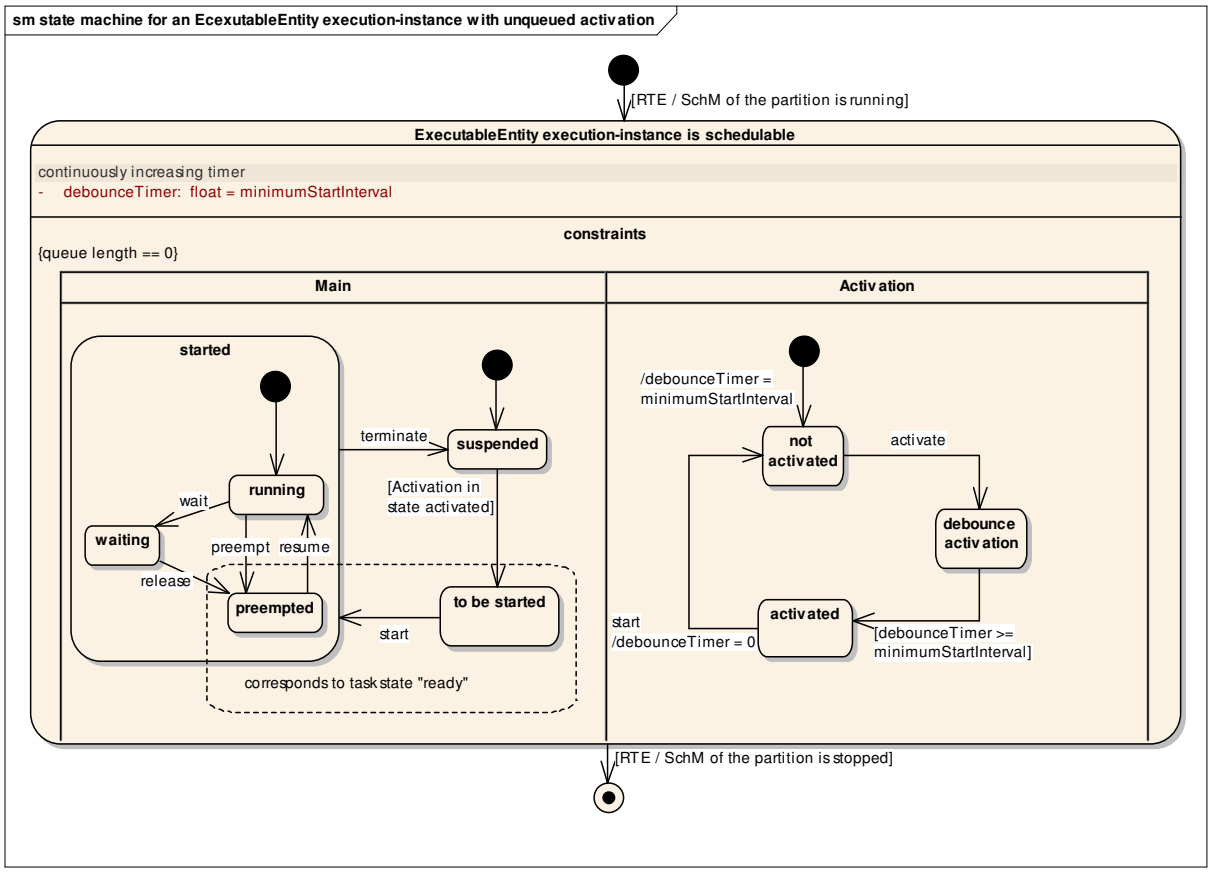


Figure 4.16: Statemachine of an unqueued execution-instance (not a server runnable)

If RTE implements an `ExecutableEntity execution-instance` by direct function call, as described in section 4.2.3.1, the simplified state machine is shown in Figure 4.19.

The state machine of an `ExecutableEntity execution-instance` is not identical to that of the task containing the `ExecutableEntity execution-instance`, but there are dependencies between them. E.g., the `ExecutableEntity execution-instance` can only be 'running' when the corresponding task is 'running'.

Table 4.1 describes all `ExecutableEntity execution-instance` states in detail. The `ExecutableEntity execution-instance` state machine is split in two threads. The Main states describe the real state of the `ExecutableEntity execution-instance` and the transitions between a suspended and a running `ExecutableEntity execution-instance`, while the supporting Activation states describe the state of the pending activations by `RTEEvent`s or `BswEvent`s.

`ExecutableEntity` execution-instance state	**description**

ExecutableEntity execution-instance is schedulable	This super state describes the life time of the state machine. Only when RTE or the SchM that runs the `ExecutableEntity` execution-instance is started in the corresponding partition, this state machine is active.
`ExecutableEntity` execution-instance Main states	
suspended	The `ExecutableEntity` execution-instance is not started and there is no pending request to start the `ExecutableEntity` execution-instance.
to be started	The `ExecutableEntity` execution-instance is activated but not yet started. Entering the to be started state, usually implies the activation of a task that starts the `ExecutableEntity` execution-instance. The `ExecutableEntity` execution-instance stays in the 'to be started' state, when the task is already running until the gluecode of the task actually calls the function implementing the `ExecutableEntity`.
running	The function, implementing the `ExecutableEntity` code is being executed. The task that contains the `ExecutableEntity` execution-instance is running.
waiting	A task containing the `ExecutableEntity` execution-instance is waiting at a `WaitPoint` within the `ExecutableEntity`.
preempted	A task containing the `ExecutableEntity` execution-instance is preempted from executing the function that implements the `ExecutableEntity`.
started	'started' is the super state of 'running', 'waiting' and 'preempted' between start and termination of the `ExecutableEntity` execution-instance.
`ExecutableEntity` execution-instance Activation states	
not activated	No `RTEEvent` / `BswEvent` requires the activation of the `ExecutableEntity` execution-instance.
debounce activation	One or more `RTEEvent`s with a `startOnEvent` relation to the `ExecutableEntity` execution-instance have occurred [2], but the debounce timer has not yet exceeded the `minimumStartInterval`. The activation will automatically advance to activated, when the debounce timer reaches the `minimumStartInterval`.

[2]Note that, e.g., the same `OperationInvokedEvent` may lead to the activation of different `ExecutableEntity` execution-instances, depending on the client that caused the event.

activated	One or more RTEEvents or BswEvents with a startOnEvent relation to the ExecutableEntity have occurred, and the debounce timer has exceeded the minimumStartInterval. While the activated state is active, the Main state of the ExecutableEntity execution-instance automatically advances from the suspended to the 'to be started' state. For a server runnable where RTE implements a serialization of server calls, an activation counter counts the number of activations. When the ExecutableEntity execution-instance starts, the activation counter will be decremented. When there is still a pending activation, the Activation state will turn to debounce activation and otherwise to no activation.

Table 4.1: States defined for each ExecutableEntity execution-instance.

Note: For tasks, the equivalent state machine does not distinguish between preempted and to be started. They are subsumed as 'ready'.

ExecutableEntity execution-instance transition	**description of event and actions**
initial transition to 'ExecutableEntity execution-instance is schedulable'	RTE or the SchM that runs the ExecutableEntity execution-instance is being started in the corresponding partition.
termination transition from 'ExecutableEntity execution-instance is schedulable'	RTE or the SchM that runs the ExecutableEntity execution-instance gets stopped in the corresponding partition.
transitions to ExecutableEntity execution-instance Main states	
initial transition to suspended	the suspended state is the initial state of the ExecutableEntity execution-instance Main states.
from started to suspended	The ExecutableEntity execution-instance has run to completion.
from suspended to 'to be started'	This transition is automatically executed, while the Activation state is 'activated'.
from 'to be started' to running	The function implementing the ExecutableEntity is called from the context of this execution-instance.
from preempted to running	A task that is preempted from executing the ExecutableEntity execution-instance changes state from preempted to running.
from running to waiting	The runnable enters a WaitPoint.

from waiting to preempted	The task that contains a runnable waiting at a wait point changes from waiting to preempted.
from running to preempted	A task containing the `ExecutableEntity` execution-instance gets preempted from executing the function that implements the `ExecutableEntity`.
transitions to `ExecutableEntity` execution-instance Activation states	
initial transition to 'not activated'	The 'not activated' state is the initial state of the `ExecutableEntity` execution-instance Activation states. The debounce timer is set to the `minimumStartInterval` value, to prevent a delay for the first activation of the `ExecutableEntity` execution-instance.
from activated to 'not activated'	The function implementing the `ExecutableEntity` is called from the context of this execution-instance and no further activations are pending. The debounce timer is reset to 0.
from 'not activated' to 'debounce activation'	The occurrence of an `RTEEvent` or `BswEvent` requires the activation of the `ExecutableEntity` execution-instance. A local activation counter is set to 1. If no `minimumStartInterval` is configured, or the debounce timer has already exceeded the `minimumStartInterval`, the 'debounce activation' state will be omitted and the transition leads directly to the activated state.
from activated to 'debounce activation'	The function implementing the `ExecutableEntity` is called from the context of this execution-instance (start), and another activation is pending (only for `server runnable`). The activation counter is decremented and the debounce timer reset to 0. If no `minimumStartInterval` is configured, the 'debounce activation' state will be omitted and the transition returns directly at the activated state.
from 'debounce activation' to 'debounce activation'	If RTE implements server call serialization for a `server runnable`, and an `OperationInvokedEvent` occurs for the server runnable. The activation counter is incremented (at most to the queue length).
from 'debounce activation' to activated	The debounce timer is expired, `debounce timer > minimumStartInterval`.

from activated to activated	If RTE implements server call serialization for a `server runnable`, and an `OperationInvokedEvent` occurs for the server runnable. The activation counter is incremented (at most to the queue length).

Table 4.2: States defined for each `ExecutableEntity` execution-instance.

[rte_sws_2697]⌈ The activation of `ExecutableEntity execution-instances` shall behave as described by the state machine in Fig. 4.15, Table 4.1, and Table 4.2. ⌋*(RTE00072, RTE00160, RTE00133, RTE00211, RTE00214, RTE00217, RTE00219)*

The RTE will not activate, start or release `ExecutableEntity execution-instances` of a terminated or restarting partition (see rte_sws_7604), or when RTE is stopped in that partition (see rte_sws_2538).

The following examples in Fig. 4.17 and Fig. 4.18 show the different timing situations of the `ExecutableEntity execution-instances` with or without a `minimumStartInterval`. The `minimumStartInterval` can reduce the number of activations by collecting more activating `RTEEvent`s / `BswEvent`s within that interval. No activation will be lost. The activations are just delayed and combined to keep the `minimumStartInterval`. The started state of the `ExecutableEntity execution-instance` Main states and the activated state of the Activation states are shown in the figures. Each flash indicates the occurrence of an `RTEEvent` or `BswEvent`.

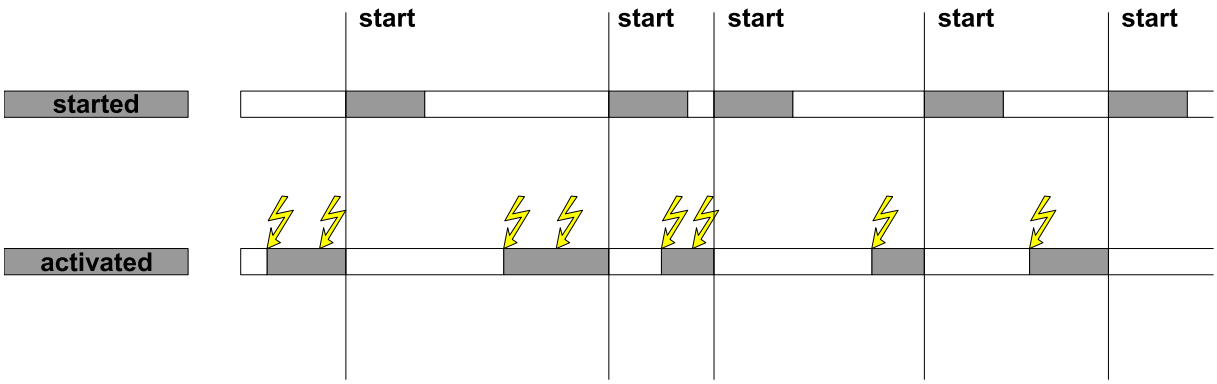


Figure 4.17: Activation of a `ExecutableEntity` execution-instance without `minimumStartInterval`

Figure 4.17 illustrates the activation of an `ExecutableEntity execution-instance` without `minimumStartInterval`. The `execution-instance` can only be activated once (does not apply for `server runnables`). The activation is not queued. The `execution-instance` can already be activated again when it is still started (see Figure 4.15).

With configuration of the RteEventToTaskMapping such activation can even be used for an immediately restart of the ExecutableEntity before other ExecutableEntitys which are mapped subsequently in the task are getting started.

[rte_sws_7061]⌈ When the parameter RteImmediateRestart / RteBswImmediateRestart is TRUE the RTE shall immediately restart the ExecutableEntity after termination if the ExecutableEntity was activated by this RTEEvent / BswEvent while it was already started. ⌋*(RTE00072)*

This can be utilized to spread a long-lasting calculation in several smaller slices with the aim to reduce the maximum blocking time of Tasks in a Cooperative Environment. Typically between each iteration one Schedule Point has to be placed and the number of iteration might depend on operating conditions of the ECU. Further on in a calculation chain the long-lasting calculation shall be completed before consecutive ExecutableEntitys are called.

Example 4.3

Example of RunnableEntity code:

```
1  LongLastingRunnable()
2  {
3      /* the very long calculation */
4      if(!finished)
5      {
6          /* further call is required to complete the calculation*/
7          Rte_IrTrigger_LongLastingCalculation_ProceedCalculation();
8      }
9  }
```

Therefore the ExecutableEntity with a long lasting calculation issues a trigger as long as the calculation is not finished. These trigger activates the ExecutableEntity again. The first activation of the ExecutableEntity might be triggered by another RTEEvent / BswEvent.

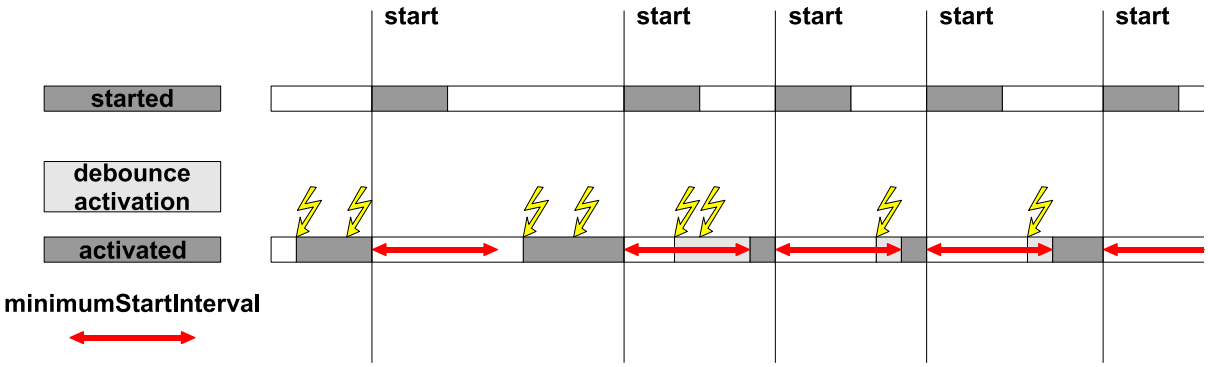


Figure 4.18: Activation of an ExecutableEntity with a minimumStartInterval

Figure 4.18 illustrates the activation of an ExecutableEntity with a minimumStartInterval. (Here no execution-instances have to be distinguished, there

is only one.) The red arrows in this figure indicate the minimumStartInterval after each start of the ExecutableEntity. An RTEEvent or BswEventwithin this minimumStartInterval leads to the debounce activation state. When the minimumStartInterval ends, the debounce activation state changes to the activated state.

When a data received event activates a runnable when it is still running, it might be that the data is already dequeued during the current execution of the runnable. Still, the runnable will be started again. So, it is possible that a runnable that is activated by a data received event finds an empty receive queue.

4.2.3.1 Activation by direct function call

In many cases, ExecutableEntity execution-instances can be implemented by RTE by a direct function call if allowed by the canBeInvokedConcurrently. In these cases, the activation and start of the ExecutableEntity execution-instance collapse to one event. The states 'to be started', 'debounce activation', and 'activated' are passed immediately.

Obviously, debounce activation is not possible (see meta model restriction rte_sws_2733).

There is one ExecutableEntity execution-instance per call point, trigger point, mode switch point, etc.. The state chart simplifies as shown in Figure 4.19.

A triggered ExecutableEntity is activated at least by one ExternalTriggerOccurredEvent or InternalTriggerOccurredEvent. In some cases, the *Trigger Event Communication* or the *Inter Runnable Triggering* is implemented by RTE generator as a direct function call of the triggered ExecutableEntity by the triggering ExecutableEntity.

An OnEntry ExecutableEntity, OnTransition ExecutableEntity, OnExit ExecutableEntity or a mode switch acknowledge ExecutableEntity might be executed in the context of the Rte_Switch API if an asynchronous mode switch procedure is implemented.

A server runnable is exclusively activated by OperationInvokedEvents and implements the server in client server communication. In some cases, the client server communication is implemented by RTE as a direct function call of the server by the client.

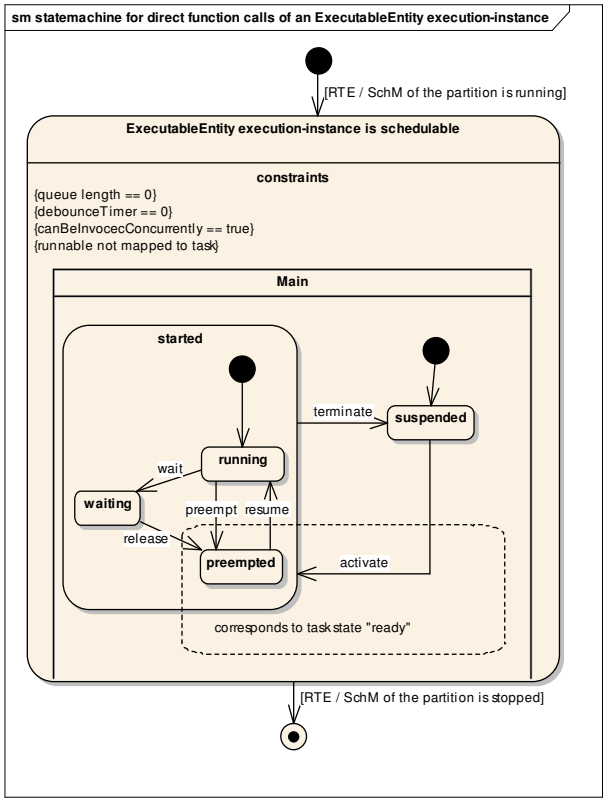


Figure 4.19: State machine of an `ExecutableEntity` execution-instance that is implemented by direct function calls.

4.2.3.2 Activation Offset for `RunnableEntity`s and `BswSchedulableEntity`s

In order to allow optimizations (smooth cpu load, mapping of `RunnableEntity`s and `BswSchedulableEntity`s with different periods in the same task to avoid data sharing, etc.), the RTE has to handle the activation offset information from a task shared reference point only for time trigger `RunnableEntity`s and `BswSchedulableEntity`s. The maximum period of a task can be calculated automatically as the greatest common divisor (GCD) of all runnables period and offset.It is assumed that the runnables worst case execution is less than the GCD. In case of the worst case execution is greater than the GCD, the behavior becomes undefined.

[rte_sws_7000]⌈ The RTE shall respect the configured activation offset of `RunnableEntity`s mapped within one OS task. ⌋*(RTE00161)*

[rte_sws_7520]⌈ The *Basic Software Scheduler* shall respect the configured activation offset of `BswSchedulableEntity`s mapped within one OS task. ⌋*(RTE00212)*

[rte_sws_ext_7521] The `RunnableEntity`s or `BswSchedulableEntity`s worst case execution time shall be less than the GCD of all `BswSchedulableEntity`s and `RunnableEntity`s period and offset in activation offset context for `RunnableEntity`s and `BswSchedulableEntity`s.

Note: The following examples are showing `RunnableEntity`s only. Nevertheless it is applicable for `BswSchedulableEntity`s or a mixture of `RunnableEntity`s and `BswSchedulableEntity`s as well.

Example 1:
This example describes 3 runnables mapped in one task with an activation offset defined for each runnables.

Runnable	Period	Activation Offset
R1	100ms	20ms
R2	100ms	60ms
R3	100ms	100ms

Table 4.3: Runnables timings

The runnables R1, R2 and R3 are mapped in the task T1 at 20 ms which is the GCD of all runnables period and activation offset.

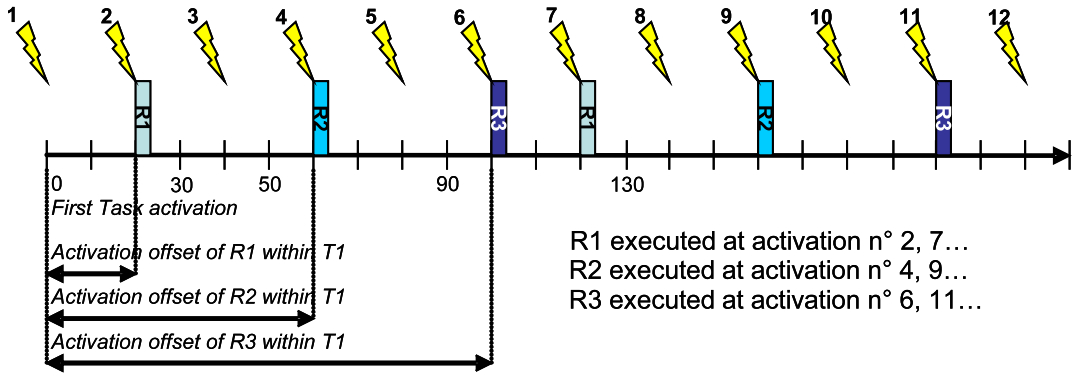


Figure 4.20: Example of activation offset for runnables

Example 2:
This example describes 4 runnables mapped in one task with an activation offset and position in task defined for each runnables.

Runnable	Period	Position in task	Activation Offset
R1	50ms	1	0ms
R2	100ms	2	0ms
R3	100ms	3	70ms
R4	50ms	4	20ms

Table 4.4: Runnables timings with position in task

The runnables R1, R2, R3 and R4 are mapped in the task T1 at 10 ms which is the GCD of all runnables period and activation offset.

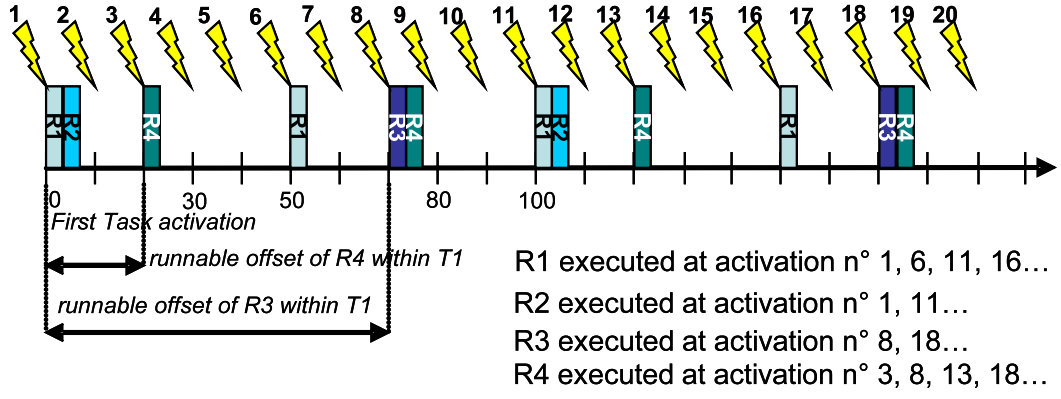


Figure 4.21: Example of activation offset for runnables with position in task

4.2.4 Interrupt decoupling and notifications

4.2.4.1 Basic notification principles

Several BSW modules exist which contain functionality which is not directly activated, triggered or called by AUTOSAR software-components but by other circumstances, like digital input port level changes, complex driver actions, CAN signal reception, etc. In most cases interrupts are a result of those circumstances. For a definition of interrupts, see the VFB [1].

Several of these BSW functionalities create situations, signalled by an interrupt, when AUTOSAR SW-Cs have to be involved. To inform AUTOSAR software components of those situations, runnables in AUTOSAR software components are activated by notifications. So interrupts that occur in the basic software have to be transformed into notifications of the AUTOSAR software components. Such a transformation has to take place at RTE level **at the latest**! Which interrupt is connected to which notification is decided either during system configuration/generation time or as part of the design of Complex Device Drivers or the Microcontroller Abstraction Layer.

This means that runnables in AUTOSAR SW-Cs have to be activated or "waiting" cat2 runnables in extended tasks have to be set to "ready to run" again. In addition some event specific data may have to be passed.

There are two different mechanisms to implement these notifications, depending on the kind of BSW interfaces.

1. **BSW with Standardized interface**. Used with COM and OS.
   Basic-SW modules with Standardized interfaces cannot create RTEEvents. So another mechanism must be chosen: "**callbacks**"
   The typical callback realization in a C/C++ environment is a function call.
2. **BSW with AUTOSAR interface**: Used in all the other BSW modules.
   Basic-SW modules with AUTOSAR-Interfaces have their interface specified in an AUTOSAR BSW description XML file which contains signal specifications according to the AUTOSAR specification. The BSW modules can employ RTE API calls like Rte_Send – see *5.6.5*). `RTEEvent`s may be connected with the RTE API calls, so realizing AUTOSAR SW-C activation.

Note that an AUTOSAR software component can send a notification to another AUTOSAR software component or a BSW module only via an AUTOSAR interface.

4.2.4.2 Interrupts

The AUTOSAR concept as stated in the VFB specification [1] does not allow AUTOSAR software components to run in interrupt context. Only the Microcontroller Abstraction Layer, Complex Device Drivers and the OS are allowed to directly interact with interrupts and implement interrupt service routines (see Requirement BSW164). This ensures hardware independence and determinism.

If AUTOSAR software components were allowed to run in interrupt context, one AUTOSAR software component could block the entire system schedule for an unacceptably long period of time. But the main reason is that AUTOSAR software components are supposed to be independent of the underlying hardware so that exchangeability between ECUs can be ensured. The schedule of an ECU is more predictable and better testable if the timing effects of interrupts are restricted to the basic software of that ECU.

Furthermore, AUTOSAR software components are not allowed to explicitly block interrupts as a means to ensure data consistency. They have to use RTE functions for this purpose instead, see Section 4.2.5.

4.2.4.3 Decoupling interrupts on RTE level

Runnables in AUTOSAR SW-Cs may be running as a consequence of an interrupt but **not** in interrupt context, which means not within an interrupt service routine! Between the interrupt service routine and an AUTOSAR SW-C activation there must always be a decoupling instance. AUTOSAR SW-C runnables are only executed in the context of tasks.

The decoupling instance is latest in the RTE. For the RTE there are several options to realize the decoupling of interrupts. Which option is the best depends on the configuration and implementation of the RTE, so only examples are given here.

Example 1:

Situation:

- An interrupt routine calls an RTE callback function

Intention:

- Start a runnable

RTE job:

- RTE starts a task containing the runnable activation code by using the "ActivateTask()" OS service call.
- Other more sophisticated solutions are possible, e.g. if the task containing the runnable is activated periodically.

Example 2:

Situation:

- An interrupt routine calls an RTE callback function

Intention:

- Make a runnable wake up from a wait point

RTE job:

- RTE sets an OS event

These scenarios described in the examples above not only hold for RTE callback functions but for other RTE API functions as well.

[rte_sws_3600]⌈ The RTE shall prevent runnable entities of AUTOSAR software-components to run in interrupt context. ⌋*(RTE00099, BSW00326)*

4.2.4.4 RTE and interrupt categories

Since category 1 interrupts are not under OS control the RTE has absolutely no possibility to influence their execution behavior. So no category 1 interrupt is allowed to reach RTE. This is different for interrupt of category 2.

[rte_sws_3594]⌈ The RTE Generator shall reject the configuration if a `SwcBswRunnableMapping` associates a `BswInterruptEntity` with a `RunnableEntity` and the attribute `interruptCategory` of the `BswInterruptEntity` is equal to `cat 1`. ⌋*(RTE00099, BSW00326, RTE00018)*

[rte_sws_ext_7816] Category 1 interrupts shall not access the RTE.

4.2.4.5 RTE and Basic Software Scheduler and `BswExecutionContext`

The RTE and *Basic Software Scheduler* do support the invocation `triggered ExecutableEntity` via direct function call in some special cases. Nevertheless it shall be prevented that an `ExecutableEntity` from a particular execution context calls a `triggered ExecutableEntity` wich requires an execution context with more permissions. The table 4.5 lists the supported combinations.

caller's `BswExecutionContext`[3]	callee's `BswExecutionContext`[3]				
	task	**interruptCat2**	**interruptCat1**	**hook**	**unspecified**
task	supported	supported	supported		supported
interruptCat2		supported	supported		supported
interruptCat1			supported		supported
hook					
unspecified	supported				supported

Table 4.5: Possible invocation of `ExecutableEntity`s by direct function call dependent from `BswExecutionContext`

For example (fourth column), the invocation of an `ExecutableEntity` with an `interruptCat1 BswExecutionContext` can be implemented with a direct function

[3]The execution context of a `RunnableEntity` is considered as `task`

call if the BswExecutionContext of the caller BswModuleEntry is set to task, interruptCat2, or interruptCat1.

This applies to the invocation of a triggered ExecutableEntity by the SchM_Trigger, SchM_ActMain or Rte_Trigger APIs, or to the invocation of an OnEntry ExecutableEntity, OnTransition ExecutableEntity, OnExit ExecutableEntity or mode switch acknowledge ExecutableEntity by the SchM_Switch or Rte_Switch APIs.

4.2.4.5.1 Interrupt decoupling for COM

COM callbacks are used to inform the RTE about something that happened independently of any RTE action. This is often interrupt driven, e.g. when a data item has been received from another ECU or when a S/R transmission is completed.
It is the RTE's job e.g. to create RTEEvents from the interrupt.

[rte_sws_3530]⌈ The RTE shall provide callback functions to allow COM to signal COM events to the RTE. ⌋*(RTE00072, RTE00099, BSW00326)*

[rte_sws_3531]⌈ The RTE shall support runnable activation by COM callbacks. ⌋*(RTE00072, RTE00099, BSW00326)*

[rte_sws_3532]⌈ The RTE shall support category 2 runnables to wake up from a wait point as a result of COM callbacks. ⌋*(RTE00072, RTE00099, BSW00326)*

See RTE callback API in chapter 5.9.

4.2.5 Data Consistency

4.2.5.1 General

Concurrent accesses to shared data memory can cause data inconsistencies. In general this must be taken into account when several code entities accessing the same data memory are running in different contexts - in other words when systems using parallel (multicore) or concurrent (singlecore) execution of code are designed. More general: Whenever task context-switches occur and data is shared between tasks, data consistency is an issue.

AUTOSAR systems use operating systems according to the AUTOSAR-OS specification which is derived from the OSEK-OS specification. The Autosar OS specification defines a priority based scheduling to allow event driven systems. This means that tasks with higher priority levels are able to interrupt (preempt) tasks with lower priority level.

The "lost update" example in Figure 4.22 illustrates the problem for concurrent read-modify-write accesses:

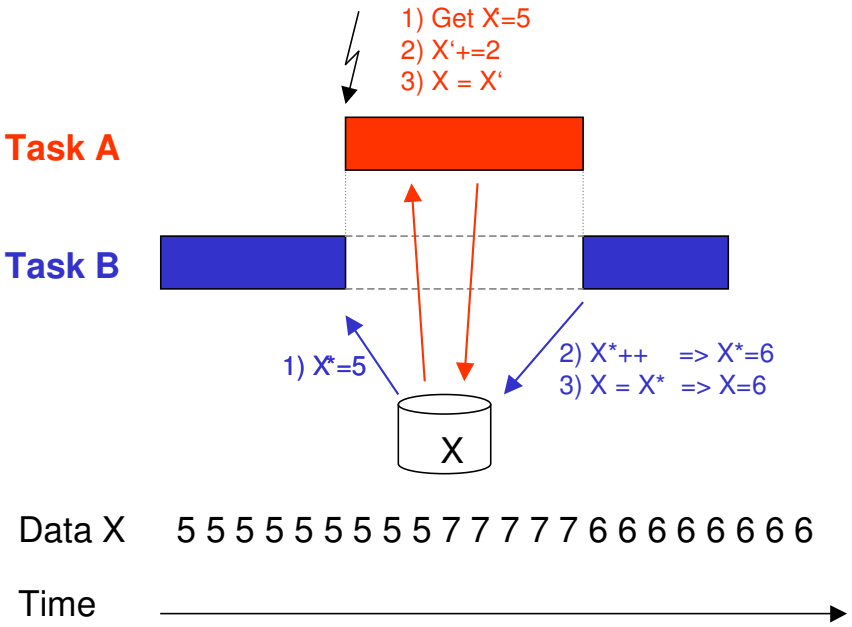


Figure 4.22: Data inconsistency example - lost update

There are two tasks. Task A has higher priority than task B. A increments the commonly accessed counter X by 2, B increments X by 1. So in both tasks there is a read (step1) – modify (step2) – write (step3) sequence. If there are no atomic accesses (fully completed read-modify-write accesses without interruption) the following can happen:

1. Assume X=5.
2. B makes read (step1) access to X and stores value 5 in an intermediate store (e.g. on stack or in a CPU register).
3. B cannot continue because it is preempted by A.
4. A does its read (step1) – modify (step2) – write (step3) sequence, which means that A reads the actual value of X, which is 5, increments it by 2 and writes the new value for X, which is 7. (X=5+2)
5. A is suspended again.
6. B continues where it has been preempted: with its modify (step2) and write (step3) job. This means that it takes the value 5 form its internal store, increments it by one to 6 and writes the value 6 to X (X=5+1).
7. B is suspended again.

The correct result after both Tasks A and B are completed should be X=8, but the update of X performed by task A has been lost.

4.2.5.2 Communication Patterns

In AUTOSAR systems the RTE has to take care that a lot of the communication is not corrupted by data consistency problems. RTE Generator has to apply suitable means if required.

The following communication mechanisms can be distinguished:

- Communication within one atomic AUTOSAR SW-C:
  Communication between Runnables of one atomic AUTOSAR SW-C running in different task contexts where communication between these Runnables takes place via commonly accessed data. If the need to support data consistency by the RTE exists, it must be specified by using the concepts of "ExclusiveAreas" or "InterRunnableVariables" only.
- Intra-partition communication between AUTOSAR SW-Cs:
  Sender/Receiver (S/R) communication between Runnables of different AUTOSAR SW-Cs using *implicit* or *explicit* data exchange can be realized by the RTE through commonly accessed RAM memory areas. Data consistency in Client/Server (C/S) communication can be put down to the same concepts as S/R communication. Data access collisions must be avoided. The RTE is responsible for guaranteeing data consistency.
- Inter-Partition communication
  The RTE has to guarantee data consistency. The different possibilities provided to the RTE for the communication between partitions are discussed in section 4.3.4.
- Intra-ECU communication between AUTOSAR SW-Cs and BSW modules with AUTOSAR interfaces:
  This is a special case of the above two.
- Inter ECU communication
  COM has to guarantee data consistency for communication between ECUs on complete path between the COM modules of different ECUs. The RTE on each ECU has to guarantee that no data inconsistency might occur when it invokes COM send respectively receive calls supplying respectively receiving data items which are concurrently accessed by application via RTE API call, especially when queueing is used since the queues are provided by the RTE and not by COM.

[rte_sws_3514]⌈ The RTE has to guarantee data consistency for communication via AUTOSAR interfaces. ⌋*(RTE00032)*

4.2.5.3 Concepts

In the AUTOSAR SW-C Template [2] chapter "Interaction between runnables within one component", the concepts of

1. ExclusiveAreas *(see section 4.2.5.5 below)*

2. InterRunnableVariables *(see section 4.2.5.6 below)*

are introduced to allow the user (SW-Designer) to specify where the RTE shall guarantee data consistency for AUTOSAR SW-C internal communication and execution circumstances. This is discussed in more detail in next sections.

Additionally exclusive areas are also available for *Basic Software Modules* to protect access to module internal data. See [9]. The exclusive areas for *Basic Software Modules* are handled by the *Basic Software Scheduler*.

The AUTOSAR SW-C template specification [2] also states that AUTOSAR SW-Cs may define `PerInstanceMemory` or `arTypedPerInstanceMemory`, allowing reservation of static (permanent) need of global RAM for the SW-C. Nothing is specified about the way Runnables might access this memory. RTE only provides a reference to this memory *(see section 5.6)* but doesn't guarantee data consistency for it.

The implementer of an AUTOSAR SW-C has to take care by himself that accesses to RAM reserved as PerInstanceMemory out of Runnables running in different task contexts don't cause data inconsistencies. On the other hand this provides more freedom in using the memory.

4.2.5.4 Mechanisms to guarantee data consistency

ExclusiveAreas and InterRunnableVariables are only mentioned in association with AUTOSAR SW-C internal communication. Nevertheless the data consistency mechanisms behind can be applied to communication between AUTOSAR SW-Cs or between AUTOSAR SW-Cs and BSW modules too. Everywhere where the RTE has to guarantee data consistency.

The data consistency guaranteeing mechanisms listed here are derived from AUTOSAR SW-C Template and from further discussions. There might be more (see section 4.3.4 for the mechanisms involved for `inter-partition` communication).
The RTE has the responsibility to apply such mechanisms if required. The details how to apply the mechanisms are left open to the RTE supplier.

Mechanisms:

- ***Sequential scheduling strategy***
  The activation code of Runnables is sequentially placed in one task so that no interference between them is possible because one Runnable is only activated after the termination of the other. Data consistency is guaranteed.
- ***Interrupt blocking strategy***
  Interrupt blocking can be an appropriate means if collision avoidance is required for a very short amount of time. This might be done by disabling respectively suspending all interrupts, Os interrupts only or - if hardware supports it - only of some interrupt levels. In general this mechanism must be applied with care

because it might influence SW in tasks with higher priority too and the timing of the complete system.

- ***Usage of OS resources***
  Usage of OS resources. Advantage in comparison to Interrupt blocking strategy is that less SW parts with higher priority are blocked. Disadvantage is that implementation might consume more resources (code, runtime) due to the more sophisticated mechanism.

- ***Task blocking strategy***
  Mutual task preemption is prohibited. This might be reached e.g. by assigning same priorities to affected tasks, by assigning same internal OS resource to affected tasks or by configuring the tasks to be non-preemptive.

- ***Cooperative Runnable placement strategy***
  The principle is that tasks containing Runnables to be protected by "Cooperative Runnable placement strategy" are not allowed to preempt other tasks also containing Runnables to be protected by "Cooperative Runnable placement strategy" when one of the Runnables to protect is active - but are allowed between Runnable executions. The RTE's job is to create appropriate task bodies and use OS services or other mechanisms to achieve the required behavior.

  To point out the difference to "Task blocking strategy":
  In "Task blocking strategy" no task containing Runnables with access to the ExclusiveArea at all is allowed to preempt another task containing Runnables with access to same ExclusiveArea. In "Cooperative Runnable placement strategy" this task blocking mechanism is limited to tasks defined to be within same cooperative context.

  Example to explain the cooperative mechanism:

  - Runnables R2 and R3a are marked to be protected by cooperative mechanism.
  - Runnables R1, R3b and R4 have no cooperative marking.
  - R1 is activated in Task T1, R2 is activated in Task T2, R3a is activated in Task T3a, R3b is activated in Task T3b, R4 is activated in Task T4.
  - Task priorities are: T4 $>$ T3a $>$ T2 $>$ T1, T3b has same priority as T3a

  This setup results in this behavior:

  - T4 can always preempt all other tasks (Higher prio than all others).
  - T3b can preempt T2 (higher prio of T3b, no cooperative restriction)
  - T3a cannot preempt T2 (Higher prio of T3a but same cooperative context). So data access of Runnable R2 to common data cannot interfere with data access by Runnable R3a. Nevertheless if both tasks T3a and T2 are ready to run, it's guaranteed that T3a is running first.

  - T1 can never preempt one of the other tasks because of lowest assigned prio.

- ***Copy strategy***
  Idea: The RTE creates copies of data items so that concurrent accesses in different task contexts cannot collide because some of the accesses are redirected to the copies.

  How it can work:

  - Application for ***read*** *conflicts*:
    For all readers with lower priority than the writer a *read copy* is provided.

    Example:
    There exist Runnable R1, Runnable R2, data item X and a copy data item X*. When Runnable R1 is running in higher priority task context than R2, and R1 is the only one writing X and R2 is reading X it is possible to guarantee data consistency by making a copy of data item X to variable X* before activation of R2 and redirecting write access from X to X* or the read access from X to X* for R2.

  - Application for ***write*** *conflicts*:
    If one or more data item receiver with a higher priority than the sender exist, a *write copy* for the sender is provided.

    Example:
    There exist Runnable R1, Runnable R2, data item X and copy data item X*. When Runnable R1 (running in lower priority task context than R2) is writing X and R2 is reading X, it is possible to guarantee data consistency by making a copy of data item X to data item X* before activation of R1 together with redirecting the write access from X to X* for R1 or the read access from X to X* for R2.

  Usage of this copy mechanism may make sense if one or more of the following conditions hold:

  - This copy mechanism can handle those cases when only one instance does the data write access.
  - R2 is accessing X several times.
  - More than one Runnable R2 has read (resp. write) access to X.
  - To save runtime is more important than to save code and RAM.
  - Additional RAM requirements to hold the copies is acceptable.

  Further issues to be taken into account:

– AUTOSAR SW-Cs provided as source code and AUTOSAR SW-Cs provided as object code may or have to be handled in different ways. The redirecting mechanism for source code could use macros for C and C++ very efficiently whereas object-code AUTOSAR SW-Cs most likely are forced to use references.

Note that the copy strategy is used to guarantee data consistency for implicit sender-receiver communication (`VariableAccess`es in the `dataReadAccess` or `dataWriteAccess` role) and for AUTOSAR SW-C internal communication using InterRunnableVariables with implicit behavior.

4.2.5.5 Exclusive Areas

The concept of `ExclusiveArea` is more a working model. It's not a concrete implementation approach, although concrete possible mechanisms are listed in AUTOSAR SW-C template specification [2].

Focus of the `ExclusiveArea` concept is to block potential concurrent accesses to get data consistency. `ExclusiveArea`s implement critical section

*ExclusiveArea*s are associated with `RunnableEntity`s. The RTE is forced to guarantee data consistency when the `RunnableEntity` runs in an `ExclusiveArea`. A `RunnableEntity` can run inside one or several `ExclusiveArea`s completely or can enter one or several `ExclusiveArea`s during their execution for one or several times .

- If an AUTOSAR SW-C requests the RTE to look for data consistency for it's internally used data (for a part of it or the complete one) using the `ExclusiveArea` concept, the SW designer can use the API calls "Rte_Enter()" in 5.6.27 and "Rte_Exit()" in 5.6.28 to specify where he wants to have the protection by RTE applied.
  "Rte_Enter()" defines the begin and "Rte_Exit()" defines the end of the code sequence containing data accesses the RTE has to guarantee data consistency for.
- If the SW designer wants to have the mutual exclusion for complete `RunnableEntity`s he can specify this by using the *ExclusiveArea* in the role "`runsInsideExclusiveArea`" in the AUTOSAR SW-C description.

In principle the `ExclusiveArea` concept can handle the access to single data items as well as the access to several data items realized by a group of instructions. It also doesn't matter if one Runnable is completely running in an `ExclusiveArea` and

another Runnable only temporarily enters the same ExclusiveArea. The RTE has to guarantee data consistency.

[rte_sws_3500]⌈ The RTE has to guarantee data consistency for arbitrary accesses to data items accessed by Runnables marked with the same ExclusiveArea. ⌋*(RTE00032, RTE00046)*

[rte_sws_3515]⌈ RTE has to provide an API enabling the SW-Cs to access and leave ExclusiveAreas. ⌋*(RTE00046)*

If Runnables accessing same ExclusiveArea are assigned to be executing in different task contexts, the RTE can apply suitable mechanisms, e.g. task blocking, to guarantee data consistency for data accesses in the common ExclusiveArea. However, specials attributes can be set that require certain data consistency mechanisms in which case the RTE generator is forced to apply the selected mechanism.

The *Basic Software Scheduler* provides ExclusiveAreas for the *Basic Software Module*s. *Basic Software Module*s have to use the API calls SchM_Enter()" in 6.5.1 and SchM_Exit()" in 6.5.2 to specify where the protection by Basic Software Scheduler has to be applied.

[rte_sws_7522]⌈ The *Basic Software Scheduler* has to guarantee data consistency for arbitrary accesses to data items accessed by BswModuleEntitys marked with the same ExclusiveArea. ⌋*(RTE00222, RTE00046)*

[rte_sws_7523]⌈ *Basic Software Scheduler* has to provide an API enabling the *Basic Software Module* to access and leave ExclusiveAreas. ⌋*(RTE00222, RTE00046)*

It is not supported, that a BswModuleEntity which is not a BswSchedulableEntity uses an ExclusiveArea in the role runsInsideExclusiveArea This is not possible, because such BswSchedulableEntity might be called directly by other *Basic Software Modules* and therefore the *Basic Software Scheduler* is not able to enter and exit the ExclusiveArea automatically.

[rte_sws_7524]⌈ The RTE generator shall reject a configuration where a BswModuleEntity which is not a BswSchedulableEntity uses an ExclusiveArea in the role runsInsideExclusiveArea. ⌋*(RTE00222, RTE00046, RTE00018)*

4.2.5.5.1 Assignment of data consistency mechanisms

The data consistency mechanism that has to be applied to anExclusiveArea might be domain, ECU or even project specific. The decision which mechanism has to be applied by RTE / *Basic Software Scheduler* is taken during ECU integration by setting the ExclusiveArea configuration parameter ExclusiveAreaImplMechanism. This parameter is an input for RTE generator.

As stated in section 4.2.5.4 there might be more mechanisms to realize *ExclusiveArea*s as mentioned in this specification. So RTE implementations might provide other mechanisms in plus by a vendor specific solutions. This allows further optimizations.

Actually following values for configuration parameter ExclusiveAreaImplMechanism must be supported:

- AllInterruptBlocking
  This value requests enabling and disabling of all Interrupts and is based on the *Interrupt blocking strategy*.
- OsInterruptBlocking
  This value requests enabling and disabling of Os Interrupts and is based on the *Interrupt blocking strategy*.
- OSResources
  This value requests to apply the *Usage of OS resources* mechanism.
- CooperativeRunnablePlacement
  This value requires to apply the *Cooperative Runnable Placement Strategy*.

The strategies / mechanisms are described in general in section 4.2.5.4.

[rte_sws_3504]⌈ If the configuration parameter ExclusiveAreaImplMechanism of an ExclusiveArea is set to value ALL_INTERRUPT_BLOCKING the RTE generator shall use the mechanism of *Interrupt blocking* (blocking all interrupts) to guarantee data consistency if data inconsistency could occur. ⌋*(RTE00032)*

[rte_sws_5164]⌈ If the configuration parameter ExclusiveAreaImplMechanism of an ExclusiveArea is set to value OS_INTERRUPT_BLOCKING the RTE generator shall use the mechanism of *Interrupt blocking* (blocking Os interrupts only) to guarantee data consistency if data inconsistency could occur. ⌋*(RTE00032)*

[rte_sws_3595]⌈ If the configuration parameter ExclusiveAreaImplMechanism of an ExclusiveArea is set to value OS_RESOURCE the RTE generator shall use OS resources to guarantee data consistency if data inconsistency could occur. ⌋*(RTE00032)*

The requirements above have the limitation "if data inconsistency could occur" because it makes no sense to apply a data consistency mechanism if no potential data inconsistency can occur. This can be relevant if e.g. the "Sequential scheduling strategy" (described in section 4.2.5.4) still has solved the item by the ECU integrator defining an appropriate runnable-to-task mapping.

[rte_sws_3503]⌈ If the configuration parameter ExclusiveAreaImplMechanism of an ExclusiveArea is set to value COOPERATIVE_RUNNABLE_PLACEMENT the RTE generator shall generate code according the *Cooperative Runnable Placement Strategy* to guarantee data consistency. ⌋*(RTE00032)*

Since the decision to select the *Cooperative Runnable Placement Strategy* to prohibit data access conflicts affects the behavior of several tasks and potentially many ExclusiveAreas the RTE generator is not allowed to override the decision.

In a SWC code, it is not allowed to use WaitPoints inside an ExclusiveArea: The RTE generator might use OSEK services to implement ExclusiveAreas and

waiting for an OS event is not allowed when an OSEK resource has been taken for example. For RunnableEntityEntersExclusiveArea, the RTE generator cannot check if WaitPoints are inside an `ExclusiveArea`. Therefore, it is the responsibility of the SWC Code writer to ensure that no `WaitPoint`s are used inside an exclusive area. But for `RunnableEntity`s running inside a `ExclusiveArea`, the RTE generator is able to do the following check.

[rte_sws_7005]⌈ The RTE generator shall reject a configuration with a `WaitPoint` applied to a `RunnableEntity` which is using the `ExclusiveArea` in the role `runsInsideExclusiveArea` ⌋*(RTE00032, RTE00018)*

4.2.5.6 InterRunnableVariables

A non-composite AUTOSAR SW-C can reserve InterRunnableVariables which can be accessed by the Runnables of this one AUTOSAR SW-C (also see section 4.3.3.1). Read and write accesses are possible. There is a separate set of those variables per AUTOSAR SW-C instance.

Again the RTE has to guarantee data consistency. Appropriate means will depend on Runnable placement decisions which are taken during ECU configuration.

[rte_sws_3516]⌈ The RTE has to guarantee data consistency for communication between Runnables of one AUTOSAR software-component instance using the same InterRunnableVariable. ⌋*(RTE00142, RTE00032)*

Next the two kinds of InterRunnableVariables are treated:

1. InterRunnableVariables with **implicit** behavior

   (`implicitInterRunnableVariable`)

2. InterRunnableVariables with **explicit** behavior

   (`explicitInterRunnableVariable`)

4.2.5.6.1 InterRunnableVariables with implicit behavior

In applications with very high SW-C communication needs and much real time constraints (like in powertrain domain) the usage of a copy mechanism to get data consistency might be a good choice because during `RunnableEntity` execution no data consistency overhead in form of concurrent access blocking code and runtime during its execution exists - independent of the number of data item accesses.
Costs are code overhead in the `RunnableEntity` prologue and epilogue which is often be minimal compared to other solutions. Additional RAM need for the copies comes in plus.

When *InterRunnableVariables with implicit behavior* are used the RTE is required to make the data available to the Runnable using the `semantics of a copy` operation but is not necessarily required to use a unique copy for each `RunnableEntity`.

Focus of *InterRunnableVariable with implicit behavior* is to avoid concurrent accesses by redirecting second, third, .. accesses to data item copies.

[rte_sws_3517]⌈ The RTE shall guarantee data consistency for *InterRunnableVariables with implicit behavior* by avoiding concurrent accesses to data items specified by `implicitInterRunnableVariable` using one or more copies and redirecting accesses to the copies.
⌋*(RTE00142, RTE00032)*

Compared with Sender/Receiver communication

- Like with `VariableAccesse`s in the `dataReadAccess` and `dataWriteAccess` roles, the Runnable IN data is stable during Runnable execution, which means that during an Runnable execution several read accesses to an `implicitInterRunnableVariable` always deliver the same data item value.
- Like with `VariableAccesse`s in the `dataReadAccess` and `dataWriteAccess` roles, the Runnable OUT data is forwarded to other Runnables not before Runnable execution has terminated, which means that during an Runnable execution write accesses to `implicitInterRunnableVariable` are not visible to other Runnables.

This behavior requires that Runnable execution terminates.

[rte_sws_3582]⌈ The value of several read accesses to `implicitInterRunnableVariable` during a `RunnableEntity` execution shall only change for write accesses performed within this `RunnableEntity` to the `implicitInterRunnableVariable` ⌋*(RTE00142)*

[rte_sws_3583]⌈ Several write accesses to `implicitInterRunnableVariable` during a `RunnableEntity` execution shall result in only one update of the `implicitInterRunnableVariable` content visible to other `RunnableEntity`s with the last written value.
⌋*(RTE00142)*

[rte_sws_3584]⌈ The update of `implicitInterRunnableVariable` done during a `RunnableEntity` execution shall be made available to other `RunnableEntity`s after the `RunnableEntity` execution has terminated.
⌋*(RTE00142)*

[rte_sws_7022]⌈ If a `RunnableEntity` has both read and write access to an `implicitInterRunnableVariable` the result of the write access shall be immediately visible to subsequent read accesses from within the same runnable entity.
⌋*(RTE00142)*

The usage of `implicitInterRunnableVariable`s is permitted for all categories of runnable entities. For runnable entities of category 2, the behavior is guaranteed only

if it has a finite execution time. A category 2 runnable that runs forever will not have its data updated.

For API of `implicitInterRunnableVariable` see sections 5.6.23 and 5.6.24.

For more details how this mechanism could work see "Copy strategy" in section 4.2.5.4.

4.2.5.6.2 InterRunnableVariables with explicit behavior

In many applications saving RAM is more important than saving runtime. Also some application require to have access to the newest data item value without any delay, even several times during execution of a Runnable.

Both requirements can be fulfilled when RTE supports data consistency by blocking second/third/.. concurrent accesses to a signal buffer if data consistency is jeopardized. (Most likely RTE has nothing to do if SW is running on a 16bit machine and making an access to an 16bit value when a 16bit data bus is present.)

Focus of *InterRunnableVariables with explicit behavior* is to block potential concurrent accesses to get data consistency.

The mechanism behind is the same as in the `ExclusiveArea` concept (see section 4.2.5.5). But although ExclusiveAreas can handle single data item accesses too, their API is made to make the RTE to apply data consistency means for a group of instructions accessing several data items as well. So when using an ExclusiveArea to protect accesses to one single common used data item each time two RTE API calls grouped around are needed. This is very inconvenient and might lead to faults if the calls grouped around might be forgotten.
The solution is to support *InterRunnableVariables with explicit behavior*.

[rte_sws_3519]⌈ The RTE shall guarantee data consistency for *InterRunnableVariables with explicit behavior* by blocking concurrent accesses to data items specified by `explicitInterRunnableVariable`.
⌋*(RTE00142, RTE00032)*

The RTE generator is not free to select on it's own if implicit or explicit behavior shall be applied. Behavior must be known at AUTOSAR SW-C design time because in case of *InterRunnableVariables with implicit behavior* the AUTOSAR SW-C designer might rely on the fact that several read accesses always deliver same data item value.

[rte_sws_3580]⌈ The RTE shall supply different APIs for *InterRunnableVariables with implicit* behavior and *InterRunnableVariables with explicit* behavior.
⌋*(RTE00142)*

For API of *InterRunnableVariables with explicit behavior* see sections 5.6.25 and 5.6.26.

4.2.6 Multiple trigger of Runnable Entities and Basic Software Schedulable Entities

Concurrent activation

The AUTOSAR SW-C template specification [2] states that runnable entities (further called "runnables") might be invoked concurrently several times if the Runnables attribute `canBeInvokedConcurrently` is set. It's then in the responsibility of the AUTOSAR SW-C designer that no data might be corrupted when the Runnable is activated several times in parallel.

If a SW-C has multiple instances, they have distinct runnables. Two runnables that use the same `RunnableEntity` description of the same `SwcInternalBehavior` description but are instantiated with two different SW-C instances are treated as two distinct runnables in the following. This kind of concurrency is always allowed between SW-Cs, even if the runnables have their `canBeInvokedConcurrently` attribute set to false.

[rte_sws_3523]⌈ The RTE shall support concurrent activation of the same instance of a runnable entity if the associative attribute `canBeInvokedConcurrently` is set to `TRUE`. This includes concurrent activation in several tasks. If the attribute is not set resp. set to `FALSE`, concurrent activation of the runnable entity is forbidden. (see requirement rte_sws_5083) ⌋*(RTE00072, RTE00133)*

The *Basic Software Module Description Template* [9] specifies the possible concurrent activation of `BswModuleEntity`s by the attribute `isReentrant`.

[rte_sws_7525]⌈ The *Basic Software Scheduler* shall support concurrent activation of the same instance of a `BswSchedulableEntity` if the attribute `isReentrant` of the referenced `BswModuleEntry` in the role `implementedEntry` is set to `true`. This includes concurrent activation in several tasks. If the attribute is set to `false` concurrent activation of the `BswSchedulableEntity` is forbidden. (see requirement rte_sws_7588) ⌋*()*

Concurrent activation of the same instance of a `ExecutableEntity` results in multiple `ExecutableEntity execution-instances`. One for each context of activation.

Activation by several RTEEvents and BswEvents

Nevertheless a Runnable whose attribute `canBeInvokedConcurrently` is NOT set might be still activated by several `RTEEvent`s if activation configuration guarantees that concurrent activation can never occur and the `minimumStartInterval` condition is kept. This includes activation in different tasks. In this case, the runnable is still considered to have only one `ExecutableEntity execution-instances`. A standard use case is the activation of same instance of a runnable in different modes.

[rte_sws_3520]⌈ The RTE shall support activation of same instance of a runnable entity by multiple `RTEEvent`s. ⌋*(RTE00072)*

RTEEvents are triggering runnable activation and may supply 0..several role parameters, see *section 5.7.3*. Role parameters are not visible in the runnables signature - except in those triggered by an OperationInvokedEvent. With the exception of the RTEEvent OperationInvokedEvent all role parameters can be accessed by user with implicit or explicit Receiver API.

[rte_sws_3524]⌈ The RTE shall support activation of same instance of a runnable entity by RTEEvents of different kinds. ⌋*(RTE00072)*

The RTE does NOT support a runnable entity triggered by an RTEEvent OperationInvokedEvent to be triggered by any other RTEEvent except for other OperationInvokedEvents of compatible operations. This limitation is stated in appendix in *section A.2* (rte_sws_3526).

The similar configuration as mentioned for the RunnableEntitys might be used for BswSchedulableEntitys. Therefore even a BswSchedulableEntity whose referenced BswModuleEntry in the role implementedEntry has its isReentrant attribute set to false can be activated by several BswEvents.

[rte_sws_7526]⌈ The *Basic Software Scheduler* shall support activation of same instance of a BswSchedulableEntity by multiple BswEvents. ⌋*()*

[rte_sws_7527]⌈ The *Basic Software Scheduler* shall support activation of same instance of a BswSchedulableEntity by BswEvents of different kinds. ⌋*()*

4.2.7 Implementation of Parameter and Data elements

4.2.7.1 General

A SWC communicates with other SWCs through ports. A port is characterized by a PortInterface and there are several kinds of PortInterfaces. In this section, we focus on the ParameterInterface, the SenderReceiverInterface, and the NvDataInterface. These three kinds of PortInterfaces aggregate some specific interface elements. For example, a ParameterInterface aggregates 0..* ParameterDataPrototypes.

4.2.7.2 Compatibility rules

A receiver port can only be connected to a compatible provider port. The compatibility rules are explained in the AUTOSAR Software Component Template [2]. The compatibility mainly depends on the attribute swImplPolicy attached to the element of the interface. The table 4.6 below gives an overview of compatibility rules.

For examples, a Require Port that expects a fixed parameter - i.e produced by a macro #define - can only be connected to a Port that provides a fixed Parameter. This is because this fixed data may be used in a compilation directive like #IF and only macro #define (fixed data) can be compiled in this case. On the other hand, this provided fixed

Provide Port			Require Port					
Port Interface			Prm			S/R		NvD
Interface Element			PDP			VDP		VDP
swImplPolicy			fixed	const	standard	standard	queued	standard
Prm	PDP	fixed	yes	yes	yes	yes	no	yes
		const	no	yes	yes	yes	no	yes
		standard	no	no	yes	yes	no	yes
S/R	VDP	standard	no	no	no	yes	no	yes
		queued	no	no	no	no	yes	no
NvD	VDP	standard	no	no	no	yes	no	yes

Table 4.6: Overview of compatibility of ParameterDataPrototype and VariableDataPrototypes

Interface Element		
PDP	:	`ParameterDataPrototype`
VDP	:	`VariableDataPrototype`
Port Interface		
Prm	:	`ParameterInterface`
S/R	:	`SenderReceiverInterface`
NvD	:	`NvDataInterface`

Table 4.7: Key to table 4.6

parameter can be connected to almost every require port, except a queued Sender/receiver interface.

The RTE doesn't have to check the compatibility between ports since this task is performed at the VFB level. But it shall provide the right implementation of interface element and API according the attribute `swImplPolicy` attached to the interface element.

4.2.7.3 Implementation of an interface element

The implementation of an interface element depends on the attribute `swImplPolicy`. The attribute `swCalibrationAccess` determines how the interface element can be accessed by e.g. an external calibration tool. The table 4.8 defines the supported combinations of `swImplPolicy` and `swCalibrationAccess` attribute setting and gives the corresponding implementation by the RTE.

[4]calibration parameter have to be allocated in RAM if data emulation with SW support is required, see 4.2.8.3.5

swImplPolicy	SwCalibrationAccess			
	not Accessible	readOnly	readWrite	Implementation
fixed	yes	not supported	not supported	macro definition or c const declaration dependent from RTE optimization
const	yes	yes	not supported	c const declaration
standard	yes	yes	yes	standard implementation i.e. a variable for VariableDataPrototype in RAM or a calibration parameter in ROM [4]
queued	yes	not supported	not supported	FIFO Queue
measurement Point	not supported	yes	not supported	Variable

Table 4.8: Data implementation according `swImplPolicy`

4.2.7.4 Initialization of `VariableDataPrototypes`

Basically the need for initialization of any `VariableDataPrototypes` is specified by the Software Component Descriptions defining the `VariableDataPrototypes`. This information is basically defined by the existence of an `initValue`, the `sectionInitializationPolicy` of the related `SwAddrMethod`. As described in section 7.11 additionally the initialization strategy can be adjusted by the integrator of the RTE to adjust the behavior to the start-up code.

[rte_sws_7046]⌈ Variables implementing `VariableDataPrototypes` shall be initialized if

- an `initValue` is defined

  AND

- no `SwAddrMethod` is defined for `VariableDataPrototype`.

⌋*(RTE00052, RTE00068, RTE00116)*

[rte_sws_3852]⌈ Variables implementing `VariableDataPrototype`s shall be initialized if

- an `initValue` is defined

  AND

- a `SwAddrMethod` is defined for `VariableDataPrototype`

  AND

- the `RteInitializationStrategy` for the `sectionInitializationPolicy` of the related `SwAddrMethod` is NOT configured to `RTE_INITIALIZATION_STRATEGY_NONE`.

⌋*(RTE00052, RTE00068, RTE00116)*

4.2.8 Measurement and Calibration

4.2.8.1 General

Calibration is the process of adjusting an ECU SW to fulfill its tasks to control physical processes respectively to fit it to special project needs or environments. To do this two different mechanisms are required and have to be distinguished:

1. Measurement
   Measure what's going on in the ECU e.g. by monitoring communication data (Inter-ECU, Inter-Partition, Intra-partition, Intra-SWC). There are several ways to get the monitor data out of the ECU onto external visualization and interpretation tools.
2. Calibration
   Based on the measurement data the ECU behavior is modified by changing parameters like runtime SW switches, process controlling data of primitive or composite data type, interpolation curves or interpolation fields. In the following for such parameters the term calibration parameter is used.

With AUTOSAR, a calibration parameter is instantiated with a `ParameterDataPrototype` class that aggregates a `SwDataDefProps` with properties `swCalibrationAccess` = `readWrite` and `swImplPolicy` = `standard`.

Nevertheless it is supported, that `VariableDataPrototype` is instantiated that aggregates a `SwDataDefProps` with properties `swCalibrationAccess` = `readWrite` and `swImplPolicy` = `standard`. But in this case the implementation of such `VariableDataPrototype` is treated identical to `swCalibrationAccess` = `readOnly` and the RTE Generator has not to implement further measures (for instance "Data emulation with SW support" 4.2.8.3.5).

It's possible that different SwDataDefProps settings are specified for a VariableDataPrototype and its referenced AutosarDataType. In this case the rules specified in the SWC-T shall be applied. See as well rte_sws_7196.

SwDataDefProps contain more information how measurement values or characteristics are to be interpreted and presented by external calibration tools. This information is needed for the ASAM2 respectively A2L file generation. Afterwards the A2L file is used by ECU-external measurement and calibration tools so that these tools know e.g. how to interpret raw data received from ECU and how to get them.

4.2.8.1.1 Definition of Calibration Parameters

Calibration parameters can be defined in AUTOSAR SW as well as in Basic-SW. In the *AUTOSAR Architecture* there are two possibilities to define calibration parameters. Which one to choose is not in the focus of this RTE specification.

1. RTE provides the calibration parameter access if they are specified via a ParamaterSwComponentType. A ParamaterSwComponentType can be defined in order to provide ParameterDataPrototypes (via ports) to other Software Components.
2. Calibration parameter access invisible for RTE
   Since multiple instantiation with code sharing is not allowed for Basic-SW and multiple instantiation is not always required for software components it's possible for these software to define own methods how calibration parameters are allocated. Nevertheless these calibration parameters shall be described in the belonging *Basic Software Module Description* respectively *Software Component Description*. In case data emulation with SW-support is used, the whole software and tool chain for calibration and measurement, e.g. Basic-SW (respectively XCP driver) which handles emulation details and data exchange with external calibration tools then has to deal with several emulation methods at once: The one the RTE uses and the other ones each Basic-SW or SWC using local calibration parameters practices.

4.2.8.1.2 Online and offline calibration

The way how measurement and calibration is performed is company, domain and project specific. Nevertheless two different basic situations can be distinguished and are important for understanding:

1. Offline calibration
   Measure when ECU is running, change calibration data when ECU is off.
   Process might look like this:
   (a) Flash the ECU with current program file
   (b) PowerUp ECU in target (actual or emulated) environment

(c) Measure running ECU behavior - log or monitor via external tooling

(d) Switch off ECU

(e) Change calibration parameters and create a new flashable program file (hex-file) e.g. by performing a new SW make run

(f) Back to (a).

Do loop as long as a need for calibration parameter change exists or the Flash survives.

2. Online calibration

Do measurement and calibration in parallel.
In this case in principle all steps mentioned in "Offline calibration" above have to be performed in parallel. So other mechanisms are introduced avoiding ECU flashing when modifying ECU parameters. ECU works temporarily with changed data and when the calibration process is over the result is an updated set of calibration data. In next step this new data set might be merged into the existing program file or the new data set might be an input for a new SW make run. In both cases the output is a new program file to flash into the ECU.

Process might look like this:

(a) Flash the ECU with current program file

(b) PowerUp ECU in target environment

(c) Measure running ECU behavior and temporarily modify calibration parameters. Store set of updated calibration parameters (not on the ECU but on the calibration tool computer). Actions in step c) may be done iteratively.

(d) Switch off ECU

(e) Create a new flashable program file (hex-file) containing the new calibration parameters

Procedure over

4.2.8.2 Measurement

4.2.8.2.1 What can be measured

The AUTOSAR SW-C template specification [2] explains to which AUTOSAR prototypes a measurement pattern can be applied.

RTE provides measurement support for

1. communication between Ports
Measurable are

- VariableDataPrototypes of a SenderReceiverInterface used in a PortPrototype (of a SwComponentPrototype) to capture sender-receiver communication or between SwComponentPrototypes
- VariableDataPrototypes of a NvDataInterface used in a PortPrototype (of a SwComponentPrototype) to capture non volatile data communication or between SwComponentPrototypes
- ArgumentDataPrototypes of an ClientServerOperation in a ClientServerInterface to capture client-server communication between SwComponentPrototypes

2. communication inside of AUTOSAR SW-Cs
   Measurable are implicitInterRunnableVariable, explicitInterRunnableVariable or arTypedPerInstanceMemory
3. data structures inside a AUTOSAR NvBlockSwComponent
   Measurable are ramBlocks and romBlocks of a NvBlockSwComponent's *NvBlock*

Further on AUTOSAR SW-Cs and *Basic Software Module*s can define measurables which are not instantiated by RTE. These are described by VariableDataPrototypes in the role staticMemory. Hence those kind of measurables are not described in the generated *McSupportData* of the RTE (see 4.2.8.4).

4.2.8.2.2 RTE support for Measurement

The way how measurement data is read out of the ECU is not focus of the RTE specification. But the RTE structure and behavior must be specified in that way that measurement values can be provided by RTE during ECU program execution.

To avoid synchronization effort it shall be possible to read out measurement data asynchronously to RTE code execution. For this the measurement data must be stable. As a consequence this might forbid direct reuse of RAM locations for implementation of several AUTOSAR communications which are independent of each other but occurring sequentially in time (e.g. usage of same RAM cell to store uint8 data sender receiver communication data between Runnables at positions 3 and 7 and later the same RAM cell for the communication between Runnables at positions 9 and 14 of same periodically triggered task). So applying measurable elements might lead to less optimizations in the generated RTE's code and to increased RAM need.

There are circumstances when RTE will store same communication data in different RAM locations, e.g. when realizing implicit sender receiver communication or Inter Runnable Variables with implicit behavior. In these cases there is only the need to have the content of one of these stores made accessible from outside.

The information that measurement shall be supported by RTE is defined in applied SwDataDefProps:

The value readOnly or readWrite of the property swCalibrationAccess defines that measurement shall be supported, any other value of the property swCalibrationAccess is to be ignored for measurement.

Please note that the definition of rte_sws_3900 and rte_sws_3902 do not have further conditions when the location in memory has to be provided to support the usage of VariableDataPrototype with the swImplPolicy = measurementPoint. In case that the MCD system is permitted to access such a VariableDataPrototype the RTE is not allowed to do optimization which would prevent such measurement even if there is no consuming software component in the input configuration.

The memory locations containing measurement values are initialized according to rte_sws_7046 and rte_sws_3852.

[rte_sws_7044]⌈ The RTE generator shall reject input configurations in which a RunnableEntity defines a read access (VariableAccess in the role readLocalVariable, dataReadAccess, dataReceivePointByValue or dataReceivePointByArgument) to an VariableDataPrototype with a swImplPolicy set to measurementPoint. ⌋*(RTE00018)*

For sender-receiver resp. client-server communication same or compatible interfaces are used to specified connected ports. So very often measurement will be demanded two times for same or compatible VariableDataPrototype on provide and require side of a 1:1 communication resp. multiple times in case of 1:N or M:1 communication. In that case providing more than one measurement value for a VariableDataPrototype doesn't make sense and would increase ECU resources need excessively. Instead only one measurement value shall be provided.

Sender-receiver communication

[rte_sws_3900]⌈ If the swCalibrationAccess of a VariableDataPrototype used in an interface of a sender-receiver port of a SwComponentPrototype is set to readOnly or readWrite the RTE generator has to provide one reference to a location in memory where the actual content of the instance specific data of the corresponding VariableDataPrototype of the communication can be accessed. ⌋*(RTE00153)*

To prohibit multiple measurement values for same communication:
(Note that affected VariableDataPrototypes might be specified in same or compatible port interfaces.)

[rte_sws_3972]⌈ For 1:1 and 1:N sender-receiver communication the RTE shall provide measurement values taken from sender side if measurement is demanded in provide and require port. ⌋*(RTE00153)*

[rte_sws_3973]⌈ For N:1 intra-ECU sender-receiver communication the RTE shall provide measurement values taken from receiver side if measurement is demanded in provide and require ports. ⌋*(RTE00153)*

Note:
See further below for support of queued communication.

[rte_sws_3974]⌈ For a `VariableDataPrototype` with measurement demand associated with received data of inter-ECU sender-receiver communication the RTE shall provide only one measurement store reference containing the actual received data even if several receiver ports demand measurement. ⌋*(RTE00153)*

[rte_sws_7344]⌈ For a `VariableDataPrototype` with measurement demand associated with received data of inter-Partition sender-receiver communication the RTE shall provide only one measurement store reference per partition containing the actual received data even if several receiver ports demand measurement in the Partition. ⌋*(RTE00153)*

Client-Server communication

[rte_sws_3901]⌈ If the `swCalibrationAccess` of an `ArgumentDataPrototype` used in an interface of a client-server port of a `SwComponentPrototype` is set to `readOnly` the RTE generator has to provide one reference to a location in memory where the actual content of the instance specific argument data of the communication can be read. ⌋*(RTE00153)*

To prohibit multiple measurement values for same communication:
(Note that affected `ArgumentDataPrototype`s might be specified in same or compatible port interfaces.)

[rte_sws_3975]⌈ For intra-ECU client-server communication the RTE shall provide measurement values taken from client side if measurement of an `ArgumentDataPrototype`s is demanded by provide and require ports. ⌋*(RTE00153)*

[rte_sws_3976]⌈ For inter-ECU client-server communication with the client being present on same ECU as the RTE, the RTE shall provide measurement values taken from client side. ⌋*(RTE00153)*

[rte_sws_3977]⌈ For inter-ECU client-server communication with the server being present on same ECU as the RTE, the RTE shall provide measurement values taken from server if no client present on same ECU as the server is connected with that server too. ⌋*(RTE00153)*

[rte_sws_7349]⌈ For inter-Partition client-server communication with the server being present on the same ECU as the RTE, the RTE shall provide measurement values taken from server if no client present on the same Partition as the server is connected with that server too. ⌋*(RTE00153)*

Note:
When a measurement is applied to a client-server call additional copy code might be produced so that a zero overhead direct server invocation is no longer possible for this call.

Mode Switch Communication

[rte_sws_6700]⌈ If the `swCalibrationAccess` of a `ModeDeclarationGroupPrototype` used in an interface of a `mode switch port` of a `SwComponentPrototype` is set to `readOnly` the RTE generator has to provide three references to locations in memory where the *current mode*, the *previous mode* and the *next mode* of the related `mode machine instance` can be accessed. ⌋*(RTE00153)*

The affected `ModeDeclarationGroupPrototype`s might be used at different ports with the same or compatible port interfaces. rte_sws_6701 prohibits the occurrence of multiple measurement values for the same communication:

[rte_sws_6701]⌈ For 1:1 and 1:N mode switch communication the RTE shall provide measurement values taken from `mode manager` side if measurement is demanded in provide and require port. ⌋*(RTE00153)*

Inter Runnable Variables

[rte_sws_3902]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` in the role `implicitInterRunnableVariable` or `explicitInterRunnableVariable` is set to `readOnly` or `readWrite` the RTE generator has to provide one reference to a location in memory where the actual content of the *Inter Runnable Variable* can be accessed for a specific instantiation of the AUTOSAR SWC.
⌋*(RTE00153)*

PerInstanceMemory

[rte_sws_7160]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` in the role `arTypedPerInstanceMemory` is set to `readOnly` or `readWrite` the RTE generator has to provide one reference to a location in memory where the actual content of the `arTypedPerInstanceMemory` can be accessed for a specific instantiation of the AUTOSAR SWC.
⌋*(RTE00153)*

Nv RAM Block

[rte_sws_7174]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` in the role `ramBlock` of a `NvBlockSwComponentType`'s `NvBlockDescriptor` is set to `readOnly` or `readWrite` the RTE generator has to provide one reference to a location in memory where the actual content of the *Nv RAM Block* can be accessed for a specific instantiation of the AUTOSAR `NvBlockSwComponentType`.
⌋*(RTE00153)*

Non Volatile Data communication

[rte_sws_7197]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` used in an `NvDataInterface` of a non volatile data port of a `SwComponentPrototype` is set to `readOnly` or `readWrite` the RTE generator has to provide one reference to a location in memory where the actual content of the instance specific data of the corresponding `VariableDataPrototype` of the communication can be accessed. ⌋*(RTE00153)*

To prohibit multiple measurement values for same communication:
(Note that affected VariableDataPrototypes might be specified in same or compatible port interfaces.)

[rte_sws_7198]⌈ For 1:1 and 1:N non volatile data communication the RTE shall provide measurement values taken from ramBlock if measurement is demanded either in provide port, any require port (rte_sws_7197 or ramBlock (rte_sws_7174). ⌋*(RTE00153)*

Unconnected ports or compatible interfaces

As stated in section 5.2.7 RTE supports handling of unconnected ports.

Measurement support for unconnected sender-receiver provide ports makes sense since a port might be intentionally added for monitoring purposes only.

Measurement support for unconnected sender-receiver require ports makes sense since the measurement is specified on the type level of the Software Component and therefore independent of the individual usage of the Software Component. In case of unconnected sender-receiver require ports the measurement shall return the initial value.

Support for unconnected client-server provide port does not make sense since the server cannot be called and with this no data can be passed there.

Support for unconnected client-server require port makes sense since the measurement is specified on the type level of the Software Component and therefore independent of the individual usage of the Software Component. In case of unconnected client-server require ports the measurement shall return the actually provided and returned values.

[rte_sws_3978]⌈ For sender-receiver communication the RTE generator shall respect measurement demands enclosed in unconnected provide ports. ⌋*(RTE00139, RTE00153)*

[rte_sws_5101]⌈ For sender-receiver communication the RTE generator shall respect measurement demands enclosed in unconnected require ports and deliver the initial value. ⌋*(RTE00139, RTE00153)*

[rte_sws_3980]⌈ For client-server communication the RTE generator shall ignore measurement demands enclosed in unconnected provide ports. ⌋*(RTE00139, RTE00153)*

[rte_sws_5102]⌈ For client-server communication the RTE generator shall respect measurement demands enclosed in unconnected require ports. The behavior shall be similar as if the require port would be connected and the server does not respond. ⌋*(RTE00139, RTE00153)*

[rte_sws_5170]⌈ For client-server communication the RTE generator shall ignore measurement requests for queued client-server communication. ⌋*(RTE00139, RTE00153)*

In case the measurement of client-server communication is not possible due to requirement rte_sws_5170 the McSupportData need to reflect this (see rte_sws_5172).

In principle the same thoughts as above are applied to unused VariableDataPrototypes for sender-receiver communication where ports with compatible but not same interfaces are connected. It's no issue for client-server due to compatibility rules for client-server interfaces since in compatible client-server interfaces all ClientServerOperations have to be present in provide and require port (see AUTOSAR SW-C Template [2]).

[rte_sws_3979]⌈ For sender-receiver communication the RTE generator shall respect measurement demands of those VariableDataPrototypes in connected ports when provide and require port interfaces are not the same (but only compatible) even when a VariableDataPrototype in the provide port has no assigned VariableDataPrototype in the require port.
⌋*(RTE00153)*

General measurement disabling switch

To support saving of ECU resources for projects where measurement isn't required at all whereas enclosed AUTOSAR SW-Cs contain SwDataDefProps requiring it, it shall be possible to switch off support for measurement. This shall not influence support for calibration (see 4.2.8.3).

[rte_sws_3903]⌈ The RTE generator shall have the option to switch off support for measurement for generated RTE code. This option shall influence complete RTE code at once. ⌋*(RTE00153)*

There also might be projects in which monitoring of ECU internal behavior is required but calibration is not.

[rte_sws_3904]⌈ The enabling of RTE support for measurement shall be independent of the enabling of the RTE support for calibration. ⌋*(RTE00153)*

Queued communication

Measurement of queued communication is not supported yet. Reasons are:

- A queue can be empty. What's to measure then?
- Which of the queue entries is the one to take the data from might differ out of user view?
- Only quite inefficient solutions possible because implementation of queues entails storage of information dynamically at different memory locations. So always additional copies are required.

[rte_sws_3950]⌈ RTE generator shall reject configurations where measurement for queued sender-receiver communication is configured. ⌋*(RTE00153, RTE00018)*

4.2.8.3 Calibration

The RTE and *Basic Software Scheduler* has to support the allocation of calibration parameters and the access to them for SW using them. As seen later on for some calibration methods the RTE and *Basic Software Scheduler* must contain support SW too (see 4.2.8.3.5). But in general the RTE and *Basic Software Scheduler* is not responsible for the exchange of the calibration data values or the transportation of them between the ECU and external calibration tools.

The following sections are mentioning only the RTE but this has to be understood in the context that the support for *Calibration* is a functionality which affects the Basic Software Scheduler part of the RTE as well. In case of the *Basic Software Scheduler Generation Phase* (see 3.4.1) this functionality might even be provided with out any other software component related RTE functionality.

With AUTOSAR, a calibration parameter (which the AUTOSAR SW-C template specification [2] calls `ParameterSwComponentType`) is instantiated with a `ParameterDataPrototype` that aggregates a `SwDataDefProps` with properties `swCalibrationAccess` = `readWrite` and `swImplPolicy` = `standard`. This chapter applies to this kind of `ParameterSwComponentType`s. For other combinations of these properties, consult the section 4.2.7

4.2.8.3.1 Calibration parameters

Calibration parameters can be defined in `ParameterSwComponentType`s, in AUTOSAR SW-Cs, `NvBlockSwComponentType`s and in *Basic Software Module*s.

1. `ParameterSwComponentType`s don't have an internal behavior but contain `ParameterDataPrototype`s and serve to provide calibration parameters used commonly by several AUTOSAR SW-Cs. The use case that one or several of the user SW-Cs are instantiated on different ECUs is supported by instantiation of the `ParameterSwComponentType` on the affected ECUs too.
   Of course several AUTOSAR SW-Cs allocated on one ECU can commonly access the calibration parameters of `ParameterSwComponentType`s too. Also several instances of an AUTOSAR SW-Cs can share the same calibration parameters of a `ParameterSwComponentType`.
2. Calibration parameters defined in AUTOSAR SW-Cs can only be used inside the SW-C and are not visible to other SW-Cs. Instance individual and common calibration parameters accessible by all instances of an AUTOSAR SW-C are possible.
3. For `NvBlockSwComponentType`s it is supported to provide calibration access to the `ParameterDataPrototype` defining the `romBlock`. These values can not be directly accessed by AUTOSAR SW-Cs but are used to serve as *ROM Block* default values for the *Nv Block*.

4. Calibration parameters defined in *Basic Software Module*s can only be used inside the defining *Basic Software Module* and are not visible to other *Basic Software Module*s. In contrast to AUTOSAR SW-Cs, *Basic Software Module*s can only define instance specific calibration parameters.

[rte_sws_3958]⌈ Several AUTOSAR SW-Cs (and also several instances of AUTOSAR SW-Cs) shall be able to share same calibration parameters defined in `ParameterSwComponentType`s. ⌋*(RTE00154, RTE00159)*

[rte_sws_7186]⌈ The generated RTE shall initialize the memory objects implementing `ParameterDataPrototype`s in *p-port*s of `ParameterSwComponentType`s according the `ValueSpecification` of the `ParameterProvideComSpec` referring the `ParameterDataPrototype` in the *p-port*,

- if such `ParameterProvideComSpec` exists and
- if no `CalibrationParameterValue` refers to the `FlatInstanceDescriptor` associated to the `ParameterDataPrototype`

This is also applicable if the `swImplPolicy` = `fixed` and if the related `ParameterDataPrototype` is implemented as preprocessor define which does not immediately allocate a memory object. ⌋*(RTE00154, RTE00159)*

[rte_sws_7029]⌈ The generated RTE shall initialize the memory objects implementing `ParameterDataPrototype`s in *p-port*s of `ParameterSwComponentType`s according the `ValueSpecification` in the role `implInitValue` of the `CalibrationParameterValue` referring the `FlatInstanceDescriptor` associated to the `ParameterDataPrototype` if such `CalibrationParameterValue` is defined. ⌋*(RTE00154)*

Note: the initialization according rte_sws_7029 and rte_sws_7030 precedes the initialization values defined in the context of an component type and used in rte_sws_7185 and rte_sws_7186. This enables to provide initial values for calibration parameter instances to:

- predefine start values for the calibration process
- utilizes the result of the calibration process
- take calibration parameter values from previous projects

[rte_sws_3959]⌈ If the `SwcInternalBehavior` aggregates an `ParameterDataPrototype` in the role `perInstanceParameter` the RTE shall support the access to instance specific calibration parameters of the AUTOSAR SW-C. ⌋*(RTE00154, RTE00158)*

[rte_sws_5112]⌈ If the `SwcInternalBehavior` aggregates an `ParameterDataPrototype` in the role `sharedParameter` the RTE shall create a common access to the shared calibration parameter. ⌋*(RTE00154, RTE00159)*

[rte_sws_7096]⌈ If the `BswInternalBehavior` aggregates an `ParameterDataPrototype` in the role `perInstanceParameter` the *Basic Software Scheduler*

shall support the access to instance specific calibration parameters of the *Basic Software Module*. ⌋*(RTE00154, RTE00158)*

[rte_sws_7185]⌈ The generated RTE and *Basic Software Scheduler* shall initialize the memory objects implementing `ParameterDataPrototype` in the role `perInstanceParameter` or `sharedParameter`

- if it has a `ValueSpecification` in the role `initValue` according to this `ValueSpecification` and
- if no `CalibrationParameterValue` refer to the `FlatInstanceDescriptor` associated to the `ParameterDataPrototype`

This is also applicable if the `swImplPolicy` = `fixed` and if the related `ParameterDataPrototype` is implemented as preprocessor define which does not immediately allocate a memory object. ⌋*(RTE00154)*

[rte_sws_7030]⌈ The generated RTE and *Basic Software Scheduler* shall initialize the memory objects implementing `ParameterDataPrototype`s in the role `perInstanceParameter` or `sharedParameter` according the `ValueSpecification` in the role the `implInitValue` of the `CalibrationParameterValue` referring the `FlatInstanceDescriptor` associated to the `ParameterDataPrototype` if such `CalibrationParameterValue` is defined. ⌋*(RTE00154)*

It might be project specific or even project phase specific which calibration parameters have to be calibrated and which are assumed to be stable. So it shall be selectable on `ParameterSwComponentType`s and AUTOSAR SW-C granularity level for which calibration parameters RTE shall support calibration.

If an r-port contains a `ParameterDataPrototype`, the following requirements specify its behavior if the port is unconnected.

[rte_sws_2749]⌈ In case of an unconnected parameter r-port, the RTE shall set the values of the `ParameterDataPrototype`s of the r-port according to the `initValue` of the r-port's `ParameterRequireComSpec` referring to the `ParameterDataPrototype`. ⌋*(RTE00139, RTE00159)*

If the port is unconnected, RTE expects an init value, see rte_sws_2750.

`ParameterDataPrototype`s in role `romBlock`

[rte_sws_7033]⌈ If the `swCalibrationAccess` of a `ParameterDataPrototype` in the role `romBlock` is set to `readWrite` the RTE generator has to provide one reference to a location in memory where the actual content of the *romBlock* can be accessed. ⌋*(RTE00154)*

[rte_sws_7034]⌈ The generated RTE shall initialize any `ParameterDataPrototype` in the role `romBlock`

- if it has a `ValueSpecification` in the role `initValue` according to this `ValueSpecification` and

- if no CalibrationParameterValue refer to the FlatInstanceDescriptor associated to the ParameterDataPrototype

⌋*(RTE00154)*

[rte_sws_7035]⌈ The generated RTE shall initialize the memory objects implementing ParameterDataPrototypes in the role romBlock according the ValueSpecification in the role the implInitValue of the CalibrationParameterValue referring the FlatInstanceDescriptor associated to the ParameterDataPrototype if such CalibrationParameterValue is defined. ⌋*(RTE00154)*

ParameterDataPrototype used as romBlock are instantiated according to rte_sws_7693.

Configuration of calibration support

[rte_sws_3905]⌈ It shall be configurable for each ParameterSwComponentType if RTE calibration support for the enclosed ParameterDataPrototypes is enabled or not. ⌋*(RTE00154, RTE00156)*

[rte_sws_3906]⌈ It shall be configurable for each AUTOSAR SW-C if RTE calibration support for the enclosed ParameterDataPrototypes is enabled or not. ⌋*(RTE00154, RTE00156)*

RTE calibration support means the creation of SW as specified in section 4.2.8.3.5 "Data emulation with SW support".

Require ports on ParameterSwComponentTypes don't make sense. ParameterSwComponentTypes only have to provide calibration parameters to other Component types. So the RTE generator shall reject configurations containing require ports attached to ParameterSwComponentTypes. (see section A.13)

4.2.8.3.1.1 Separation of calibration parameters

Sometimes it is required that one or more calibration parameters out of the mass of calibration parameters of an ParameterSwComponentType respectively an AUTOSAR SW-C shall be placed in another memory location than the other parameters of the ParameterSwComponentType respectively the AUTOSAR SW-C. This might be due to security reasons (separate normal operation from monitoring calibration data in memory) or the possibility to change calibration data during a diagnosis session (which the calibration parameter located in NVRAM).

[rte_sws_3907]⌈ The RTE generator shall support separation of calibration parameters from ParameterSwComponentTypes, AUTOSAR SW-Cs and *Basic Software Modules* depending on the ParameterDataPrototype property swAddrMethod. ⌋*(RTE00154, RTE00158)*

4.2.8.3.2 Support for offline calibration

As described in section 4.2.8.1 when using an offline calibration process measurement is decoupled from providing new calibration parameters to the ECUs SW. During measurement phase information is collected needed to define to which values the calibration parameters are to be set best. Afterwards the new calibration parameter set is brought into the ECU e.g. by using a bootloader.

[rte_sws_3971]⌈ The RTE generator shall have the option to switch off all *data emulation* support for generated RTE code. This option shall influence complete RTE code at once. ⌋*(RTE00154, RTE00156)*

The term *data emulation* is related to mechanisms described in section 4.2.8.3.3.

Out of view of RTE the situation is same as when *data emulation without SW support* (described in section 4.2.8.3.4) is used:
The RTE is only responsible to provide access to the calibration parameters via the RTE API as specified in section 5.6. Exchange of `ParameterDataPrototype` content is done invisibly for ECU program flow and with this for RTE too.

When no *data emulation support* is required calibration parameter accesses to parameters stored in FLASH could be performed by direct memory read accesses without any indirection for those cases when accesses are coming out of single instantiated AUTOSAR SW-Cs or from *Basic Software Module*s. Nevertheless it's not goal of this specification to require direct accesses since this touches implementation. It might be ECU HW dependent or even be project dependent if other accesses are more efficient or provide other significant advantages or not.

4.2.8.3.3 Support for online calibration: Data emulation

To allow **online calibration** it must be possible to provide alternative calibration parameters invisible for application. The mechanisms behind are described here. We talk of *data emulation*.

In the following several calibration methods are described:

1. Data emulation without SW support and
2. several methods of data emulation with SW-support.

The term **data emulation** is used because the change of calibration parameters is emulated for the ECU SW which uses the calibration data. This change is invisible for the user-SW in the ECU.

RTE is significantly involved when SW support is required and has to create calibration method specific SW. Different calibration methods means different support in Basic SW which typically is ECU integrator specific. So it does not make sense to support DIFFERENT data emulation with SW support methods in ANY one RTE build. But

it makes sense that the RTE supports direct access (see section 4.2.8.3.4) for some AUTOSAR SW-Cs resp. `ParameterSwComponentType`s resp. *Basic Software Modules* and one of the data emulation with SW support methods (see section 4.2.8.3.5) for all the other AUTOSAR SW-Cs resp. `ParameterSwComponentType`s resp. *Basic Software Modules* at the same time.

[rte_sws_3909]⌈ The RTE shall support only one of the data emulation with SW support methods at once. ⌋*(RTE00154, RTE00156)*

4.2.8.3.4 Data emulation without SW support (direct access)

For "online calibration" (see section 4.2.8.1) the ECU is provided with additional hardware which consists of control logic and memory to store modified calibration parameters in. During ECU execution the brought in control logic redirects memory accesses to new bought in memory whose content is modified by external tooling without disturbing normal ECU program flow. Some microcontrollers contain features supporting this. A lot of smaller microcontrollers don't. So this methods is highly HW dependent.

To support these cases the RTE doesn't have to provide e.g. a reference table like described in section 4.2.8.3.5. Exchange of `ParameterDataPrototype` content is done invisibly for program flow and for RTE too.

[rte_sws_3942]⌈ The RTE generator shall have the option to switch off *data emulation with SW support* for generated RTE code. This option shall influence complete RTE code at once. ⌋*(RTE00154, RTE00156)*

4.2.8.3.5 Data emulation with SW support

In case "online calibration" (see section 4.2.8.1) is required, quite often data emulation without support by special SW constructs isn't possible. Several methods exist, all have the consequence that additional need of ECU resources like RAM, ROM/FLASH and runtime is required.

Data emulation with SW support is possible in different manners. During calibration process in each of these methods modified calibration data values are kept typically in RAM. Modification is controlled by ECU external tooling and supported by ECU internal SW located in AUTOSAR basic SW or in complex driver.

If calibration process isn't active the accessed calibration data is originated in ROM/FLASH respectively in NVRAM in special circumstances (as seen later on).

Since multiple instantiation is to be supported several instances of the same `ParameterDataPrototype`s have to be allocated. Because the RTE is the only one SW in an AUTOSAR ECU able to handle the different instances the access to these

calibration parameters can only be handled by the RTE. So the RTE has to provide additional SW constructs required for data emulation with SW support for calibration.

However the RTE doesn't know which of the ECU functionality shall be calibrated during a calibration session. To allow expensive RAM to be reused to calibrate different ECU functionalities in one or several online calibration sessions (see 4.2.8.1) in case of the single and double pointered methods for data emulation with SW support described below the RTE has only to provide the access to `ParameterDataPrototype`s during runtime but allowing other SW (a BSW module or a complex driver) to redirect the access to alternative calibration parameter values (e.g. located in RAM) invisibly for application.
The RTE is neither the instance to supply the alternative values for `ParameterDataPrototype`s nor in case of the pointered methods for data emulation with SW support to do the redirection to the alternative values.

[rte_sws_3910]⌈ The RTE shall support *data emulation with SW support* for calibration. ⌋*(RTE00154, RTE00156)*

[rte_sws_3943]⌈ The RTE shall support these data emulation methods with SW support:

- Single pointered calibration parameter access
  further called "single pointered method"
- Double pointered calibration parameter access further called "double pointered method"
- Initialized RAM parameters further called "initRAM parameter method"

⌋*(RTE00154, RTE00156)*

Please note that the support data emulation methods is applicable for calibration parameters provided for software components as well as calibration parameters provided for basic software modules.

ParameterElementGroup

To save RAM/ROM/FLASH resources in single pointered method and double pointered method `ParameterDataPrototype` allocation is done in groups. One entry of the calibration reference table references the begin of a group of `ParameterDataPrototype`s. For better understanding of the following, this group is called `ParameterElementGroup` (which is no term out of the AUTOSAR SW-C template specification [2]). One `ParameterElementGroup` can contain one or several `ParameterDataPrototype`s.

[rte_sws_3911]⌈ If data emulation with SW support is enabled, the RTE generator shall allocate all `ParameterDataPrototype`s marked with same property `swAddrMethod` of one instance of a `ParameterSwComponentType` consecutively. To-

gether they build a separate ParameterElementGroup. ⌋*(RTE00154, RTE00156, RTE00158)*

[rte_sws_3912]⌈ If data emulation with SW support is enabled, the RTE shall guarantee that all non-shared ParameterDataPrototypes marked with same property swAddrMethod of an AUTOSAR SWC instance are allocated consecutively. Together they build a separate ParameterElementGroup. ⌋*(RTE00154, RTE00158)*

[rte_sws_5194]⌈ If data emulation with SW support is enabled, the RTE shall guarantee that all shared ParameterDataPrototypes marked with same property swAddrMethod of an AUTOSAR SWC type are allocated consecutively. Together they build a separate ParameterElementGroup. ⌋*(RTE00154, RTE00158)*

It is not possible to access same calibration parameter inside of a ParameterSwComponentType via several ports. This is a consequence of the need to support the use case that a ParameterSwComponentType shall be able to contain several calibration parameters derived from one ParameterDataPrototype which is contained in one interface applied to several ports of the ParameterSwComponentType. Using only the ParameterDataPrototype names for the names of the elements of a ParameterElementGroup would lead to a name clash since then several elements with same name would have to created. So port prototype and ParameterDataPrototype name are concatenated to specify the ParameterElementGroup member names.
This use case cannot be applied to AUTOSAR SW-C internal calibration parameters since they cannot be accessed via AUTOSAR ports.

[rte_sws_3968]⌈ The names of the elements of a ParameterElementGroup derived from a ParameterSwComponentType shall be <port>_<element> where <port> is the short-name of the provided AUTOSAR port prototype and <element> the short-name of the ParameterDataPrototype within the ParameterInterface categorizing the PPort. ⌋*(RTE00154, RTE00156)*

4.2.8.3.5.1 Single pointered method

There is one calibration reference table in RAM with references to one or several ParameterElementGroups. Accesses to calibration parameters are indirectly performed via this reference table.

Action during calibration procedure e.g. calibration parameter value exchange is not focus of this specification. Nevertheless an example is given for better understanding.

Example how the exchange of calibration parameters could be done for single pointered method:

1. Fill a RAM buffer with the modified calibration parameter values for complete ParameterElementGroup
2. Modify the corresponding entry in the calibration reference table so that a redirection to new ParameterElementGroup is setup

Now calibration parameter accesses deliver the modified values.

Figure 4.23 illustrates the method.

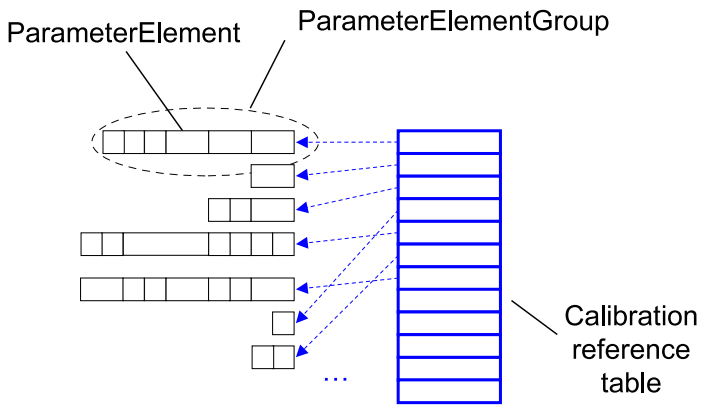


Figure 4.23: ParameterElementGroup in single pointered method context

[rte_sws_3913]⌈ If data emulation with SW support with single pointered method is enabled, the RTE generator shall create a table located in RAM with references to `ParameterElementGroups`. The type of the table is an array of void pointers. ⌋*(RTE00154, RTE00156)*

One reason why in this approach the calibration reference table is realized as an array is to make ECU internal reference allocation traceable for external tooling. Another is to allow a Basic-SW respectively a complex driver to emulate other calibration parameters which requires the standardization of the calibration reference table too.

[rte_sws_3947]⌈ If data emulation with SW support with single method is enabled the name (the label) of the calibration reference table shall be `<RteParameterRefTab>`. ⌋*(RTE00154, RTE00156)*

Calibration parameters located in NVRAM are handled same way (also see section 4.2.8.3.6).

[rte_sws_3936]⌈ If data emulation with SW support with single or double pointered method is enabled and calibration parameter respectively a `ParameterElement-Groups` is located in NVRAM the corresponding calibration reference table entry shall reference the PerInstanceMemory working as the NVRAM RAM buffer. ⌋*(RTE00154, RTE00156, RTE00157)*

4.2.8.3.5.2 Double pointered method

There is one calibration reference table in ROM respectively Flash with references to one or several `ParameterElementGroups`. Accesses to calibration parameters are performed through a double indirection access. During system startup the base

reference is initially filled with a reference to the calibration reference table.

Action during calibration procedure e.g. calibration parameter value exchange is not focus of this specification. Nevertheless an example is given for better understanding.

Example how the exchange of calibration parameters could be done for double pointered method:

1. Copy the calibration reference table into RAM
2. Fill a RAM buffer with modified calibration parameter values for complete `ParameterElementGroup`
3. Modify the corresponding entry in the RAM copy of the reference table so that a redirection to new `ParameterElementGroup` is setup
4. Change the content of the base reference so that it references the calibration reference table copy in RAM.

Now calibration parameter accesses deliver the modified values.

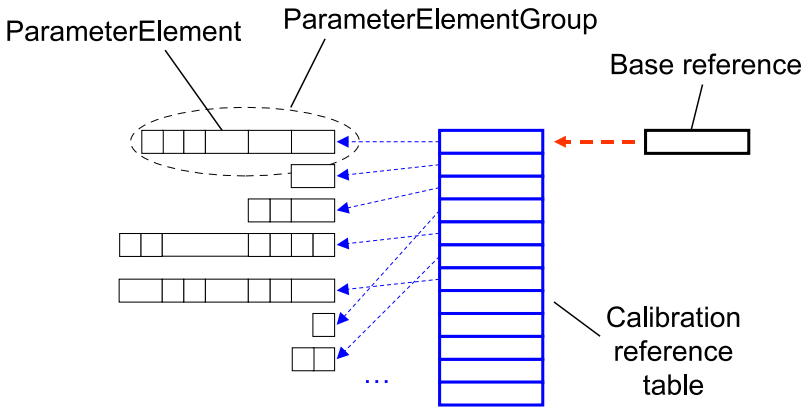


Figure 4.24: ParameterElementGroup in double pointered method context

[rte_sws_3914]⌈ If data emulation with SW support with double pointered method is enabled, the RTE generator shall create a table located in ROM respectively FLASH with references to `ParameterElementGroups`. The type of the table is an array of void pointers. ⌋*(RTE00154, RTE00156)*

Figure 4.24 illustrates the method.

To allow a Basic-SW respectively a complex driver to emulate other calibration parameters the standardization of the base reference is required.

[rte_sws_3948]⌈ If data emulation with SW support with double method is enabled the name (the label) of the calibration base reference shall be `<RteParameterBase>`. This label and the base reference type shall be exported and made available to other SW on same ECU.
⌋*(RTE00154, RTE00156)*

Calibration parameters located in NVRAM are handled same way (also see section 4.2.8.3.6).

For handling of calibration parameters located in NVRAM with single or double pointered method see rte_sws_3936 in section 4.2.8.3.5.1. General information is found in section 4.2.8.3.6).

4.2.8.3.5.3 InitRam parameter method

For each instance of a `ParameterDataPrototype` the RTE generator creates a calibration parameter in RAM and a corresponding value in ROM/FLASH. During startup of RTE the calibration parameter values of ROM/FLASH are copied into RAM. Accesses to calibration parameters are performed through a direct access to RAM without any indirection.

Action during calibration procedure e.g. calibration parameter value exchange is not focus of this specification. Nevertheless an example is given for better understanding: An implementation simply would have to exchange the content of the RAM cells during runtime.

[rte_sws_3915]⌈ If data emulation with SW support with initRam parameter method is enabled, the RTE generator shall create code guaranteeing that

1. calibration parameters are allocated in ROM/Flash and
2. a copy of them is allocated in RAM made available latest during RTE startup

for those `ParameterDataPrototype`s for which calibration support is enabled. ⌋*(RTE00154, RTE00156)*

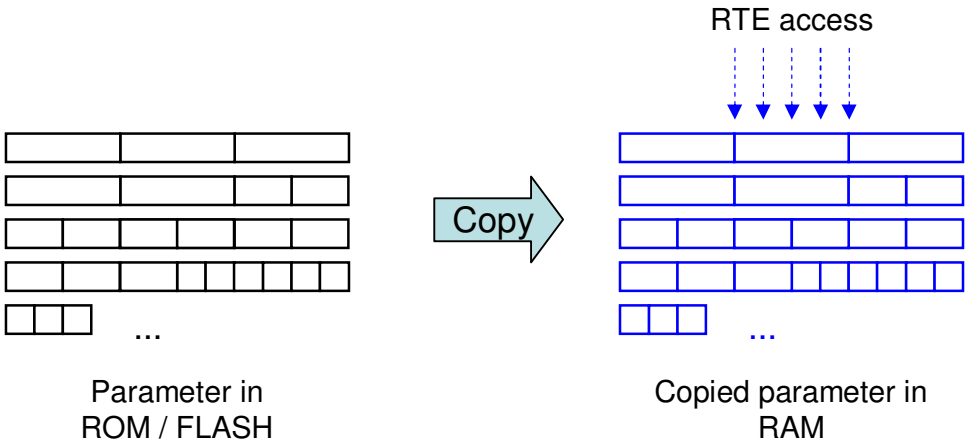


Figure 4.25: initRam Parameter method setup

Figure 4.25 illustrates the method.

A special case is the access of ParameterDataPrototypes instantiated in NVRAM (also see section 4.2.8.3.6). In this no extra RAM copy is required because a RAM location containing the calibration parameter value still exists.

[rte_sws_3935]⌈ If data emulation with SW support with initRam parameter method is enabled, the RTE generator shall create direct accesses to the PerInstanceMemory working as RAM buffer for the calibration parameters defined to be in NVRAM. ⌋*(RTE00154, RTE00156)*

4.2.8.3.5.4 Arrangement of a ParameterElementGroup for pointered methods

For data emulation with SW support with single or double pointered methods the RTE has to guarantee access to each single member of a ParameterElementGroup for source code and object code delivery independent if the member is a primitive or a composite data type. For this the creation of a record type for a ParameterElementGroup was chosen.

[rte_sws_3916]⌈ One ParameterElementGroup shall be realized as one record type. ⌋*(RTE00154, RTE00156)*

The sequence order of ParameterDataPrototype in a ParameterElementGroup and the order of ParameterElementGroups in the reference table will be documented by the RTE Generator by the means of the RteSwEmulationMethodSupport, see 4.2.8.4.4.

4.2.8.3.5.5 Further definitions for pointered methods

As stated in section 4.2.8.3.1.1, dependent of the value of property swAddrMethod calibration parameters shall be separated in different memory locations.

[rte_sws_3908]⌈ If data emulation with SW support with single or double pointered method is enabled the RTE shall create a separate instance specific ParameterElementGroup for all those ParameterDataPrototypes with a common value of the appended property swAddrMethod. Those ParameterDataPrototypes which have no property swAddrMethod appended, shall be grouped together too. ⌋*(RTE00154, RTE00156, RTE00158)*

To allow traceability for external tooling the sequence order of ParameterDataPrototype in a ParameterElementGroup and the order of ParameterElementGroups in the reference table will be documented by the RTE Generator by the means of the RteSwEmulationMethodSupport, see 4.2.8.4.4.

4.2.8.3.5.6 Calibration parameter access

Calibration parameters are derived from ParameterDataPrototypes. The RTE has to provide access to each calibration parameter via a separate API call.

API is specified in 5.6.

[rte_sws_3922]⌈ If data emulation with SW support and single or double pointered method is enabled the RTE generator shall export the label of the calibration reference table. ⌋*(RTE00154, RTE00156)*

[rte_sws_3960]⌈ If data emulation with SW support and double pointered method is enabled the RTE generator shall export the label and the type of the calibration base reference. ⌋*(RTE00154, RTE00156)*

[rte_sws_3932]⌈ If data emulation with SW support with single pointered method is enabled the RTE generator shall create API calls using single indirect access via the calibration reference table for those ParameterDataPrototypes which are in a ParameterElementGroup for which calibration is enabled. ⌋*(RTE00154, RTE00156)*

[rte_sws_3933]⌈ If data emulation with SW support with double pointered method is enabled the RTE generator shall create API calls using double indirection access via the calibration base reference and the calibration reference table for those ParameterDataPrototypes which are in a ParameterElementGroup for which calibration is enabled. ⌋*(RTE00154, RTE00156)*

[rte_sws_3934]⌈ If data emulation with SW support with double pointered method is enabled, the calibration base reference shall be located in RAM. ⌋*(RTE00154, RTE00156)*

4.2.8.3.5.7 Calibration parameter allocation

Since only the RTE knows which instances of AUTOSAR SW-Cs, ParameterSwComponentTypes and *Basic Software Module*s are present on the ECU the RTE has to allocate the calibration parameters and reserve memory for them. This approach is also covering multiple instantiated object code integration needs. So memory for instantiated ParameterDataPrototypes is neither provided by ParameterSwComponentTypes nor by AUTOSAR SW-C.

Nevertheless AUTOSAR SW-Cs and *Basic Software Module*s can define calibration parameters which are not instantiated by RTE. These are described by ParameterDataPrototypes in the role constantMemory. Further on the RTE can not implement any software support for data emulation for such calibration parameters. Hence those kind of calibration parameters are not described in the generated *McSupportData* of the RTE (see 4.2.8.4).

[rte_sws_3961]⌈ The RTE shall allocate the memory for calibration parameters. ⌋*(RTE00154, RTE00156)*

A ParameterDataPrototype can be defined to be instance specific or can be shared over all instances of an AUTOSAR SW-C or a ParameterSwComponentType. The input for the RTE generator contains the values the RTE shall apply to the calibration parameters.

To support online and offline calibration (see section 4.2.8.1) all parameter values for all instances have to be provided.
Background:

- For online calibration often initially the same default values for calibration parameters can be applied. Variation is then handled later by post link tools. Initial ECU startup is not jeopardized. This allows the usage of a default value e.g. by AUTOSAR SW-C or ParameterSwComponentType supplier for all instances of a ParameterDataPrototype.
- On the other hand applying separate default values for the different instances of a ParameterDataPrototype will be required often for online calibration too, to make a vehicle run initially. This requires additional configuration work e.g. for integrator.
- Offline calibration based on new SW build including new RTE build and compilation process requires all calibration parameter values for all instances to be available for RTE.

Shared ParameterDataPrototypes

[rte_sws_3962]⌈ For accesses to a shared ParameterDataPrototype the RTE API shall deliver the same one value independent of the instance the calibration parameter is assigned to. ⌋*(RTE00154, RTE00156)*

[rte_sws_3963]⌈ The calibration parameter of a shared ParameterDataPrototype shall be stored in one memory location only. ⌋*(RTE00154, RTE00156)*

Requirements rte_sws_3962 and rte_sws_3963 are to guarantee that only one physical location in memory has to be modified for a change of a shared ParameterDataPrototype. Otherwise this could lead to unforeseeable confusion.
Multiple locations are possible for calibration parameters stored in NVRAM. But there a shared ParameterDataPrototype is allowed to have only one logical data too.

Instance specific ParameterDataPrototypes

[rte_sws_3964]⌈ For accesses to an instance specific ParameterDataPrototype the RTE API shall deliver a separate calibration parameter value for each instance of a ParameterDataPrototype. ⌋*(RTE00154, RTE00156)*

[rte_sws_3965]⌈ For an instance specific ParameterDataPrototype the calibration parameter value of each instance of the ParameterDataPrototype shall be stored in a separate memory location. ⌋*(RTE00154, RTE00156)*

Usage of swAddrMethod

SwDataDefProps contain the optional property *swAddrMethod*. It contains meta information about the memory section in which a measurement data store resp. a calibration parameter shall be allocated in. This abstraction is needed to support the reuse of unmodified AUTOSAR SW-Cs resp. ParameterSwComponentTypes in different projects but allowing allocation of measurement data stores resp. calibration parameters in different sections.
Section usage typically depends on availability of HW resources. In one project the micro controller might have less internal RAM than in another project, requiring that most measurement data have to be placed in external RAM. In another project one addressing method (e.g. indexed addressing) might be more efficient for most of the measurement data - but not for all. Or some calibration parameters are accessed less often than others and could be - depending on project specific FLASH availability - placed in FLASH with slower access speed, others in FLASH with higher access speed.

[rte_sws_3981]⌈ The memory section used to store measurement values in shall be the memory sections associated with the swAddrMethod enclosed in the SwDataDefProps of a measurement definition. ⌋*(RTE00153)*

Since it's measurement data obviously this must be in RAM.

[rte_sws_3982]⌈ The memory section used to store calibration parameters in shall be the memory sections associated with the swAddrMethod enclosed in the SwDataDefProps of a calibration parameter definition. ⌋*(RTE00153)*

4.2.8.3.6 Calibration parameters in NVRAM

Calibration parameters can be located in NVRAM too. One use case for this is to have the possibility to modify calibration parameters via a diagnosis service without need for special calibration tool.

To allow NVRAM calibration parameters to be accessed, NVRAM with statically allocated RAM buffer in form of PIM memory for the calibration parameters has to be defined or the ramBlock of a NvBlockSwComponentType defines readWrite access for the MCD system. Please see as well rte_sws_7174 and rte_sws_7160.

Note:

As the NVRAM Manager might not be able to access the PerInstanceMemory across core boundaries in a multi core environment, the support of Calibration parameters in NVRAM for multi core controllers is limited. See also note in 4.2.9.1.

4.2.8.4 Generation of *McSupportData*

The RTE Generator supports the definition, allocation and access to measurement and calibration data for Software Components as well as for Basic Software. The

specific support of measurement and calibration tools however is neither in the focus of the RTE Generator nor AUTOSAR. This would require the generation of an "A2L"-file (like specified in [22]) which is the standard in this domain – but out of the focus of AUTOSAR.

The RTE Generator however shall support an intermediate exchange format called `McSupportData` which is building the bridge between the ECU software and the final "A2L"-file needed by the measurement and calibration tools. The details about the `McSupportData` format and the involved methodology are described in the Basic Software Module Description Template document [9].

In this section the requirements on the RTE Generator are collected which elements shall be provided in the `McSupportData` element.

4.2.8.4.1 Export of the *McSupportData*

Figure 4.26 shows the structure of the `McSupportData` element. The `McSupportData` element and its sub-content is part of the `Implementation` element. In case of the RTE this is the `BswImplementation` element which is generated / updated by the RTE Generator in the Generation Phase (see rte_sws_5086 in chapter 3.4.2).

[rte_sws_5118]⌈ The RTE Generator in Generation Phase shall create the `McSupportData` element as part of the `BswImplementation` description of the generated RTE. ⌋*(RTE00189)*

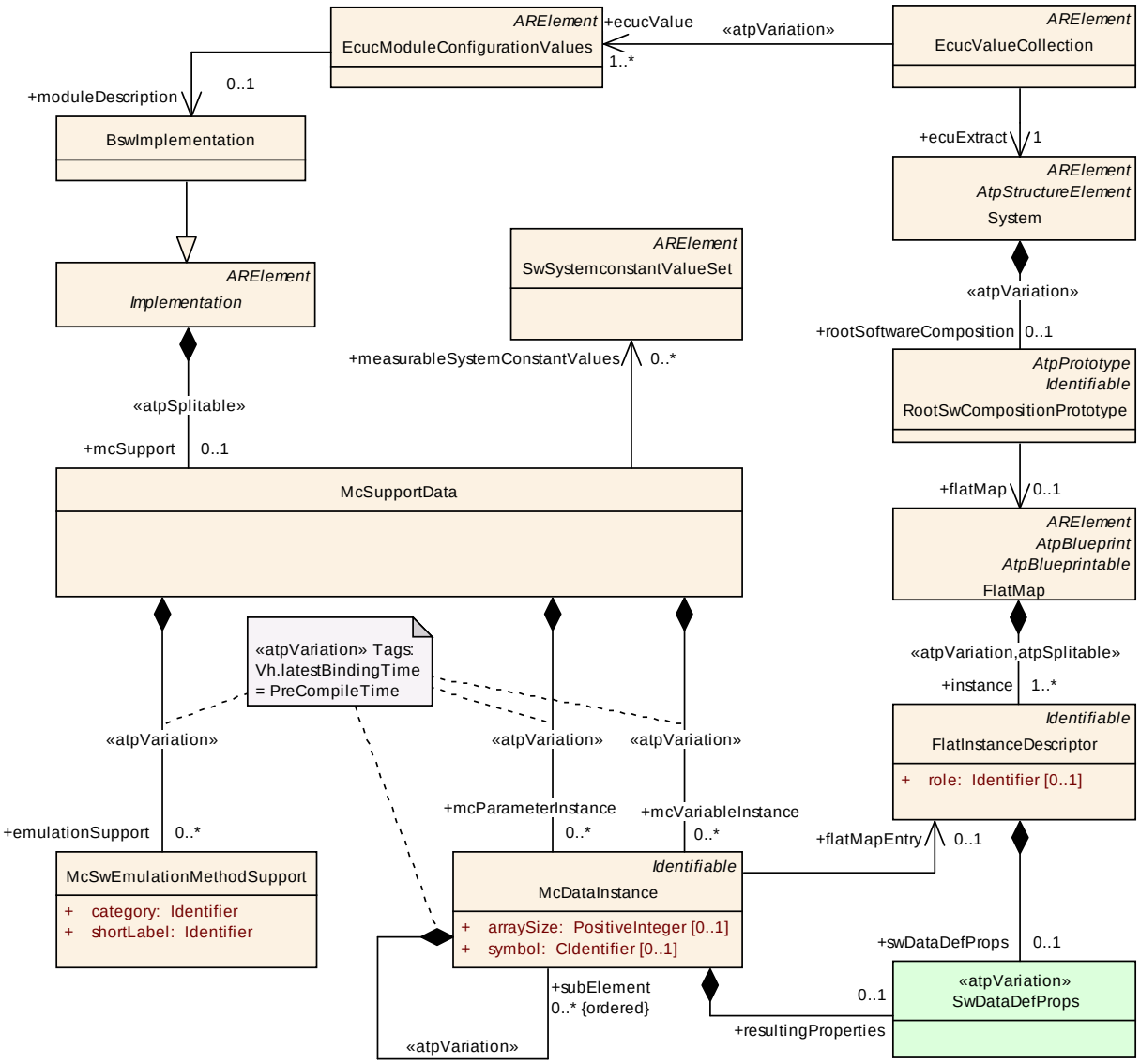


Figure 4.26: Overview of the `McSupportData` element

The individual measurable and calibratable data is described using the element `McDataInstance`. This is aggregated from `McSupportData` in the role `mcVariableInstance` (for measurement) or `mcParameterInstance` (for calibration).

Usage of the *FlatMap*

The `FlatMap` is part of the *Ecu Extract of System Description* and contains a collection of `FlatInstanceDescriptor` elements. The details of the `FlatMap` are described in the *Specification of the System Template* [8].

Common attributes of McDataInstance

The element MsDataInstance specifies one element of the McSupportData. The following requirement specify common attributes which shall to be filled in a harmonized way.

[rte_sws_5130]⌈ The RTE Generator shall use the shortName of the FlatInstanceDescriptor as the shortName of the McDataInstance. ⌋*(RTE00189)*

[rte_sws_5131]⌈ If the input element (e.g. ApplicationDataType or ImplementationDataType) has a Category specified the Category value shall be copied to the McDataInstance element. ⌋*(RTE00189)*

[rte_sws_5132]⌈ If the input element (e.g. ApplicationDataType or ImplementationDataType) specifies an array, the attribute arraySize of McDataInstance shall be set to the size of the array. ⌋*(RTE00189)*

[rte_sws_5133]⌈ If the input element (e.g. ApplicationDataType or ImplementationDataType) specifies a record, the McDataInstance shall aggregate the record element's parts as subElements of type McDataInstance. ⌋*(RTE00189)*

[rte_sws_5119]⌈ The McSupportData element and its sub-structure shall be self-contained in the sense that there is no need to deliver the whole upstream descriptions of the ECU (including the ECU Extract, Software Component descriptions, Basic Software Module descriptions, ECU Configuration Values descriptions, Flat Map, etc.) in order to later generate the final "A2L"-file. This means that the RTE Generator has to copy the required information from the upstream descriptions into the McSupportData element. ⌋*(RTE00189)*

[rte_sws_5129]⌈ The RTE Generator in Generation Phase shall export the effective SwDataDefProps (including all of the referenced and aggregated sub-elements like e.g. CompuMethod or SwRecordLayout) in the role resultingProperties for each McDataInstance after resolving the precedence rules defined in the SW-Component Template [2] chapter *Properties of Data Definitions*. ⌋*(RTE00189)*

[rte_sws_5135]⌈ If a ParameterDataPrototype is associated with a ParameterAccess the corresponding SwDataDefProps and their sub-structure shall be exported. ⌋*(RTE00189)*

[rte_sws_5134]⌈ For the export of the effective SwDataDefProps (rte_sws_5129) the information from the ApplicationDataType specification takes precedence over information from the ImplementationDataType. ⌋*(RTE00189)*

4.2.8.4.2 Export of Measurement information

Sender-Receiver communication

[rte_sws_5120]⌈ If the swCalibrationAccess of a VariableDataPrototype used in an interface of a sender-receiver port of a SwComponentPrototype is set to readOnly or readWrite and RteMeasurementSupport is set to *true* the RTE Generator shall create a McDataInstance element with

- symbol set to the C-symbol name used for the allocation (see also rte_sws_3900)
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the VariableDataPrototype

⌋*(RTE00153, RTE00189)*

Client-Server communication

[rte_sws_5121]⌈ If the swCalibrationAccess of an ArgumentDataPrototype used in an interface of a client-server port of a SwComponentPrototype is set to readOnly and RteMeasurementSupport is set to *true* the RTE Generator shall create a McDataInstance element with

- symbol set to the C-symbol name used for the allocation (see also rte_sws_3901)
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ArgumentDataPrototype

⌋*(RTE00153, RTE00189)*

[rte_sws_5172]⌈ If the measurement of client-server communication is ignored due to requirement rte_sws_5170 the corresponding McDataInstance in the McSupportData shall have a resultingProperties swCalibrationAccess set to notAccessible. ⌋*(RTE00153)*

Mode Switch Communication

[rte_sws_6702]⌈ If the swCalibrationAccess of a ModeDeclarationGroupPrototype used in an interface of a mode switch port of a SwComponentPrototype is set to readOnly and RteMeasurementSupport is set to *true* the RTE Generator shall create three McDataInstance elements with

- symbol set to the C-symbol name used for the allocation (see also rte_sws_6700)
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ModeDeclarationGroupPrototype

Thereby the McDataInstance element corresponding to the

- *current mode* has to reference the FlatInstanceDescriptor which role attribute is set to CURRENT_MODE,
- *previous mode* has to reference the FlatInstanceDescriptor which role attribute is set to PREVIOUS_MODE and
- *next mode* has to reference the FlatInstanceDescriptor which role attribute is set to NEXT_MODE

⌋*(RTE00153, RTE00189)*

InterRunnableVariable

[rte_sws_5122]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` in the role `implicitInterRunnableVariable` or `explicitInterRunnableVariable` is set to `readOnly` or `readWrite` and `RteMeasurementSupport` is set to *true* the RTE Generator shall create a `McDataInstance` element with

- `symbol` set to the C-symbol name used for the allocation (see also rte_sws_3902)
- `flatMapEntry` referencing to the corresponding `FlatInstanceDescriptor` element of the `VariableDataPrototype`

⌋*(RTE00153, RTE00189)*

PerInstanceMemory

[rte_sws_5123]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` in the role `arTypedPerInstanceMemory` is set to `readOnly` or `readWrite` and `RteMeasurementSupport` is set to *true* the RTE Generator shall create a `McDataInstance` element with

- `symbol` set to the C-symbol name used for the allocation (see also rte_sws_7160)
- `flatMapEntry` referencing to the corresponding `FlatInstanceDescriptor` element of the `VariableDataPrototype`

⌋*(RTE00153, RTE00189)*

Nv RAM Block

[rte_sws_5124]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` in the role `ramBlock` of a `NvBlockSwComponentType`'s `NvBlockDescriptor` is set to `readOnly` or `readWrite` and `RteMeasurementSupport` is set to *true* the RTE Generator shall create a `McDataInstance` element with

- `symbol` set to the C-symbol name used for the allocation (see also rte_sws_7174)
- `flatMapEntry` referencing to the corresponding `FlatInstanceDescriptor` element of the `NvBlockSwComponentType`

⌋*(RTE00153, RTE00189)*

Non Volatile Data communication

[rte_sws_5125]⌈ If the `swCalibrationAccess` of a `VariableDataPrototype` used in an `NvDataInterface` of a non volatile data port of a `SwComponentPrototype` is set to `readOnly` or `readWrite` and `RteMeasurementSupport` is set to *true* the RTE Generator shall create a `McDataInstance` element with

- `symbol` set to the C-symbol name used for the allocation (see also rte_sws_7197)

- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the VariableDataPrototype

⌋*(RTE00153, RTE00189)*

4.2.8.4.3 Export Calibration information

Calibration can be either actively supported by the RTE using the pre-defined calibration mechanisms of section 4.2.8.3.5 or calibration can be transparent to the RTE. In both cases the location and attributes of the calibratable data has to be provided by the RTE Generator in the Generation Phase in order to support the setup of the measurement and calibration tools.

ParameterDataPrototypes of ParameterSwComponentType

[rte_sws_5126]⌈ For each ParameterDataPrototype in a PortPrototype of a ParameterSwComponentType with the swCalibrationAccess set to readOnly or readWrite an entry in the McSupportData with the role mcParameterInstance shall be created with the following attributes:

- symbol set to the C-symbol name used for the allocation
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ParameterDataPrototype

⌋*(RTE00189)*

Shared ParameterDataPrototypes

[rte_sws_5127]⌈ For each ParameterDataPrototype of a AtomicSwComponentType's SwcInternalBehavior aggregated in the role sharedParameter with the swCalibrationAccess set to readOnly or readWrite an entry in the McSupportData with the role mcParameterInstance shall be created with the following attributes:

- symbol set to the C-symbol name used for the allocation
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ParameterDataPrototype

⌋*(RTE00189)*

Instance specific ParameterDataPrototypes

[rte_sws_5128]⌈ For each ParameterDataPrototype of a AtomicSwComponentType's SwcInternalBehavior aggregated in the role perInstanceParameter with the swCalibrationAccess set to readOnly or readWrite an entry in the McSupportData with the role mcParameterInstance shall be created with the following attributes:

- symbol set to the C-symbol name used for the allocation

- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ParameterDataPrototype

⌋*(RTE00189)*

[rte_sws_7097]⌈ For each ParameterDataPrototype of a BswModuleDescription's BswInternalBehavior aggregated in the role perInstanceParameter with the swCalibrationAccess set to readOnly or readWrite an entry in the McSupportData with the role mcParameterInstance shall be created with the following attributes:

- symbol set to the C-symbol name used for the allocation
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ParameterDataPrototype

⌋*(RTE00189)*

NvRom Block

[rte_sws_5136]⌈ If the swCalibrationAccess of a ParameterDataPrototype in the role romBlock is set to readOnly or readWrite an entry in the McSupportData with the role mcParameterInstance shall be created with the following attributes:

- symbol set to the C-symbol name used for the allocation in rte_sws_7033
- flatMapEntry referencing to the corresponding FlatInstanceDescriptor element of the ParameterDataPrototype

⌋*(RTE00153, RTE00189)*

4.2.8.4.4 Export of the Calibration Method

The RTE does provide several Software Emulation Methods which can be selected in the Ecu Configuration of the RTE (see section 7.3).

Which Software Emulation Method has been used for a particular RTE Generation shall be documented in the McSupportData in order to allow measurement and calibration tools to support the RTE's Software Emulation Methods. Additionally it is also possible for an RTE Vendor to add custom Software Emulation Methods which needs to be documented as well. The structure of the McSwEmulationMethodSupport is shown in figure 4.27.

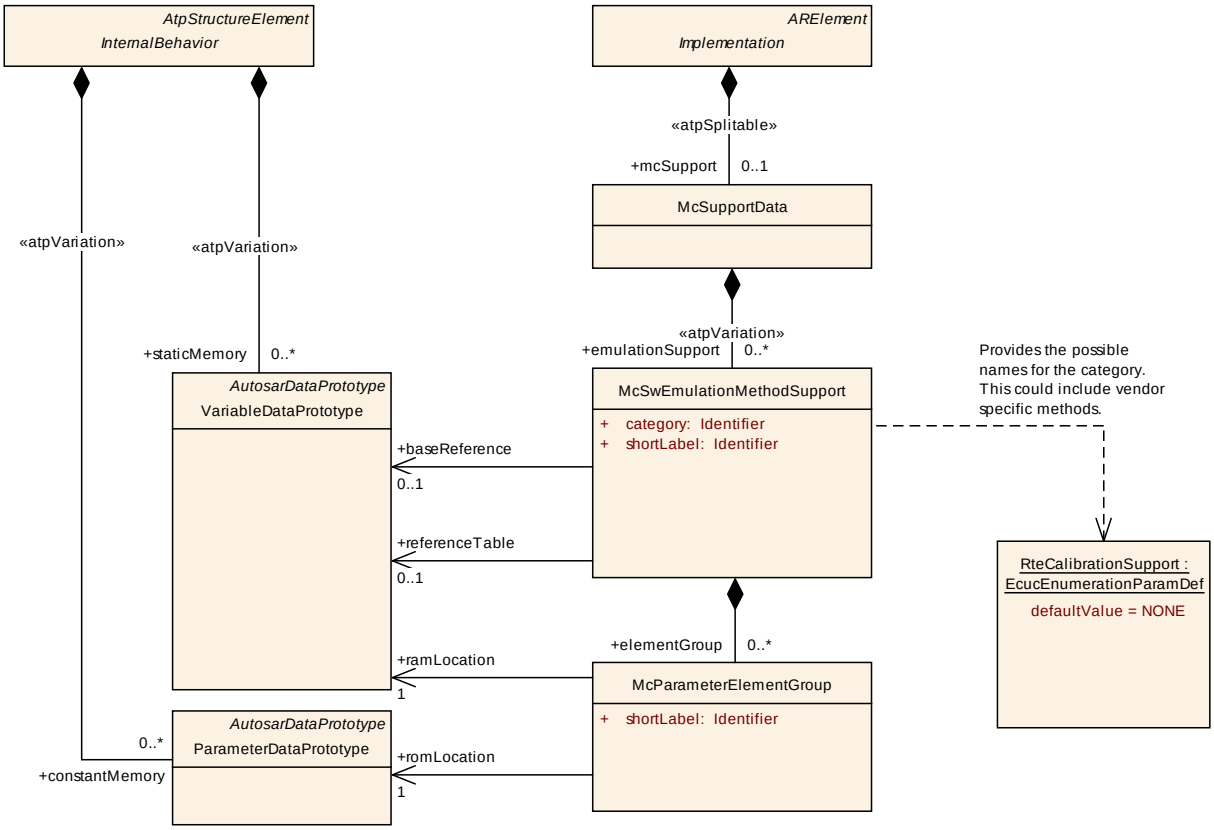


Figure 4.27: Structure of the `McSwEmulationMethodSupport` element

[rte_sws_5137]⌈ The RTE Generator in Generation Phase shall create the `McSwEmulationMethodSupport` element as part of the `McSupportData` description of the generated RTE. ⌋*(RTE00189)*

[rte_sws_5138]⌈ The RTE Generator in Generation Phase shall set the value of the `category` attribute of `McSwEmulationMethodSupport` element according to the implemented Software Emulation Method based on the Ecu configuration parameter `RteCalibrationSupport`:

- `NONE`
- `SINGLE_POINTERED`
- `DOUBLE_POINTERED`
- `INITIALIZED_RAM`
- *custom category name*: vendor specific Software Emulation Method

⌋*(RTE00189)*

The description of the generated structures is using the existing mechanisms already available in the Basic Software Module Description Template [9].

Description of ParameterElementGroup

For the description of the ParameterElementGroup an ImplementationDataType representing a structure of the group is created (rte_sws_5139).

[rte_sws_5139]⌈ For each generated ParameterElementGroup an ImplementationDataType shall be created. The contained ParameterDataPrototypes are aggregated with the role subElement as ImplementationDataTypeElement. ⌋*(RTE00189)*

In the example figure 4.28 the ImplementationDataTypes are called RteMcSupportGroupType1 and RteMcSupportGroupType2.

McSupport description of the InitRam parameter method

For the description of the InitRam parameter method the specific ParameterElementGroups allocated in ram and rom are specified (rte_sws_5140 and rte_sws_5141). Then the collection and correspondence of these groups is specified (in rte_sws_5142).

[rte_sws_5140]⌈ If the RTE Generator is configured to support the (INITIALIZED_RAM) method the RTE Generator in generation phase shall generate for each ParameterElementGroup a ParameterDataPrototype with the role constantMemory in the InternalBehavior of the RTE's Basic Software Module Description. The ParameterDataPrototype shall have a reference to the corresponding ImplementationDataType from rte_sws_5139 with the role type. ⌋*(RTE00189)*

[rte_sws_5141]⌈ If the RTE Generator is configured to support the (INITIALIZED_RAM) method the RTE Generator in generation phase shall generate for each ParameterElementGroup a VariableDataPrototype with the role staticMemory in the InternalBehavior of the RTE's Basic Software Module Description. The VariableDataPrototype shall have a reference to the corresponding ImplementationDataType from rte_sws_5139 with the role type. ⌋*(RTE00189)*

[rte_sws_5142]⌈ If the RTE Generator is configured to support the (INITIALIZED_RAM) method the RTE Generator in generation phase shall generate for each ParameterElementGroup a McParameterElementGroup with the role elementGroup in the McSwEmulationMethodSupport rte_sws_5137 element.

- The McParameterElementGroup shall have a reference to the corresponding ParameterDataPrototype from rte_sws_5140 with the role romLocation.
- The McParameterElementGroup shall have a reference to the corresponding VariableDataPrototype from rte_sws_5141 with the role ramLocation.

⌋*(RTE00189)*

McSupport description of the Single pointered method

For the description of the Single pointered method the specific ParameterElementGroups allocated in rom are specified (rte_sws_5143). Then an array data type is

specified which contains as many number of elements (void pointers) as there are ParameterElementGroups (rte_sws_5144). Then the instance of this array is specified in ram (rte_sws_5152) and referenced from the McSwEmulationMethodSupport (rte_sws_5153). The actual values for each array element are specified as references to the ParameterElementGroup prototypes (rte_sws_5154).

[rte_sws_5143]⌈ If the RTE Generator is configured to support the (SINGLE_POINTERED) method the RTE Generator in generation phase shall generate for each ParameterElementGroup a ParameterDataPrototype with the role constantMemory in the InternalBehavior of the RTE's Basic Software Module Description. The ParameterDataPrototype shall have a reference to the corresponding ImplementationDataType from rte_sws_5139 with the role type. ⌋*(RTE00189)*

[rte_sws_5144]⌈ If the RTE Generator is configured to support the (SINGLE_POINTERED) method the RTE Generator in generation phase shall generate an ImplementationDataType with one ImplementationDataTypeElement in the role subElement.

- The ImplementationDataTypeElement shall have the attribute arraySize set to the number of ParameterElementGroups from rte_sws_5139.
- The ImplementationDataTypeElement shall have a SwDataDefProps element with a reference to an ImplementationDataType representing a *void pointer*, in the role implementationDataType.

⌋*(RTE00189)*

[rte_sws_5152]⌈ If the RTE Generator is configured to support the (SINGLE_POINTERED) method the RTE Generator in generation phase shall generate a VariableDataPrototype with the role staticMemory in the InternalBehavior of the RTE's Basic Software Module Description. The VariableDataPrototype shall have a reference to the ImplementationDataType from rte_sws_5144 with the role type. ⌋*(RTE00189)*

[rte_sws_5153]⌈ If the RTE Generator is configured to support the (SINGLE_POINTERED) method the RTE Generator in generation phase shall generate a reference from the McSwEmulationMethodSupport rte_sws_5137 element to the VariableDataPrototype rte_sws_5152 in the role referenceTable. ⌋*(RTE00189)*

[rte_sws_5154]⌈ If the RTE Generator is configured to support the (SINGLE_POINTERED) method the RTE Generator in generation phase shall generate an ArrayValueSpecification as the initValue of the array rte_sws_5152 and for each ParameterElementGroup a ReferenceValueSpecification element in the ArrayValueSpecification defining the references to the individual ParameterElementGroup prototypes rte_sws_5143. ⌋*(RTE00189)*

McSupport description of the Double pointered method

The description of the Double pointered method is quite similar to the Single pointered method, but the allocation to ram and rom is different and it allocates the additional pointer parameter. The specific ParameterElementGroups allocated in rom are specified (rte_sws_5155). Then an array data type is specified which contains as many number of elements (void pointers) as there are ParameterElementGroups (rte_sws_5156). Then the instance of this array is specified in rom (rte_sws_5157) and referenced from the McSwEmulationMethodSupport (rte_sws_5158). The actual values for each array element are specified as references to the ParameterElementGroup prototypes (rte_sws_5159). Then the type of the base pointer is then created (rte_sws_5160) and an instance is allocated in ram (rte_sws_5161). The reference is initialized to the array in rom (rte_sws_5162).

[rte_sws_5155]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate for each ParameterElementGroup a ParameterDataPrototype with the role constantMemory in the InternalBehavior of the RTE's Basic Software Module Description. The ParameterDataPrototype shall have a reference to the corresponding ImplementationDataType from rte_sws_5139 with the role type. ⌋*(RTE00189)*

In the example figure 4.28 the ParameterDataPrototypes are called RteMcSupportParamGroup1 and RteMcSupportParamGroup1.

[rte_sws_5156]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate an ImplementationDataType with one ImplementationDataTypeElement in the role subElement.

- The ImplementationDataTypeElement shall be of category ARRAY with the attribute arraySize set to the number of ParameterElementGroups from rte_sws_5139.
- The ImplementationDataTypeElement shall have a SwDataDefProps element with a reference to an ImplementationDataType representing a *void pointer*, in the role implementationDataType.

⌋*(RTE00189)*

In the example figure 4.28 the ImplementationDataType is called RteMcSupportPointerTableType.

[rte_sws_5157]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate a ParameterDataPrototype with the role constantMemory in the InternalBehavior of the RTE's Basic Software Module Description. The ParameterDataPrototype shall have a reference to the ImplementationDataType from rte_sws_5156 with the role type. ⌋*(RTE00189)*

In the example figure 4.28 the ParameterDataPrototype is called RteMcSupportPointerTable.

[rte_sws_5158]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate a reference from the McSwEmulationMethodSupport rte_sws_5137 element to the ParameterDataPrototype rte_sws_5157 in the role referenceTable. ⌋*(RTE00189)*

[rte_sws_5159]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate an ArrayValueSpecification as the initValue of the array rte_sws_5157 and for each ParameterElementGroup a ReferenceValueSpecification element in the ArrayValueSpecification defining the references to the individual ParameterElementGroup prototypes rte_sws_5155. ⌋*(RTE00189)*

In the example figure 4.28 the ArrayValueSpecification is called RteMcSupportPointerTableInit. The ReferenceValueSpecifications are called RteMcSupportParamGroup1Ref and RteMcSupportParamGroup2Ref.

[rte_sws_5160]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate an ImplementationDataType with one ImplementationDataTypeElement being a reference to the array type from rte_sws_5156. ⌋*(RTE00189)*

In the example figure 4.28 the ImplementationDataType is called RteMcSupportBasePointerType.

[rte_sws_5161]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate a VariableDataPrototype with the role staticMemory in the InternalBehavior of the RTE's Basic Software Module Description. The VariableDataPrototype shall have a reference to the ImplementationDataType from rte_sws_5160 with the role type. ⌋*(RTE00189)*

In the example figure 4.28 the VariableDataPrototype is called RteMcSupportBasePointer.

[rte_sws_5162]⌈ If the RTE Generator is configured to support the (DOUBLE_POINTERED) method the RTE Generator in generation phase shall generate a ReferenceValueSpecification to the array from rte_sws_5157 as the initValue of the reference rte_sws_5161. ⌋*(RTE00189)*

In the example figure 4.28 the ReferenceValueSpecification is called RteMcSupportBasePointerInit.

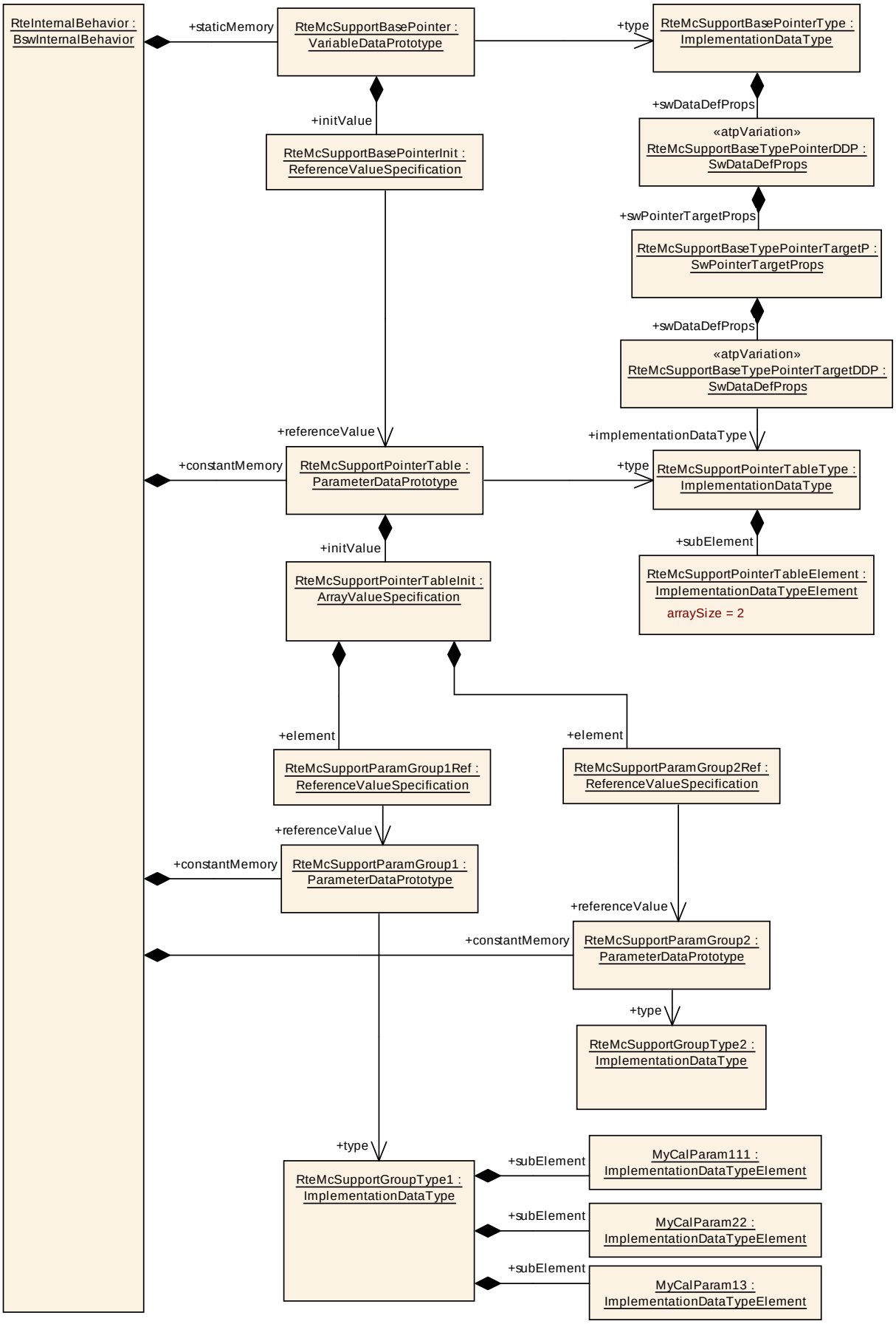

RteInternalBehavior : BswInternalBehavior
+staticMemory
RteMcSupportBasePointer : VariableDataPrototype
+type
RteMcSupportBasePointerType : ImplementationDataType
+initValue
RteMcSupportBasePointerInit : ReferenceValueSpecification
+swDataDefProps
«atpVariation» RteMcSupportBaseTypePointerDDP : SwDataDefProps
+swPointerTargetProps
RteMcSupportBaseTypePointerTargetP : SwPointerTargetProps
+swDataDefProps
«atpVariation» RteMcSupportBaseTypePointerTargetDDP : SwDataDefProps
+referenceValue
+implementationDataType
+constantMemory
RteMcSupportPointerTable : ParameterDataPrototype
+type
RteMcSupportPointerTableType : ImplementationDataType
+initValue
+subElement
RteMcSupportPointerTableInit : ArrayValueSpecification
RteMcSupportPointerTableElement : ImplementationDataTypeElement
arraySize = 2
+element
+element
RteMcSupportParamGroup1Ref : ReferenceValueSpecification
RteMcSupportParamGroup2Ref : ReferenceValueSpecification
+referenceValue
+constantMemory
RteMcSupportParamGroup1 : ParameterDataPrototype
+referenceValue
+constantMemory
RteMcSupportParamGroup2 : ParameterDataPrototype
+type
RteMcSupportGroupType2 : ImplementationDataType
+type
+subElement
RteMcSupportGroupType1 : ImplementationDataType
MyCalParam111 : ImplementationDataTypeElement
+subElement
MyCalParam22 : ImplementationDataTypeElement
+subElement
MyCalParam13 : ImplementationDataTypeElement

Figure 4.28: Example of the structure for Double Pointered Method

4.2.8.4.5 Export of Variant Handling

The Rte Generator shall provide information on values of system constants. The values are part of the input information and need to be collected and copied into a dedicated artifact to be delivered with the `McSupportData`.

[rte_sws_5168]⌈ The Rte Generator in generation phase shall create an elements of type `SwSystemconstantValueSet` and create copies of all system constant values found in the input information of type `SwSystemconstValue` where the referenced `SwSystemconst` element has the `swCalibrationAccess` set to `readOnly`. ⌋*(RTE00153, RTE00191)*

In case the `SwSystemconstValue` is subject to variability and the variability can be resolved during Rte generation phase

[rte_sws_5176]⌈ If a `SwSystemconst` with `swCalibrationAccess` set to `readOnly` has an assigned `SwSystemconstValue` which is subject to variability with the latest binding time `SystemDesignTime` or `CodeGenerationTime` the related `SwSystemconstValue` copy in the `SwSystemconstantValueSet` according to rte_sws_5168 shall contain the resolved value. ⌋*(RTE00153, RTE00191)*

[rte_sws_5174]⌈ If a `SwSystemconst` with `swCalibrationAccess` set to `readOnly` has an assigned `SwSystemconstValue` which is subject to variability with the latest binding time `PreCompileTime` the related `SwSystemconstValue` copy in the `SwSystemconstantValueSet` according to rte_sws_5168 shall have an `AttributeValueVariationPoint`. The *PreBuild* conditions of the `AttributeValueVariationPoint` shall correspond to the *PreBuild* conditions of the input `SwSystemconstValue`'s conditions. ⌋*(RTE00153, RTE00191)*

[rte_sws_5169]⌈ The Rte Generator in generation phase shall create a reference from the `McSupportData` element (rte_sws_5118) to the `SwSystemconstantValueSet` element (rte_sws_5168). ⌋*(RTE00153, RTE00191)*

In case the RTE Generator implements variability on a element which is accessible by a MCD system the related existence condition has to be documented in the `McSupportData` structure as well.

[rte_sws_5175]⌈ If an element in the `McSupportData` is related to an element in the input configuration which is subject to variability with the latest binding time `PreCompileTime` or *PostBuild* the RTE Generator shall add a `VariationPoint` for such element. The *PreBuild* and *PostBuild* conditions of the `VariationPoint` shall correspond to the *PreBuild* and *PostBuild* conditions of the input element's conditions. ⌋*(RTE00153, RTE00191)*

4.2.9 Access to NVRAM data

4.2.9.1 General

There are different methods available for AUTOSAR SW-Cs to access data stored in NVRAM:

- **"Calibration data"** – Calibrations can be stored in NVRAM, but are not modified during a "normal" execution of the ECU. Calibrations are usually directly read from their memory location, but can also be read from a RAM buffer when the access time needs to be optimized (e.g. for interpolation tables). They are described in section 4.2.8.
- **"Access to NVM blocks"** – This method uses `PerInstanceMemory` as a RAM mirror for the NVRAM blocks. While this method is efficient, its use is restricted.

  The NVRAM Manager [23] is a BSW module which provides services for SW-C to access NVRAM blocks during runtime. The NVM block data is not accessed directly, but through a RAM mirror, which can be a `PerInstanceMemory` instantiated by the RTE, or a SW-C internal buffer. When this method is used, the RTE does not provide any data consistency mechanisms (i.e. different runnables from the SW-C and the NVM can access the RAM mirror concurrently without being protected by the RTE).

  Note:

  This mechanism permits efficient usage of NVRAM data, but requires the SW-C designer to take care that accesses to the `PerInstanceMemory` from different task contexts don't cause data inconsistencies. The "Access to NVM blocks" should not be used in multi core environments. In AUTOSAR release 4.0, it can not be expected that the NVRAM Manager can access the `PerInstanceMemory` of another core. The presence of a shared memory section is not required by AUTOSAR. Only in the case of `arTypedPerInstanceMemory`, a `SwDataDefProps` item is available to assign the `PerInstanceMemory` to a shared memory section.
- **"Access to NVRAM data with a `NvBlockSwComponentType`** – The data is accessed through a `NvDataInterface` connected to a `NvBlockSwComponentType`s. This access is modeled at the VFB level, and, when necessary, protected by the RTE against concurrent accesses. It will be described further in this section.

4.2.9.2 Usage of the NvBlockSwComponentType

The code of NvBlock `SwComponentPrototype`s is implemented by the RTE Generator. `NvBlockSwComponentType`s provide a port interface for the access and management of data stored in NVRAM.

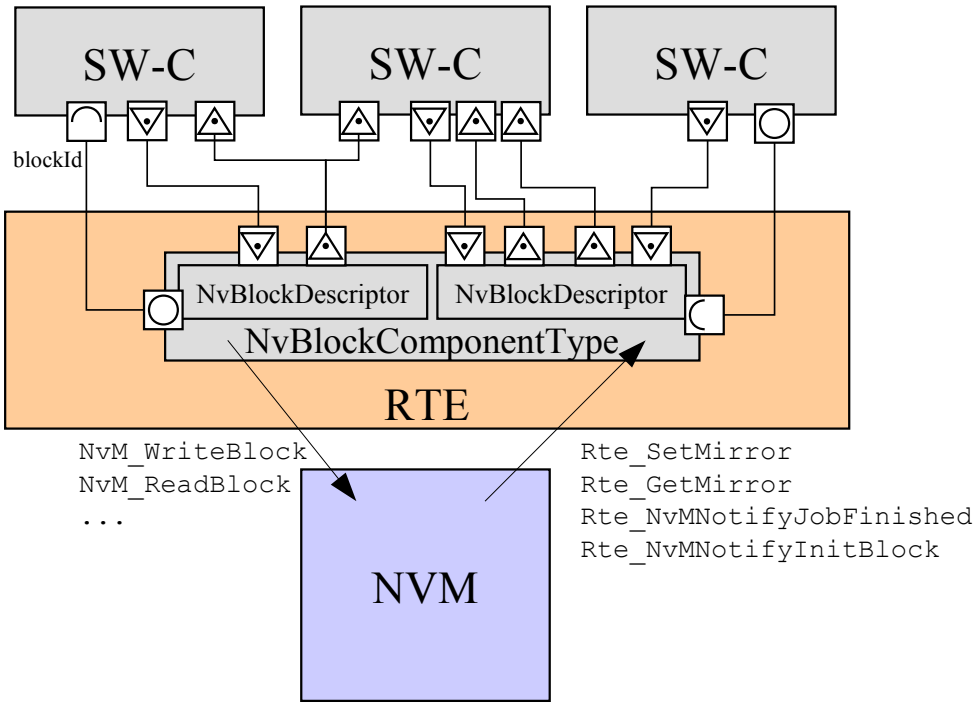


Figure 4.29: Connection to the `NvBlockSwComponentType`

Figure 4.29 illustrates the usage of a `NvBlockSwComponentType`.

[rte_sws_7301]⌈ Several AUTOSAR SW-Cs (and also several instances of a AUTOSAR SW-C) shall be able to read the same `VariableDataPrototype`s of a `NvBlockSwComponentType`. ⌋*(RTE00176)*

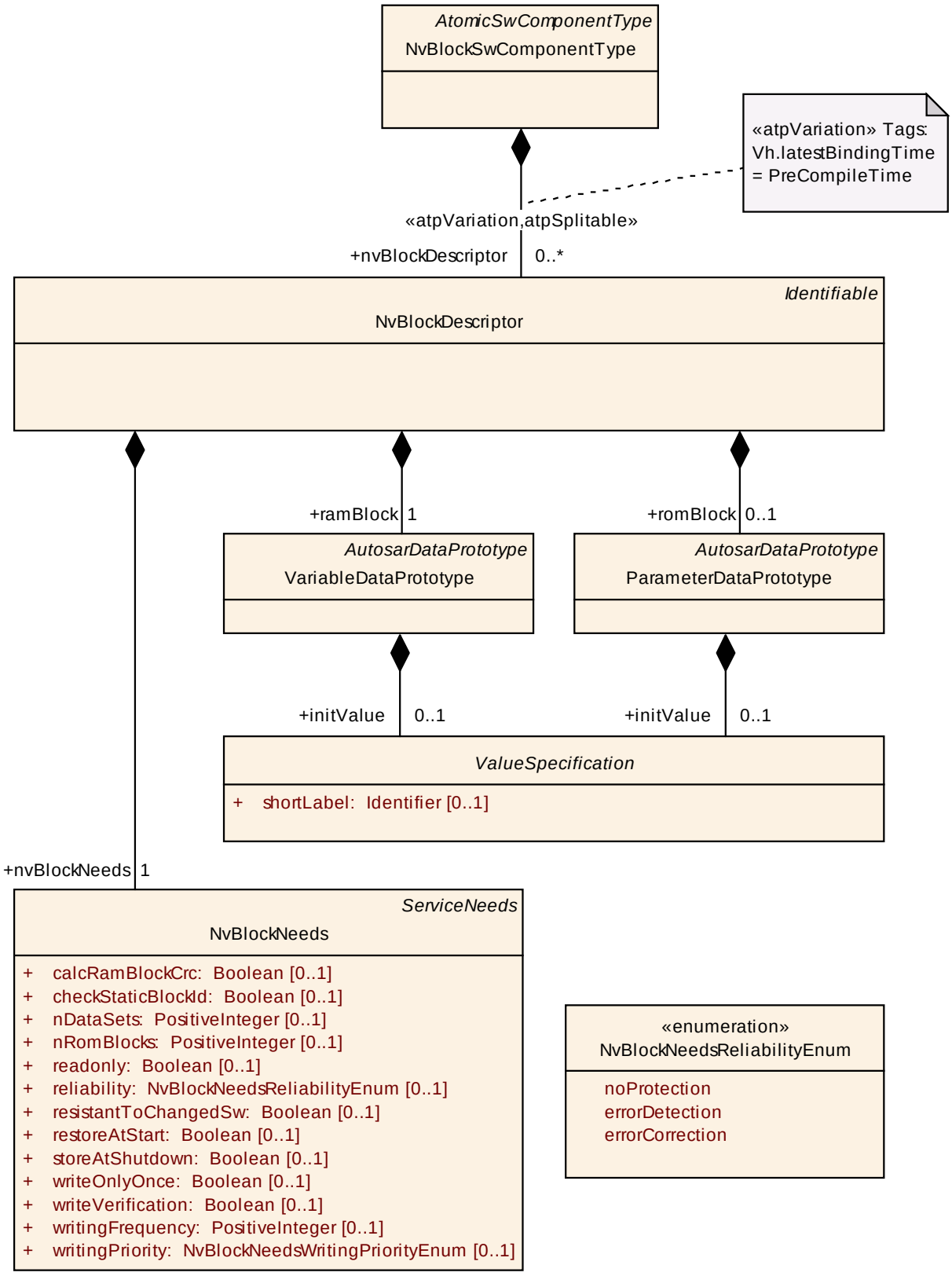


Figure 4.30: `NvBlockSwComponentType` and `NvBlockDescriptor`

A `NvBlockSwComponentType` contains multiple `NvBlockDescriptor`s. Each of these `NvBlockDescriptor` is associated to exactly one NVM block.

A NvBlockDescriptor contains a VariableDataPrototype which acts as a RAM mirror for the NVM block, and possibly a ParameterDataPrototype to act as the default ROM value for the NVM block.

[rte_sws_7353]⌈ The RTE Generator shall reject configurations where a NvBlockDescriptor of a NvBlockSwComponentType contains a romBlock whose data type is not compatible with the type of the ramBlock. ⌋*(RTE00177, RTE00018)*

[rte_sws_7303]⌈ The RTE shall allocate memory for the ramBlock VariableDataPrototype of the NvBlockDescriptor instances. ⌋*(RTE00177)*

[rte_sws_7632]⌈ The variables allocated for the ramBlocks shall be initialized if the general initialization conditions in rte_sws_7046 are fulfilled. The initialization as to be applied during Rte_Start and Rte_RestartPartition depending from the configured RteInitializationStrategy. ⌋*(RTE00177)*

Note: When blocks are configured to be read by NvM_ReadAll, the initialization may erase the value read by the NVM. These blocks should not have an initValue.

[rte_sws_7355]⌈ For each NvBlockDescriptor with a romBlock ParameterDataPrototype, the RTE shall allocate a constant ROM block. ⌋*(RTE00177)*

[rte_sws_7633]⌈ The constants allocated for the romBlocks shall be initialized to the value of the initValue, if they have an initValue. ⌋*(RTE00177)*

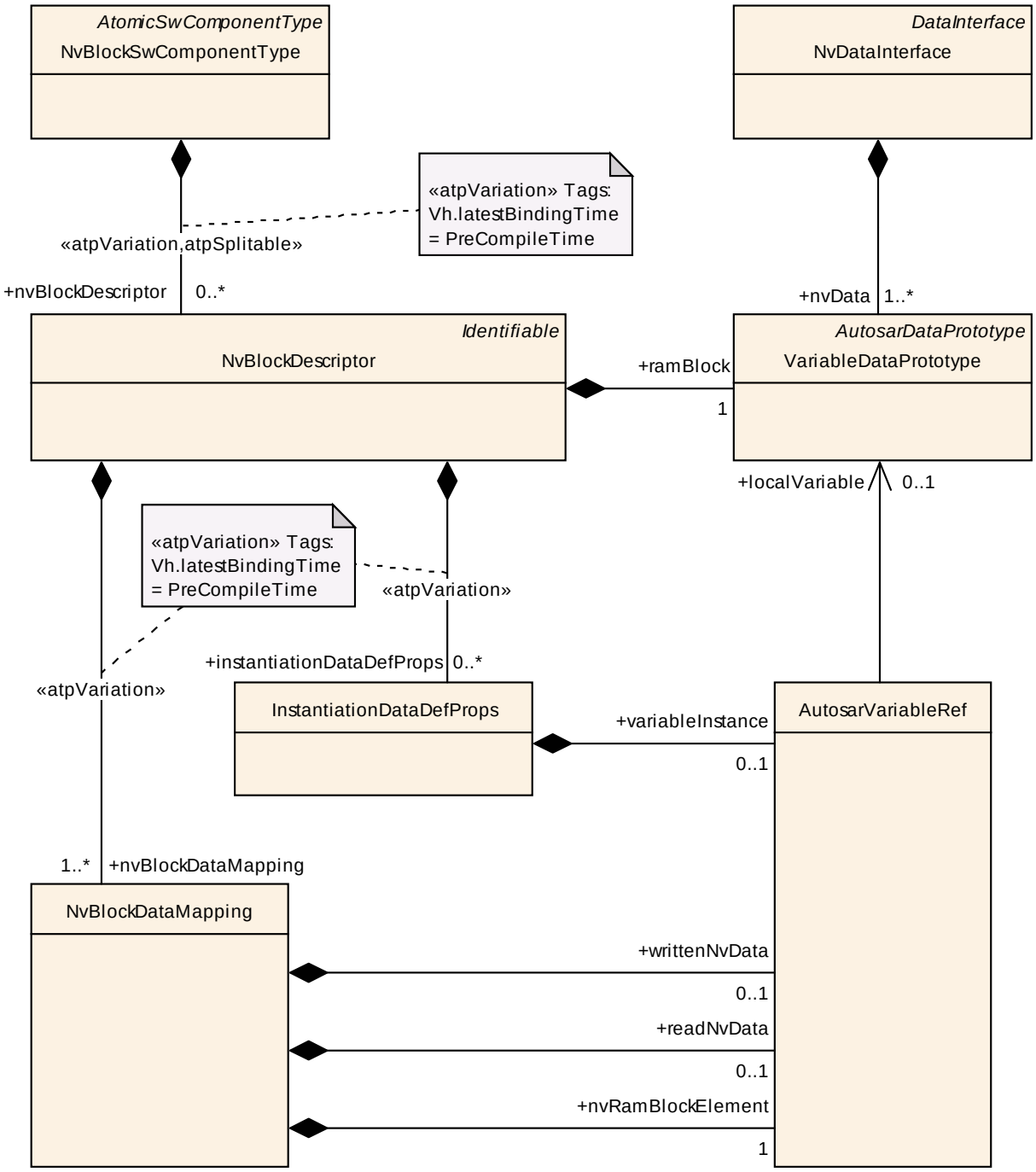


Figure 4.31: `NvBlockDataMapping`

For each element stored in the NvM block of a `NvBlockDescriptor`, there should be one `NvBlockDataMapping` to associate the `VariableDataPrototype`s of the ports used for read and write access and the `VariableDataPrototype` defining the location of the element in the `ramBlock`.

[rte_sws_7621]⌈ The RTE Generator shall reject configurations where a `NvBlock-DataMapping` references a `VariableDataPrototype` of the provide port (`writtenNvData`), a `VariableDataPrototype` of the require port (`readNvData`),

and a VariableDataPrototype defining the storage in the ramBlock which are not of compatible DataTypes. ⌋*(RTE00018)*

[rte_sws_7343]⌈ The RTE Generator shall reject configurations where a VariableDataPrototype instance in the role ramBlock is accessed by SW-C instances of different partitions. ⌋*(RTE00177, RTE00018)*

The rational for rte_sws_7343 is to allow the implementation of cleanup activities in case of termination or restart of a partition. These cleanup activities may require to invalidate the RAM mirror or reload data from the NVRAM device, which would impact other partitions if a the ramBlock is shared by SW-Cs of different partitions.

A NvBlockSwComponentType can be used to reduce the quantity of NVRAM blocks needed on an ECU:

- the same block can be used to store different flags or other small DataElements;
- the same DataElement can be used by different SW-Cs or different instances of a SW-C.

It also permits to simplify processes and algorithms when it must be guaranteed that two SW-Cs of an ECU use the same NVRAM data.

Note: this feature can increase the RAM usage of the ECU because it forces the NVRAM Manager to instantiate an additional RAM buffer. However, when the same DataElements have to be shared between SW-Cs, it reduces the number of RAM mirrors needed to be instantiated by the RTE, and can reduce the overall RAM usage of the ECU.

[rte_sws_7356]⌈ The RTE Generator shall reject configurations where a VariableDataPrototype referenced by a NvDataInterface has a *queued* swImplPolicy. ⌋*(RTE00018)*

[rte_sws_7357]⌈ The RTE Generator shall reject configurations where a DataReceivedEvent is referenced by a WaitPoint and references a VariableDataPrototype referenced by a NvDataInterface. ⌋*(RTE00018)*

[rte_sws_ext_7351] The NVM block associated to the NvBlockDescriptors of a NvBlockSwComponentType shall be configured with the NvmBlockUseSyncMechanism feature enabled, and the NvmWriteRamBlockToNvm and NvmReadRamBlockFromNvm parameters set to the Rte_GetMirror and Rte_SetMirror API of the NvBlockDescriptor.

An NvBlockSwComponentType may have unconnected p-ports or r-ports (see rte_sws_1329).

[rte_sws_7669]⌈ An NvBlockSwComponentType with an unconnected r-port shall behave as if no updated data were received for VariableDataPrototypes this unconnected r-port. ⌋*(RTE00139)*

4.2.9.3 Interface of the NvBlockSwComponentType

4.2.9.3.1 Access to the NVRAM data

The NvBlockSwComponentType provides PPortPrototypes and RPortPrototypes with an NvDataInterface data Sender-Receiver semantic to read the value of the NVRAM data or write the new value.

Like the SenderReceiverInterfaces, each of these NvDataInterfaces can provide access to multiple VariableDataPrototypes.

The same Rte_Read, Rte_IRead, Rte_DRead, Rte_Write, Rte_IWrite, Rte_IWriteRef APIs are used to access these VariableDataPrototypes as for SenderReceiverInterfaces.

[rte_sws_7667]⌈ The RTE Generator shall reject configurations where an r-port typed with an NvDataInterface is not connected and no NvRequireComSpec with a initValue are provided for each VariableDataPrototype of this NvDataInterface. This requirement does not apply if the r-port belongs to a NvBlockSwComponentType. ⌋*(RTE00018, RTE00139)*

rte_sws_7667 is required to avoid unconnected r-port without a defined initValue. Please note that for NvBlockSwComponent unconnected r-ports without init values are not a fault because the init values are defined in the NvBlockDescriptors ramBlock (see as well rte_sws_7632, rte_sws_7669)

[rte_sws_7668]⌈ The RTE shall initialize the VariableDataPrototypes of an r-port according to the initValue of the r-port's NvRequireComSpec referring to the VariableDataPrototype. ⌋*(RTE00139, RTE00108, RTE00068)*

4.2.9.3.2 NVM interfaces

The NvBlockSwComponentType can also have ports used for NV data management and typed by Client-Server interfaces compatible to the NVRAM Manager [23] standardized one. Note that these ports shall always have a PortInterface with the attribute isService set to FALSE.

The standardized NvM Client-Server interfaces are composed as follows:

- NvMService

  This interface is used to send commands to the NVM. The NvBlockSwComponentType provides a server port intended to be used by the SW-C users of this NvBlockSwComponentType.

- NvMNotifyJobFinished

  This interface is used by the NVM to notify the end of job. The NvBlockSwComponentType provides a server port intended to be used by the NVM, and client

ports intended to be connected to the SW-C users of this `NvBlockSwComponentType`.

- `NvMNotifyInitBlock`

  This interface is used by the NVM to request users to provide the default values in the RAM mirror. The `NvBlockSwComponentType` provides a server port intended to be used by the NVM, and client ports intended to be connected to the SW-C users of this `NvBlockSwComponentType`.

- `NvMAdmin`

  This interface is used to order some administrative operations to the NVM. The `NvBlockSwComponentType` provides a server port intended to be used by the SW-C users of this `NvBlockSwComponentType`.

Note: no restrictions have been added to the NVM interfaces. However, some operations of the NVM might require cooperation between the different users of the `NvBlockSwComponentType`. For example, a ReadBlock operation will erase the RAM mirror, which might affect multiple SW-Cs.

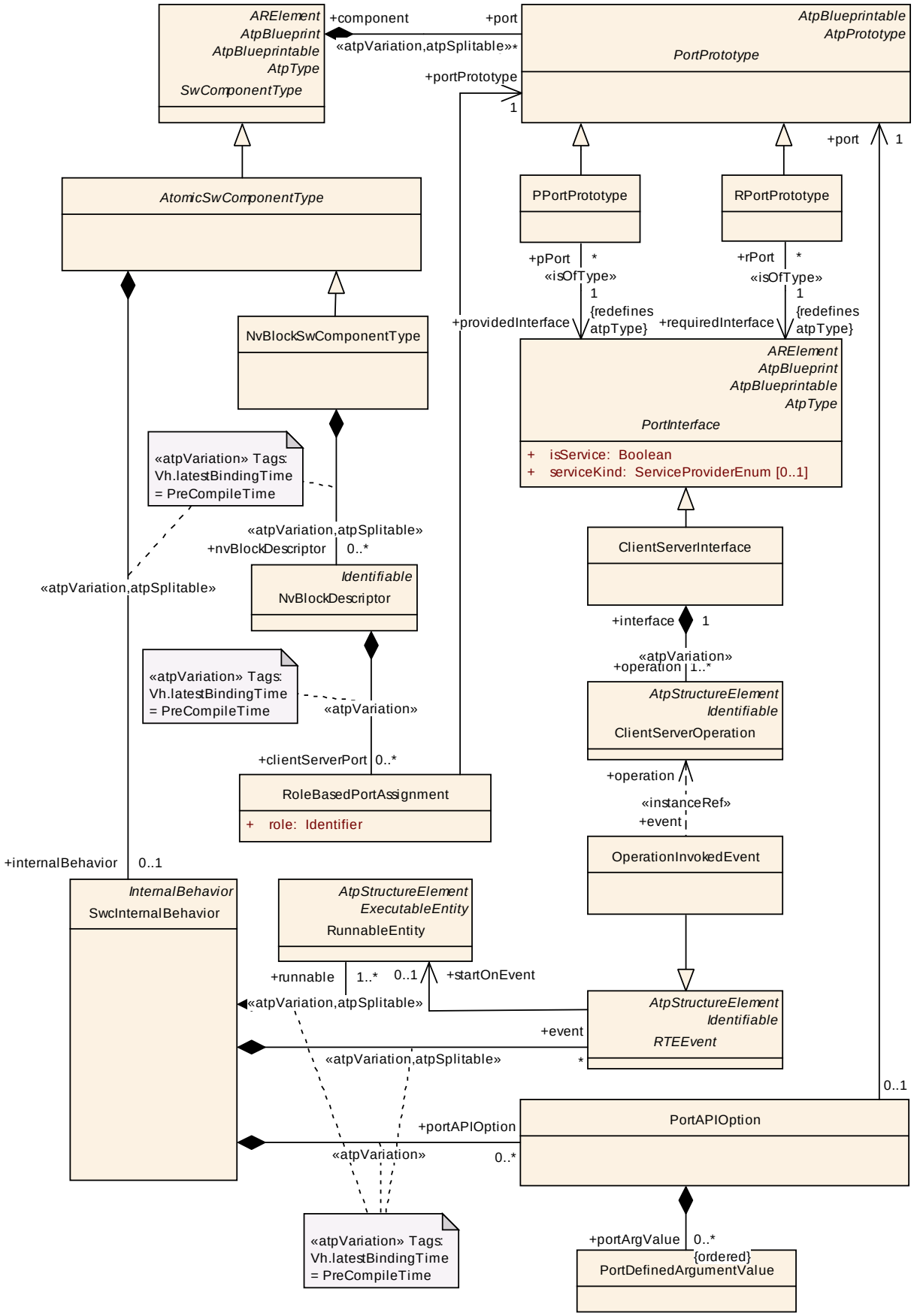


Figure 4.32: `SwcInternalBehavior` of `NvBlockSwComponentTypes`

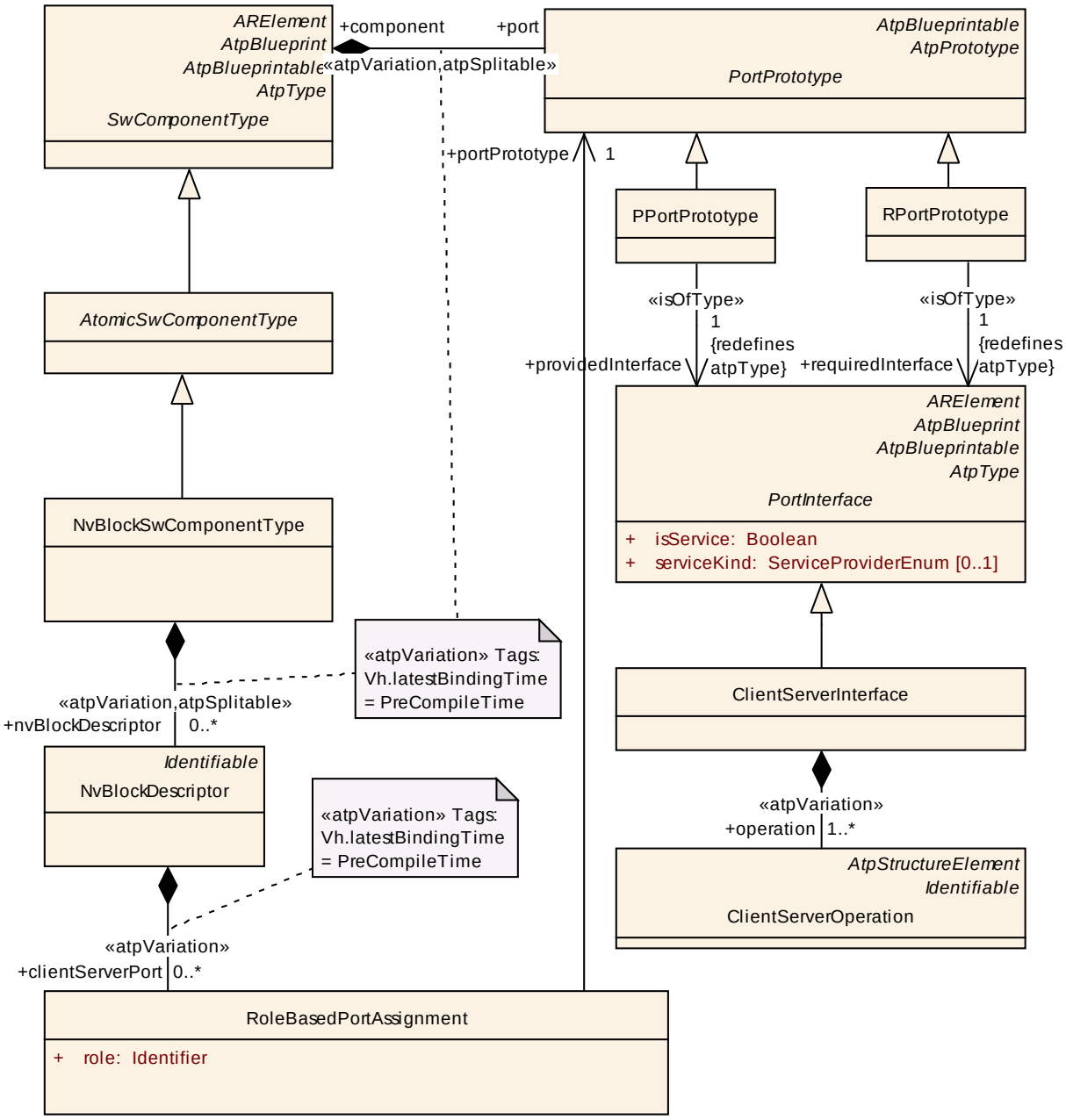


Figure 4.33: NVM notifications

The requests received from the SW-C side are forwarded by the `NvBlockSwComponentType`'s runnables to the NVM module, using the NVM C API indicated by the `RoleBasedPortAssignment`. See figure 4.32.

Notifications received from the NVM are forwarded to all the SW-C connected to the notification interfaces of the NvBlockSwComponentType with a RoleBasedPortAssignment of the corresponding type. See figure 4.33.

[rte_sws_7398]⌈ The RTE Generator shall implement runnables for each connected server port of a NvBlockSwComponentType. ⌋*(RTE00177)*

[rte_sws_7399]⌈ The NvBlockSwComponentType's runnables used as servers connected to the SW-C shall forward the request to the NVM by calling the associated NVM API. ⌋*(RTE00177)*

Note: A BlockId PortDefinedArgumentValue is also provided to runnables and used as a first argument in the NVM APIs.

4.2.9.4 Data Consistency

A VariableDataPrototype contained in a NvBlockSwComponentType is accessed when SW-Cs read the value or write a new value. It is also accessed by the NVM when read or write requests are processed by the NVM for the associated block.

The NVM does not access directly the VariableDataPrototypes, but shall use the Rte_GetMirror, and Rte_SetMirror APIs specified in section 5.9.4

The RTE has to ensure the data consistency of the VariableDataPrototypes, with any of the data consistency mechanisms defined in section 4.2.5. Depending on the user's input, an efficient scheduling with the use of implicit APIs should permit a low resources (OS resources, RAM, and code) implementation.

4.3 Communication Paradigms

AUTOSAR supports two basic communication paradigms: Client-Server and Sender-Receiver. AUTOSAR software-components communicate through well defined ports and the behavior is statically defined by attributes. Some attributes are defined on the modeling level and others are closely related to the network topology and must be defined on the implementation level.

The RTE provides the implementation of these communication paradigms. For inter-ECU communication the RTE uses the functionalities provided by COM. For inter-Partition communication (within the same ECU) the RTE uses functionalities provided by the IOC module. For intra-Partition the RTE provides the functionality on its own.

With Sender-Receiver communication there are two main principles: Data Distribution and Event Distribution. When data is distributed, the last received value is of interest (last-is-best semantics). When events are distributed the whole history of received events is of interest, hence they must be queued on receiver side. Therefore the software implementation policy can be queued or non queued. This is stated in the swImplPolicy attribute of the SwDataDefProps, which can have the value 'queued'

(corresponding to event distribution with a queue) or 'standard' (corresponding to last-is-best data distribution). If a data element has event semantics, the `swImplPolicy` is set to 'queued'. The other possible values of this attribute correspond to data semantics.

[rte_sws_7192]⌈ The RTE generator shall reject the configuration when an `r-port` is connected to an r-port or a `p-port` is connected to a `p-port` with an `AssemblySwConnector`⌋*(RTE00018)*

For example, a require port (`r-port`) of a component typed by an AUTOSAR sender-receiver interface can read data elements of this interface. A provide port (`p-port`) of a component typed by an AUTOSAR sender-receiver interface can write data elements of this interface.

[rte_sws_7006]⌈ The RTE generator shall reject the configuration when an `r-port` is connected to a `p-port` or a `p-port` is connected to an `r-port` with a `DelegationSwConnector`. ⌋*(RTE00018)*

4.3.1 Sender-Receiver

4.3.1.1 Introduction

Sender-receiver communication involves the transmission and reception of signals consisting of atomic data elements that are sent by one component and received by one or more components. A sender-receiver interface can contain multiple data elements. Sender-receiver communication is one-way - any reply sent by the receiver is sent as a separate sender-receiver communication.

A require port (r-port) of a component typed by an AUTOSAR sender-receiver interface can read data elements of this interface. A provide port (p-port) of a component typed by an AUTOSAR sender-receiver interface can write data elements of this interface.

4.3.1.2 Receive Modes

The RTE supports multiple receive modes for passing data to receivers. The four possible receive modes are:

- **"Implicit data read access"** – when the receiver's runnable executes it shall have access to a "copy" of the data that remains unchanged during the execution of the runnable.

  [rte_sws_6000]⌈ For data elements specified with implicit data read access, the RTE shall make the receive data available to the runnable through the `semantics of a copy`. ⌋*(RTE00128, RTE00019)*

  [rte_sws_6001]⌈ For data elements specified with implicit data read access the receive data shall not change during execution of the runnable. ⌋*(RTE00128)*

When “implicit data read access” is used the RTE is required to make the data available as a “copy”. It is not necessarily required to use a unique copy for each runnable. Thus the RTE may use a unique copy of the data for each runnable entity or may, if several runnables (even from different components) need the same data, share the same copy between runnables. Runnable entities can only share a copy of the same data when the scheduling structure can make sure the contents of the data is protected from modification by any other party.

[rte_sws_6004]⌈ The RTE shall read the data elements specified with implicit data read access before the associated runnable entity is invoked. ⌋*(RTE00128)*

Composite data types shall be handled in the same way as primitive data types, i.e. RTE shall make a “copy” available for the `RunnableEntity`.

[rte_sws_6003]⌈ The “implicit data read access” receive mode shall be valid for all categories of runnable entity (i.e. 1A, 1B and 2). ⌋*(RTE00134)*

- **“Explicit data read access”** – the RTE generator creates a non-blocking API call to enable a receiver to poll (and read) data. This receive mode is an “explicit” mode since an explicit API call is invoked by the receiver.

  The explicit “data read access” receive mode is only valid for category 1B or 2 runnable entities [RTE00134].

- **“wake up of wait point”** – the RTE generator creates a blocking API call that the receiver invokes to read data.

  [rte_sws_6002]⌈ The “wake up of wait point” receive mode shall support a time-out to prevent infinite blocking if no data is available. ⌋*(RTE00109, RTE00069)*

  The “wake up of wait point” receive mode is inherently only valid for a category 2 runnable entity.

  A category 2 runnable entity is required since the implementation may need to suspend execution of the caller if no data is available.

- **“activation of runnable entity”** – the receiving runnable entity is invoked automatically by the RTE whenever new data is available. To access the new data, the runnable entity either has to use “implicit data read access” or “explicit data read access”, i.e. invoke an `Rte_IRead`, `Rte_Read`, `Rte_DRead` or `Rte_Receive` call, depending on the input configuration. This receive mode differs from “implicit data read access” since the receiver is invoked by the RTE in response to a `DataReceivedEvent`.

  [rte_sws_6007]⌈ The “activation of runnable entity” receive mode shall be valid for category 1A, 1B and 2 runnable entities. ⌋*(RTE00134)*

The validity of receive modes in conjunction with different categories of runnable entity is summarized in Table 4.9.

Receive Mode	Cat 1A	Cat 1B	Cat 2
Implicit Data Read Access	Yes	Yes	Yes

Explicit Data Read Access	No	Yes	Yes
Wake up of wait point	No	No	Yes
Activation of runnable entity	Yes	Yes	Yes

Table 4.9: Receive mode validity

The category of a runnable entity is not an inherent property but is instead determined by the features of the runnable. Thus the presence of explicit API calls makes the runnable at least category 1B and the presence of a `WaitPoint` forces the runnable to be category 2.

4.3.1.2.1 Applicability

The different receive modes are not just used for receivers in sender-receiver communication. The same semantics are also applied in the following situations:

- **Success feedback** – The mechanism used to return transmission acknowledgments to a component. See Section 5.2.6.9.
- **Asynchronous client-server result** – The mechanism used to return the result of an asynchronous client-server call to a component. See Section 5.7.5.4.

4.3.1.2.2 Representation in the Software Component Template

The following list serves as a reference for how the RTE Generator determines the Receive Mode from its input [RTE00109]. Note that references to "the `VariableDataPrototype`" within this sub-section will implicitly mean "the `VariableDataPrototype` for which the API is being generated".

- **"wake up of wait point"** – A `VariableAccess` in the `dataReceivePointByValue` or `dataReceivePointByArgument` role references a `VariableDataPrototype` and a `WaitPoint` references a `DataReceivedEvent` which in turn references the same `VariableDataPrototype`.
- **"activation of runnable entity"** – a *DataReceivedEvent* references the `VariableDataPrototype` and a runnable entity to start when the data is received.
- **"explicit data read access"** – A `VariableAccess` in the `dataReceivePointByValue` or `dataReceivePointByArgument` role references the `VariableDataPrototype`.
- **"implicit data read access"** – A `VariableAccess` in the `dataReadAccess` role references the `VariableDataPrototype`.

It is possible to combine certain access methods; for example 'activation of runnable entity' can be combined with 'explicit' or 'implicit' data read access (indeed, one of these

pairings is necessary to cause API generation to actually *read* the datum) but it is an input error if 'activation of runnable entity' and 'wakeup of wait point' are combined (i.e. a `WaitPoint` references a `DataReceivedEvent` that references a runnable entity). It is also possible to specify both implicit and explicit data read access simultaneously.

For details of the semantics of "implicit data read access" and "explicit data read access" see Section 4.3.1.5.

4.3.1.3 Multiple Data Elements

A sender-receiver interface can contain one or more data elements. The transmission and reception of elements is independent – each data element, e.g. AUTOSAR signal, can be considered to form a separate logical data channel between the "provide" port and a "require" port.

[rte_sws_6008]⌈ Each data element in a sender-receiver interface shall be sent separately. ⌋*(RTE00089)*

Example 4.4

Consider an interface that has two data elements, `speed` and `freq` and that a component template defines a provide port that is typed by the interface. The RTE generator will then create two API calls; one to transmit `speed` and another to transmit `freq`.

Where it is important that multiple data elements are sent simultaneously they should be combined into a composite data structure (Section 4.3.1.11.1). The sender then creates an instance of the data structure which is filled with the required data before the RTE is invoked to transmit the data.

4.3.1.3.1 Initial Values

[rte_sws_6009]⌈ For each data element in an interface specified with data semantics, the RTE shall support the `initValue` attribute. ⌋*(RTE00108)*

The `initValue` attribute is used to ensure that AUTOSAR software-components always access valid data even if no value has yet been received. This information is required for inter-ECU, inter-Partition, and intra-Partition communication. For inter-ECU communication initial values can be handled by COM but for intra-ECU communication RTE has to guarantee that `initValue` is handled.

In general, the specification of an `initValue` is mandatory for each data element prototype with data semantics, see rte_sws_7642. If all senders and receivers are located in the same partition, this restriction is relaxed, see rte_sws_4501.

[rte_sws_6010]⌈ The RTE shall use any specified initial value to prevent the receiver performing calculations based on invalid (i.e. uninitialized) values when the `swIm-`

plPolicy is not queued and if the general initialization conditions in rte_sws_7046 are fulfilled. ⌋*(RTE00107)*

The above requirement ensures that RTE API calls return the initialized value until a "real" value has been received, possibly via the communication service. The requirement does *not* apply when "event" semantics are used since the implied state change when the event data is received will mean that the receiver will not start to process invalid data and would therefore never see the initialized value.

[rte_sws_4500]⌈ An initial value cannot be specified when the implementation policy is set to 'queued' attribute is specified as true. ⌋*(RTE00107)*

For senders, an initial value is not used directly by the RTE (since an AUTOSAR SW-C must supply a value using Rte_Send) however it may be needed to configure the communication service - for example, an un-initialised signal can be transmitted if multiple signals are mapped to a single frame and the communication service transmits the whole frame when any contained signal is sent by the application. Note that it is not the responsibility of the RTE generator to configure the communication service.

It is permitted for an initial value to be specified for either the sender or receiver. In this case the same value is used for both sides of the communication.

[rte_sws_4501]⌈ If in context of one partition a sender specifies an initial value and the receiver does not (or *vice versa*) the same initial value is used for both sides of the communication. ⌋*(RTE00108)*

It is also permitted for both sender and receiver to specify an initial value. In this case it is defined that the receiver's initial value is used by the RTE generator for both sides of the communication.

[rte_sws_4502]⌈ If in context of one partition both receiver and sender specify an initial value the specification for the *receiver* takes priority. ⌋*(RTE00108)*

4.3.1.4 Multiple Receivers and Senders

Sender-receiver communication is not restricted to communication connections between a single sender and a single receiver. Instead, sender receiver communication connection can have multiple senders ('n:1' communication) or multiple receivers ('1:m' communication) with the restrictions that multiple senders are not allowed for mode switch notifications, see metamodel restriction rte_sws_2670.

The RTE does not impose any co-ordination on senders – the behavior of senders is independent of the behavior of other senders. For example, consider two senders A and B that both transmit data to the same receiver (i.e. 'n:1' communication). Transmissions by either sender can be made at any time and there is no requirement that the senders co-ordinate their transmission. However, while the RTE does not impose any co-ordination on the senders it does ensure that simultaneous transmissions do not conflict.

In the same way that the RTE does not impose any co-ordination on senders there is no co-ordination imposed on receivers. For example, consider two receivers P and Q that both receive the same data transmitted by a single sender (i.e. '1:m' communication). The RTE does not guarantee that multiple receivers see the data simultaneously even when all receivers are on the same ECU.

4.3.1.5 Implicit and Explicit Data Reception and Transmission

[rte_sws_6011]⌈ The RTE shall support 'explicit' and 'implicit' data reception and transmission. ⌋*(RTE00019, RTE00098, RTE00129, RTE00128, RTE00141)*

Implicit data access transmission means that a runnable does not actively initiate the reception or transmission of data. Instead, the required data is received automatically when the runnable starts and is made available for other runnables at the earliest when it terminates.

Explicit data reception and transmission means that a runnable employs an explicit API call to send or receive certain data elements. Depending on the category of the runnable and on the configuration of the according ports, these API calls can be either blocking or non-blocking.

4.3.1.5.1 Implicit

Implicit Read

For the implicit reading of data, VariableAccesses aggregated with a dataReadAccess role [RTE00128], the data is made available when the runnable starts using the semantics of a copy operation and the RTE ensures that the 'copy' will not be modified until after the runnable terminates.

When a runnable *R* is started, the RTE reads all VariableDataPrototypes referenced by a VariableAccess in the dataReadAccess role, if the data elements may be changed by other runnables a copy is created that will be available to runnable *R*. The runnable *R* can read the data element by using the RTE APIs for implicit read (see the API description in Section 5.6.18). That way, the data is guaranteed not to change (e.g. by write operations of other runnables) during the entire lifetime of *R*. If several runnables (even from different components) need the data, they can share the *same* buffer. This is only applicable when the scheduling structure can make sure the contents of the data is protected from modification by any other party.

Note that this concept implies that the runnable does in fact terminate. Therefore, while implicit read is allowed for category 1A and 1B runnable entities as well as category 2 only the former are guaranteed to have a finite execution time. A category 2 runnable that runs forever will not see any updated data.

VariableAccess in the dataReadAccess role is only allowed for VariableDataPrototypes with their swImplPolicy different from 'queued' (rte_sws_3012).

Implicit Write

Implicit writing, VariableAccesses aggregated with a dataWriteAccess role [RTE00129], is the opposite concept. VariableDataPrototypes referenced by a VariableAccess in the dataWriteAccess role are sent by the RTE after the runnable terminates. The runnable can write the data element by using the RTE APIs for implicit write (see the API description in Sect. 5.6.19 and 5.6.20). The sending is independent from the position in the execution flow in which the Rte_IWrite is performed inside the Runnable. When performing several write accesses during runnable execution to the same data element, only the last one will be recognized. Here we have a last-is-best semantics.

Note:
If a VariableDataPrototype is referenced by a VariableAccess in the dataWriteAccess role, but no RTE API for implicit write of this VariableDataPrototype is called during an execution of the runnable, an undefined value is written back when the runnable terminates.

[rte_sws_3570]⌈ For VariableAccesses in the dataWriteAccess role the RTE shall make the sent data available to others (other runnables, other AUTOSAR SWCs, Basic SW, ..) with the semantics of a copy. ⌋*(RTE00129)*

[rte_sws_3571]⌈ For VariableAccesses in the dataWriteAccess role the RTE shall make the sent data available to others (other runnables, other AUTOSAR SWCs, Basic SW, ..) at the earliest when the runnable returns (exits the 'Running' state). ⌋*(RTE00129)*

[rte_sws_3572]⌈ For VariableAccesses in the dataWriteAccess role several accesses to the same VariableDataPrototype performed inside a runnable during one runnable execution shall lead to only one transmission of the VariableDataPrototype. ⌋*(RTE00129)*

[rte_sws_3573]⌈ If several VariableAccesses in the dataWriteAccess role referencing the same VariableDataPrototype are performed inside a runnable during the runnable execution, the RTE shall use the last value written. (last-is-best semantics) ⌋*(RTE00129)*

A VariableAccess in the dataWriteAccess role is only sensible for runnable entities that are guaranteed to terminate, i.e. category 1A and 1B. If it is used for a category 2 runnable which does not terminate then no data write-back will occur.

[rte_sws_3574]⌈ VariableAccess in the dataWriteAccess role shall be valid for all categories of runnable entity. ⌋*(RTE00129, RTE00134)*

To get common behavior in RTEs from different suppliers further requirements defining the semantic of implicit communication exist:

Please note that the behavior of Implicit Communication can be adjusted with ECU Configuration. For further information see section 7.8.

Implicit Communication Behavior in case of Incoherent Implicit Data Access

[rte_sws_3954]⌈ The RTE generator shall use exactly one buffer to contain data copies of the same `VariableDataPrototype` per `Preemption Area` for the implementation of the `copy semantic` of `Incoherent Implicit Data Access`. ⌋*(RTE00128, RTE00129, RTE00134)*

Requirement rte_sws_3954 means that all runnable entities mapped to tasks of a `Preemption Area` with a `Incoherent Implicit Read Access` or `Incoherent Implicit Write Access` access the same buffers.

[rte_sws_3598]⌈ For implicit communication, a single shared read/write buffer shall be used when no runnable entity mapped to tasks of the `Preemption Area` has `Incoherent Implicit Read Access` and `Incoherent Implicit Write Access` referencing the same `VariableDataPrototype`. ⌋*(RTE00128, RTE00129)*

If either the sender or the receiver uses a `Data Element with Status` and the other uses a `Data Element without Status`, a `Data Element with Status` can be implemented and casted in the component data structure when a pointer to a `Data Element without Status` is needed.

[rte_sws_3955]⌈ For implicit communication, separate read and write buffers shall be used when at least one runnable entity mapped to tasks of the `Preemption Area` has `Implicit Read Access` and `Implicit Write Access` referencing the same `VariableDataPrototype`. ⌋*(RTE00128, RTE00129)*

Please note that the content of the write buffers are copied into the read buffer of the `Preemption Area` after the `RunnableEntity` with the write access terminates (see rte_sws_7041). Therefore the write buffer might be implemented as temporary buffer.

[rte_sws_3599]⌈ For implicit communication with `Incoherent Implicit Data Access` all readers within a `Preemption Area` shall access the same buffer. ⌋*(RTE00128)*

[rte_sws_3953]⌈ For implicit communication with `Incoherent Implicit Data Access` all writers within a `Preemption Area` shall access the same buffer. ⌋*(RTE00129)*

The content of a shared buffer (see rte_sws_3598) is not guaranteed to stay constant during the whole task since a writer will change the shared copy and hence readers mapped in the task after the writer will access the updated copy. When buffers are shared, written data is visible to other `RunnableEntity`s within the same execution of the task. However since no runnable within the task will both read and write the same buffer (rte_sws_3598 and rte_sws_3955) consistency *within a runnable* is ensured.

When separate buffers used for implicit communication (see rte_sws_3955) any data written by a runnable is not visible (to either other `RunnableEntity`s or to the writing runnable) until the data is written back after the runnable has terminated.

Implicit Communication Behavior in case of Coherent Implicit Data Access

[rte_sws_7062]⌈ The RTE generator shall use exactly one buffer to contain data copies of the same `VariableDataPrototype` per `Coherency Group` for the implementation of the `copy semantic` of `Coherent Implicit Data Access`. ⌋*(RTE00128, RTE00129, RTE00134)*

Requirement rte_sws_7062 means that all runnable entities with `Coherent Implicit Data Accesses` access the same buffers. Please note that it is only supported to group `Implicit Read Accesses` or `Implicit Write Accesses` of `RunnableEntity`s executed in the same OS Task. Therefore a `Coherent Implicit Data Access` results in a task local buffer as it was specified in previous AUTOSAR releases. With this means a backward compatible bahavior of the RTE can be ensured.

Please note that rte_sws_3955 applies as well for Coherent Implicit Data Access. rte_sws_7062 includes already that a single shared read/write buffer shall be used when no runnable entity has `Coherent Implicit Read Access` and `Coherent Implicit Write Access` belonging to the same `Coherency Group`.

Implicit Communication buffer handling

[rte_sws_3956]⌈ The content of a `Preemption Area` specific buffer used for an `Incoherent Implicit Read Access` to a `VariableDataElement` shall be filled with actual data by a copy action between the beginning of the task and the execution of the first `RunnableEntity` with access to this `VariableDataElement` in the task. ⌋*(RTE00128)*

[rte_sws_7687]⌈ There should not be more update of the `Preemption Area` specific buffer within one task than required. ⌋*(RTE00128)*

[rte_sws_7020]⌈ If the `RteImmediateBufferUpdate = TRUE` is configured for a `Incoherent Implicit Read Access` to a `VariableDataElement` the content of a `Preemption Area` specific buffer used for that `VariableAccess` shall be filled with actual data by a copy action immediately before the `RunnableEntity` with the related implicit read access to the `VariableDataElement` starts. ⌋*(RTE00128)*

[rte_sws_7041]⌈ The content of a separate write buffer (see rte_sws_3955) modified by a `Incoherent Implicit Write Access` of a `RunnableEntity` shall be made available to `RunnableEntity`s using a `Implicit Read Access` allocated in the **same** `Preemption Area` immediately after the execution of the `RunnableEntity` with the related `Implicit Write Access` to the `VariableDataElement`. ⌋*(RTE00129)*

[rte_sws_3957]⌈ The content of a `Preemption Area` specific buffer modified by a `Incoherent Implicit Write Access` in one task shall be made available to `RunnableEntity`s using an `Implicit Read Access` allocated in **other** `Preemp-`

tion Areas at latest after the execution of the last RunnableEntity mapped to the task. ⌋*(RTE00129)*

[rte_sws_7688]⌈ The Preemption Area specific buffer should not be made available more often than required. ⌋*(RTE00129)*

[rte_sws_7021]⌈ If the RteImmediateBufferUpdate = TRUE is configured for a Incoherent Implicit Write Access the content of a Preemption Area specific buffer shall be made available to RunnableEntitys using a Implicit Read Access allocated in **other** Preemption Areas immediately after the execution of the RunnableEntity with the related Implicit Write Access to the VariableDataElement. ⌋*(RTE00129)*

Note:
It's the semantic of implicit communication that a VariableAccess in the dataWriteAccess role is interpreted as writing the whole dataElement.

Explicit Schedule Points defined by RteOsSchedulePoints are placed between RunnableEntities after the data written with implicit write access by the RunnableEntity are propagated to other RunnableEntitys and before the Preemption Area specific buffer used for a implicit read access of the successor RunnableEntity are filled with actual data by a copy action according rte_sws_7020. This ensures that the data produced by one RunnableEntity is propagated before RunnableEntitys assigned to other Os Tasks are activated due to Task scheduling caused by the explicit Schedule Point. See as well rte_sws_7042 and rte_sws_7043.

Implicit Communication buffer handling for Coherent Implicit Data Access

[rte_sws_7063]⌈ The content of a Coherency Group specific buffer used for an Coherent Implicit Read Access to one or more VariableDataElements shall be filled with actual data by a copy action between the beginning of the task and the execution of the first RunnableEntity in the task with a Coherent Implicit Read Access belonging to the Coherency Group. ⌋*(RTE00128)*

[rte_sws_7064]⌈ If the RteImmediateBufferUpdate = TRUE is configured for Coherent Implicit Read Accesses the content of a Coherency Group specific buffer used for these VariableAccesses shall be filled with actual data by a copy action immediately before the first RunnableEntity in the task with a Coherent Implicit Read Access belonging to the Coherency Group starts. ⌋*(RTE00128)*

[rte_sws_7065]⌈ The content of a separate write buffer (see rte_sws_3955) modified by a Coherent Implicit Write Access of a RunnableEntity shall be made available to RunnableEntitys using a Coherent Implicit Read Access belonging to the same Coherency Group immediately after the execution of the RunnableEntity with the related Coherent Implicit Write Access. ⌋*(RTE00129)*

[rte_sws_7066]⌈ The content of a Coherency Group specific buffer modified by Coherent Implicit Write Accesses in one task shall be made available to other RunnableEntitys at earliest after the execution of the last RunnableEntity

with a Coherent Implicit Write Access belonging to this Coherency Group. ⌋*(RTE00129)*

[rte_sws_7067]⌈ The content of a Coherency Group specific buffer modified by Coherent Implicit Write Accesses in one task shall be made available to other RunnableEntitys at latest after the execution of the last RunnableEntity mapped to the task. ⌋*(RTE00129)*

[rte_sws_7068]⌈ If the RteImmediateBufferUpdate = TRUE is configured for a Coherent Implicit Write Accesses the content of a Coherency Group specific buffer modified by Coherent Implicit Write Accesses in one task shall be made available to other readers not belonging to this Coherency Group immediately after the execution of the last RunnableEntity with a Coherent Implicit Write Access belonging to this Coherency Group ⌋*(RTE00129)*

4.3.1.5.2 Explicit

The behavior of explicit reception depends on the category of the runnable and on the configuration of the according ports.

An explicit API call can be either non-blocking or blocking. If the call is non-blocking (i.e. there is a VariableAccess in the dataReceivePointByValue or dataReceivePointByArgument role referencing the VariableDataPrototype for which the API is being generated, but no WaitPoint referencing a DataReceivedEvent which references the VariableDataPrototype for which the API is being generated), the API call immediately returns the next value to be read and, if the communication is queued (event reception), it removes the data from the receiver-side queue, see Section 4.3.1.10

[rte_sws_6012]⌈ A non-blocking RTE API “read” call shall indicate if no data is available. ⌋*(RTE00109)*

In contrast, a blocking call (i.e. the VariableDataPrototype, referenced by a VariableAccess in the role dataReceivePointByArgument, and for which the API is being generated, is referenced by a DataReceivedEvent which is itself referenced by a WaitPoint) will suspend execution of the caller until new data arrives (or a timeout occurs) at the according port. When new data is received, the RTE resumes the execution of the waiting runnable. (RTE00092)

To prevent infinite waiting, a blocking RTE API call can have a timeout applied. The RTE monitors the timeout and if it expires without data being received returns a particular error status.

[rte_sws_6013]⌈ A blocking RTE API “read” call shall indicate the expiry of a timeout. ⌋*(RTE00069)*

The “timeout expired” indication also indicates that no data was received before the timeout expired.

Blocking reception of data (“wake up of wait point” receive mode as described in Section 4.3.1.2) is only applicable for category 2 runnables whereas non-blocking reception (“explicit data read access” receive mode) can be employed by runnables of category 2 or 1B. Neither blocking nor non-blocking explicit reception is applicable for category 1A runnable because they must not invoke functions with unknown execution time (see table 4.9).

[rte_sws_6016]⌈ The RTE API call for explicit sending (`VariableAccess`in the `dataSendPoint` role, [RTE00098]) shall be non-blocking. ⌋*(RTE00098)*

Using this API call, the runnable can explicitly send new values of the `VariableDataPrototype`.

Explicit writing is valid for runnables of category 1b and 2 only. Explicit writing is not allowed for a category 1A runnable since these require API calls with constant execution time (i.e. macros).

Although the API call for explicit sending is non-blocking, it is possible for a category 2 runnable to block waiting for a notification whether the (explicit) send operation was successful. This is specified by the AcknowledgementRequest attribute and occurs by a separate API call `Rte_Feedback`. If the feedback method is ’wake_up_of_wait_point’, the runnable will block and be resumed by the RTE either when a positive or negative acknowledgment arrives or when the timeout associated with the `WaitPoint` expires.

4.3.1.5.3 Concepts of data access

Tables 4.10 and 4.11 summarize the characteristics of implicit versus explicit data reception and transmission.

Implicit Read	Explicit Read
Receiving of data element values is performed only once when runnable starts	Runnable decides when and how often a data element value is received
Values of data elements do not change while runnable is running.	Runnable can always decide to receive the latest value
Several API calls to the same signal always yield the same data element value	Several API calls to the same signal may yield different data element values
Runnable must terminate (all categories)	Runnable is of cat. 1B or 2

Table 4.10: Implicit vs. explicit read

Implicit Write	Explicit Write
Sending of data element values is only done once after runnable returns	Runnable can decide when sending of data element values is done via the API call
Several usages of the API call inside the runnable cause only one data element transmission	Several usages of the API call inside the runnable cause several transmissions of the data element content. (Depending on the behavior of COM, the number of API calls and the number of transmissions are not necessarily equal.)
Runnable must terminate (all categories)	Runnable is cat. 1B or 2

Table 4.11: Implicit vs. explicit write

4.3.1.6 Transmission Acknowledgement

When `TransmissionAcknowledgementRequest` is specified, the RTE will inform the sending component if the signal has been sent correctly or not. Note that there is no insurance that the signal has actually been *received* correctly by the corresponding

receiver AUTOSAR software-component. Thus, only the RTE on the sender side is involved in supporting TransmissionAcknowledgementRequest.

[rte_sws_5504]⌈ The RTE shall support the use of TransmissionAcknowledgementRequest independently for each data item of an AUTOSAR software-component's AUTOSAR interface. ⌋*(RTE00122)*

[rte_sws_5506]⌈ The RTE generator shall reject specification of the TransmissionAcknowledgementRequest attribute for transmission acknowledgment for 1:n communication.Restriction: In some cases, when more than one receiver is connected via one physical bus, this can not be discovered by the RTE generator. ⌋*(RTE00125, RTE00018)*

The result of the feedback can be collected using "wake up of wait point", "explicit data read access", "implicit data read access" or "activation of runnable entity".

The TransmissionAcknowledgementRequest allows to specify a timeout.

[rte_sws_3754]⌈ If TransmissionAcknowledgementRequest is specified, the RTE shall ensure that timeout monitoring is performed, regardless of the receive mode of the acknowledgment. ⌋*(RTE00069, RTE00122)*

For inter-ECU communication, AUTOSAR COM provides the necessary functionality, for intra-ECU communication, the RTE has to implement the timeout monitoring.

If a WaitPoint is specified to collect the acknowledgment, two timeout values have to be specified, one for the TransmissionAcknowledgementRequest and one for the WaitPoint.

[rte_sws_3755]⌈ If different timeout values are specified for the TransmissionAcknowledgementRequest for a VariableDataPrototype and for the WaitPoint associated with the DataSendCompletedEvent for the VariableAccess in the dataSendPoint role for that VariableDataPrototype, the configuration shall be rejected by the RTE generator. ⌋*(RTE00018)*

The DataSendCompletedEvent associated with the VariableAccess in the dataSendPoint role for a VariableDataPrototype shall indicate that the transmission was successful or that the transmission was not successful. The status information about the success of the transmission shall be available as the return value of the generated RTE API call.

[rte_sws_3756]⌈ For each transmission of a VariableDataPrototype only one acknowledgment shall be passed to the sending component by the RTE. The acknowl-

edgment indicates either that the transmission was successful or that the transmission was not successful. ⌋*(RTE00122)*

[rte_sws_3757]⌈ The status information about the success or failure of the transmission shall be available as the return value of the RTE API call to retrieve the acknowledgment. ⌋*(RTE00122)*

[rte_sws_3604]⌈ The status information about the success or failure of the transmission shall be buffered with last-is-best semantics. When a data item is sent, the status information is reset. ⌋*(RTE00122)*

rte_sws_3604 implies that once the `DataSendCompletedEvent` has occurred, repeated API calls to retrieve the acknowledgment shall always return the same result until the next data item is sent.

[rte_sws_3758]⌈ If the timeout value of the `TransmissionAcknowledgementRequest` is 0, no timeout monitoring shall be performed. ⌋*(RTE00069, RTE00122)*

4.3.1.7 Communication Time-out

When sender-receiver communication is performed using some physical network there is a chance this communication may fail and the receiver does not get an update of data (in time or at all). To allow the receiver of a `data element` to react appropriately to such a condition the SW-C template allows the specification of a time-out which the infrastructure shall monitor and indicate to the interested software components.

A “data element” is the actual information exchanged in case of sender-receiver communication. In the COM specification this is represented by a `ComSignal`. In the SW-C template a data element is represented by the instance of a `VariableDataPrototype`.

[rte_sws_5020]⌈ When present, the `aliveTimeout` attribute[5] enables the monitoring of the timely reception of the `data element` with data semantics transmitted over the network. ⌋*(RTE00147)*

The monitoring functionality is provided by the COM module, the RTE transports the event of reception time-outs to software components as “data element outdated”. The software components can either subscribe to that event (activation of runnable entity) or get that situation passed by the implicit and explicit status information (using API calls).

[rte_sws_5021]⌈ If communication is local to the partition, time-out monitor will be disabled and no notification of the software components will occur, independently of presence of `aliveTimeout`. ⌋*(RTE00147)*

[5]This attribute is called “LIVELIHOOD” in the VFB specification

Therefore the Software Component shall not rely in its functionality on the time-out notification, because for local communication the notification will never occur. Time-out notification is intended as pure error reporting.

[rte_sws_2710]⌈ If `aliveTimeout` is present, and the communication is between different partitions of the same ECU, time-out monitoring is disabled. Instead, a timeout notification of the receiver will occur immediately, when the partition of the sender is stopped and the last correctly received value shall be provided to the software components. ⌋*()*

Therefore the Software Component shall not rely in its functionality on the time-out notification, because for local communication the notification will never occur. Time-out notification is intended as pure error reporting.

[rte_sws_3759]⌈ If the `aliveTimeout` attribute is 0, no timeout monitoring shall be performed. ⌋*(RTE00069, RTE00147)*

[rte_sws_5022]⌈ If a time-out has been detected in inter ECU communication, the value provided from COM shall be provided to the software components. ⌋*(RTE00147)*

[rte_sws_8004]⌈ If a signal is received, even if the signal is marked as invalid, the time-out for the same signal shall be restarted. ⌋*(RTE00078, RTE00147)*

Note: time-out detection may already be implemented by COM. Nevertheless this is the expected behavior towards the software components.

The time-out support (called “deadline monitoring” in COM) provided by COM has some restrictions which have to be respected when using this mechanism. Since the COM module is configured based on the System Description the restrictions mainly arise from the `data element` to I-PDU mapping. This already has to be considered when developing the System Description and the RTE Generator can only provide warnings when inconsistencies are detected. Therefore the RTE Generator needs to have access to the configuration information of COM.

In case time-out is enabled on a `data element` with update bit, there shall be a separate time-out monitoring for each `data element` with an update bit [COM292].

There shall be an I-PDU based time-out for `data elements` without an update bit [COM290]. For all data elements without update bits within the same I-PDU, the smallest configured time-out of the associated data elements is chosen as time-out for the I-PDU[COM291]. The notification from COM to RTE is performed per data element.

In case one `data element` coming from COM needs to be distributed to several AUTOSAR software-components the AUTOSAR Software Component Template allows to specify different `aliveTimeout` values at each Port. But COM does only support one `aliveTimeout` value per `data element`, therefore the smallest `aliveTimeout` value shall be used for the notification of the time-out to several software-components.

4.3.1.8 Data Element Invalidation

The Software Component template allows to specify whether a `data element`, defined in an AUTOSAR Interface, can be invalidated by the sender. The communication infrastructure shall provide means to set a data element to invalid and also indicate an invalid data element to the receiving software components. This functionality is called "data element invalidation". For an overview see figure 4.43.

[rte_sws_5024]⌈ If the `handleInvalid` attribute of the `InvalidationPolicy` (when present) is set to `keep` or `replace` the invalidation support for this `dataElement` is enabled on sender side. The actual value used to represent the invalid data element shall be specified in the Data Semantics part of the data element definition defined in `invalidValue`[6]. ⌋*(RTE00078)*

[rte_sws_5032]⌈ On receiver side the `handleInvalid` attribute of the associated `InvalidationPolicy` specifies how to handle the reception of the invalid value. ⌋*()*

[rte_sws_5033]⌈ `Data element invalidation` is only supported for data elements with a `swImplPolicy` different from 'queued'. ⌋*()*

The API to set a `dataElement` to invalid shall be provided to the `RunnableEntity`s on data element level.

In case an invalidated data element is received a software component can be notified using the activation of runnable entity. If an invalidated data element is read by the SW-C the invalid status shall be indicated in the status code of the API.

[rte_sws_8005]⌈ If the `initValue` of an unqueued data element equals the `invalidValue` and `handleInvalid` is set to `keep` and the `handleNeverReceived` is set to `FALSE`, the RTE APIs `Rte_Read()` and `Rte_IStatus()` shall return `RTE_E_INVALID` until first reception of data element. In this case the APIs `Rte_Read()` and `Rte_IRead()` shall provide the `invalidValue`. ⌋*()*

[rte_sws_8008]⌈ If the `initValue` of an unqueued data element equals the `invalidValue` and `handleInvalid` is set to `keep` and the `handleNeverReceived` is not defined, the RTE APIs `Rte_Read()` and `Rte_IStatus()` shall return `RTE_E_INVALID` until first reception of data element. In this case the APIs `Rte_Read()` and `Rte_IRead()` shall provide the `invalidValue`. ⌋*(RTE00184)*

[rte_sws_8009]⌈ If the `initValue` of an unqueued data element equals the `invalidValue` and `handleInvalid` is set to `keep` and the `handleNeverReceived` is set to `TRUE`, the RTE APIs `Rte_Read()` and `Rte_IStatus()` shall return `RTE_E_NEVER_RECEIVED` until first reception of data element. In this case the APIs `Rte_Read()` and `Rte_IRead()` shall provide the `initValue`. ⌋*(RTE00184)*

[rte_sws_8007]⌈ The RTE Generator shall reject configurations in which the `initValue` of an unqueued data element equals the `invalidValue` and `handleInvalid` is set to `replace`. ⌋*()*

[6] When `InvalidationPolicy` is set to `keep` or `replace` but there is no `invalidValue` specified it is considered as an invalid configuration.

4.3.1.8.1 Data Element Invalidation in case of Inter-ECU communication

Sender:

If `data element invalidation` is enabled and the communication is Inter-ECU:

- explicit data transmission: data element invalidation will be performed by COM (COM needs to be configured properly).
- implicit data transmission: data element invalidation will be performed by RTE.

Receiver:

If `data element invalidation` is enabled and the communication is Inter-ECU and:

- if all receiving software components requesting the same value for `handleInvalid` attribute of the `InvalidationPolicy` associated to one `dataElement`: data element invalidation will be performed by COM (COM needs to be configured properly), see rte_sws_5026, rte_sws_5048.
- if the receiving software components requesting different values for `handleInvalid` attribute of the `InvalidationPolicy` associated to one `dataElement`: data element invalidation will be performed by RTE , see rte_sws_7031, rte_sws_7032. This can occur in case of 1:n communication where for one connector a `VariableAndParameterInterfaceMapping` is applied to two `SenderReceiverInterface`s with different `InvalidationPolicy`s for the mapped `VariableDataPrototype`.

[rte_sws_5026]⌈ If a data element has been received invalidated in case of Inter-ECU communication and the attribute `handleInvalid` is set to `keep` for all receiving software components – the query of the value shall return the value provided by COM together with an indication of the invalid case. Then the reception of the invalid value will be handled as an error and the activation of runnable entities can be performed using the `DataReceiveErrorEvent`. ⌋*()*

[rte_sws_5048]⌈ If a data element has been received invalidated in case of Inter-ECU communication and the attribute `handleInvalid` is set to `replace` for all receiving software components – COM shall be configured to perform the "invalid value substitution" (`ComDataInvalidAction` is *REPLACE* [COM314]) with the `initValue`. Then the reception will be handled as if a valid value would have been received (activation of runnable entities using the `DataReceivedEvent`). ⌋*()*

[rte_sws_7031]⌈ If a data element has been invalidated in case of Inter-ECU communication where receiving software components requesting different values for `handleInvalid` and the attribute `handleInvalid` is set to `keep` for an particular `r-port` – the query of the value shall return for the `r-port` the same value as if COM would have handled the invalidation (copy COM behavior). Then the reception of the

invalid value will be handled as an error and the activation of runnable entities can be performed using the `DataReceiveErrorEvent`. ⌋()

[rte_sws_7032]⌈ If a data element has been received invalidated in case of Inter-ECU communication where receiving software components requesting different values for `handleInvalid` and the attribute `handleInvalid` is set to `replace` for an particular `r-port` – RTE shall perform the "invalid value substitution" with the `initValue` for the `r-port`. Then the reception will be handled as if a valid value would have been received (activation of runnable entities using the `DataReceivedEvent`). ⌋()

4.3.1.8.2 Data Element Invalidation in case of Intra-ECU communication

Sender:

[rte_sws_5025]⌈ If `data element invalidation` is enabled, and the communication is Intra-ECU, data element invalidation can be implemented by the RTE⌋()

In case of implicit data transmission the RTE shall always implement the data element invalidation and therefore provide an API to set the data element's value to the invalid value. The actual invalid value is specified in the SW-C template `invalidValue`.

Receiver:

[rte_sws_5030]⌈ If a data element has been invalidated in case of Intra-ECU communication and the attribute `handleInvalid` is set to *keep* – the query of the value shall return the same value as if COM would have handled the invalidation (copy COM behavior). Then the reception of the invalid value will be handled as an error and the activation of runnable entities can be performed using the `DataReceiveErrorEvent`. ⌋()

[rte_sws_5049]⌈ If a data element has been received invalidated in case of Intra-ECU communication and the attribute `handleInvalid` is set to *replace* – RTE shall perform the "invalid value substitution" with the `initValue`. Then the reception will be handled as if a valid value would have been received (activation of runnable entities using the `DataReceivedEvent`). ⌋()

4.3.1.9 Filters

By means of the `filter` attribute [RTE00121] an additional filter layer can be added on the receiver side of unqueued S/R-Communication. *Value-based* filters can be defined, i.e. only signal values fulfilling certain conditions are made available for the receiving component. The possible filter algorithms are taken from OSEK COM version 3.0.2. They are listed in the meta model (see [2]. According to the SW-C template [2], filters are only allowed for signals that are compatible to C language unsigned integer types (i.e. characters, unsigned integers and enumerations). Thus, filters cannot be

applied to composite data types like for instance `ApplicationRecordDataType` or `ApplicationArrayDataType`.

[rte_sws_5503]⌈ The RTE shall provide value-based filters on the receiver-side of unqueued S/R-Communication as specified in the SW-C template [2]. ⌋*(RTE00121)*

[rte_sws_5500]⌈ For inter-ECU communication, the RTE shall use the filter implementation of the COM layer [RTE00121]. For intra-ECU and inter-Partition communication, the RTE can use the filter implementation of COM, but may also implement the filters itself for efficiency reasons, without using COM. ⌋*(RTE00019, RTE00121)*

[rte_sws_5501]⌈ The RTE shall support a different filter specification for each data element in a component's AUTOSAR interface. ⌋*(RTE00121)*

4.3.1.10 Buffering

[rte_sws_2515]⌈ The buffering of sender-receiver communication shall be done on the receiver side. This does not imply that COM does no buffering on the sender side. On the receiver side, two different approaches are taken for the buffering of 'data' and of 'events', depending on the value of the software implementation policy. ⌋*(RTE00110)*

4.3.1.10.1 Last-is-Best-Semantics for 'data' Reception

[rte_sws_2516]⌈ On the receiver side, the buffering of 'data' (`swImplPolicy` not 'queued') shall be realized by the RTE by a single data set for each data element instance. ⌋*(RTE00107)*

The use of a single data set provides the required semantics of a single element queue with overwrite semantics (new data replaces old). Since the RTE is required to ensure data consistency, the generated RTE should ensure that non-atomic reads and writes of the data set (e.g. for composite data types) are protected from conflicting concurrent access. RTE may use lower layers like COM to implement the buffer.

[rte_sws_2517]⌈ The RTE shall initialize this data set rte_sws_2516 with a startup value depending on the ports attributes and if the general initialization conditions in rte_sws_7046 are fulfilled. ⌋*(RTE00068, RTE00108)*

[rte_sws_2518]⌈ Implicit or explicit read access shall always return the last received data. ⌋*(RTE00107)*

Requirement rte_sws_2518 applies whether or not there is a `DataReceivedEvent` referencing the `VariableDataPrototype` for which the API is being generated.

[rte_sws_2519]⌈ Explicit read access shall be non blocking in the sense that it does not wait for new data to arrive. The RTE shall provide mutual exclusion of read and write accesses to this data, e.g., by ExclusiveAreas. ⌋*(RTE00109)*

[rte_sws_2520]⌈ When new data is received, the RTE shall silently discard the previous value of the data, regardless of whether it was read or not. ⌋*(RTE00107)*

4.3.1.10.2 Queueing for 'event' Reception

The application of event semantics implies a state change. Events usually have to be handled. In many cases, a loss of events can not be tolerated. Hence the `swImplPolicy` is set to 'queued' to indicate that the received 'events' have to be buffered in a queue.

[rte_sws_2521]⌈ The RTE shall implement a receive queue for each event-like data element (`swImplPolicy` =queued) of a receive port. ⌋*(RTE00107)*

The queueLength attribute of the `QueuedReceiverComSpec` referencing the event assigns a constant length to the receive queue.

[rte_sws_2522]⌈ The events shall be written to the end of the queue and read (consuming) from the front of the queue (i.e. the queue is first-in-first-out). ⌋*(RTE00107, RTE00110)*

[rte_sws_2523]⌈ If a new event is received when the queue is already filled, the RTE shall discard the received event and set an error flag. ⌋*(RTE00107, RTE00110)*

[rte_sws_2524]⌈ The error flag described in rte_sws_2523 shall be reset during the next explicit read access on the queue. In this case, the status value RTE_E_LOST_DATA shall be presented to the application together with the data. ⌋*(RTE00107, RTE00110, RTE00094)*

[rte_sws_2525]⌈ If an empty queue is polled, the RTE shall return with a status RTE_E_NO_DATA to the polling function, (see chap. 5.5.1). ⌋*(RTE00107, RTE00110, RTE00094)*

The minimum size of the queue is 1.

[rte_sws_2526]⌈ The RTE generator shall reject a queueLength attribute of an `QueuedReceiverComSpec` with a queue length ≤ 0. ⌋*(RTE00110, RTE00018)*

4.3.1.10.3 Queueing of mode switches

The communication of mode switch notifications is typically event driven. Accordingly, RTE offers a similar queueing mechanism as for the 'queued' sender receiver communication, described above.

[rte_sws_2718]⌈ The RTE shall implement a receive queue for the mode switch notifications of each mode machine instance. ⌋*(RTE00107)*

The queueLength attribute of the ModeSwitchSenderComSpec referencing the mode machine instance, assigns a constant length to the receive queue. In contrast to the event communication, for mode switch communication, the length is associated with the sender side, the mode manager, because it is unique for the mode machine instance.

[rte_sws_2719]⌈ The mode switch notification shall be written to the end of the queue and read (consuming) from the front of the queue (i.e. the queue is first-in-first-out). ⌋*(RTE00107, RTE00110)*

[rte_sws_2720]⌈ If a new mode switch notification is received when the queue is already filled, the RTE shall discard the received notification. ⌋*(RTE00107, RTE00110)* In this case, Rte_Switch will return an error, see rte_sws_2675.

[rte_sws_2721]⌈ RTE shall dequeue a mode switch notification, when the mode switch is completed. ⌋*(RTE00107, RTE00110, RTE00094)*

The minimum size of the queue is 1.

[rte_sws_2723]⌈ The RTE generator shall reject a queueLength attribute of an ModeSwitchSenderComSpec with a queue length ≤ 0. ⌋*(RTE00110, RTE00018)*

In case of a queue length of 1, RTE will reject new mode switch notifications during the mode transition.

4.3.1.11 Operation

4.3.1.11.1 Inter-ECU Mapping

This section describes the mapping from VariableDataPrototypes to COM signals or COM signal groups for sender-receiver communication. The mapping is described in the input of the RTE generator, in the DataMapping section of the System Template [8].

If a VariableDataPrototype is mapped to a COM signal or COM signal group but the communication is local, the RTE generator can use the COM signal/COM signal group for the transmission or it can use its own direct implementation of the communication for the transmission.

4.3.1.11.1.1 Primitive Data Types

[rte_sws_4504]⌈ If a data element is a primitive type and the communication is inter-ECU, the DataMappings element shall contain a mapping of the data element to at least one COM signal, else the missing data mapping shall be interpreted as an unconnected port. ⌋*(RTE00091)*

The mapping defines all aspects of the signal necessary to configure the communication service, for example, the network signal endianess and the communication bus. The RTE generator only requires the COM signal handle id since this is necessary for invoking the COM API.

[rte_sws_4505]⌈ The RTE shall use the `ComHandleId` of the corresponding `ComSignal` when invoking the COM API for signal. ⌋*(RTE00091)*

The actual COM handle id has to be gathered from the ECU configuration of the COM module. The input information `ComSignalHandleId` is used to establish the link between the `ComSignal` of the COM module's configuration and the corresponding `ISignal` of the System Template.

4.3.1.11.1.2 Composite Data Types

When a data element is a composed type the RTE is required to perform more complex actions to marshall the data [RTE00091] than is the case for primitive data types.

The DataMappings element of the ECU configuration contains (or reference) sufficient information to allow the data item or operation parameters to be transmitted. The mapping indicates the COM signals or signal groups to be used when transmitting a given data item of a given port of a given software component instance within the composition.

[rte_sws_4506]⌈ If a data element is a composite data type and the communication is inter-ECU, the DataMappings element shall contain a mapping of the data element to COM signals such that each element of the composite data type that is a primitive data type is mapped to a separate COM signal(s), else the missing data mapping shall be interpreted as an unconnected port. ⌋*(RTE00091)*

[rte_sws_4507]⌈ If a data element is typed by a composite data type and the communication is inter-ECU, the DataMappings element shall contain a mapping of the data element to COM signals such that each element of the composite data type that is itself a composite data type shall be recursively mapped to a primitive type and hence to a separate COM signal(s). ⌋*(RTE00091)*

The above requirements have two key features; firstly, COM is responsible for endianness conversion (if any is required) of primitive types and, secondly, differing structure member alignment between sender and receiver is irrelevant since the COM signals are packed into I-PDUs by the COM configuration.

The DataMappings shall contain sufficient COM signals to map each primitive element[7] of the AUTOSAR signal.

[rte_sws_4508]⌈ If a data element is a composite data type and the communication is inter-ECU, the DataMappings element shall contain at least one COM signal for each primitive element of the AUTOSAR signal. ⌋*(RTE00091)*

[rte_sws_2557]⌈

1. Each signal that is mapped to an element of the same composite data item shall be mapped to the same signal group.
2. If two signals are not mapped to an element of the same composite data item, they shall not be mapped to the same signal group.
3. If a signal is not mapped to an element of a composite data item, it shall not be mapped to a signal group.

⌋*(RTE00091)*

[rte_sws_5081]⌈ The RTE shall use the `ComHandleId` of the corresponding `ComSignalGroup` when invoking the COM API for signal groups. ⌋*(RTE00091)*

[rte_sws_5173]⌈ The RTE shall use the `ComHandleId` of the corresponding `ComGroupSignal` when invoking the COM API for shadow signals. ⌋*(RTE00091)*

The actual COM handle id has to be gathered from the ECU configuration of the COM module. The input information `ComHandleId` is used to establish the link between the `ComSignalGroup` of the COM module's configuration and the corresponding `ISignalGroup` of the System Template.

The input information `ComHandleId` of shadow signals is used to establish the link between the `ComGroupSignal` of the COM module's configuration and the corresponding `ISignal` of the System Template.

4.3.1.11.2 Atomicity

[rte_sws_4527]⌈ The RTE is required to treat AUTOSAR signals transmitted using sender-receiver communication atomically [RTE00073]. To achieve this the "signal group" mechanisms provided by COM shall be utilized. See rte_sws_2557 for the mapping. ⌋*(RTE00019, RTE00073, RTE00091)*

The RTE decomposes the composite data type into single signals as described above and passes them to the COM module by using the COM API call `Com_SendSignal` (if parameter `RteUseComShadowSignalApi` is `FALSE`) or `Com_UpdateShadowSignal` (if parameter `RteUseComShadowSignalApi` is `TRUE`). As this set of single signals has to be treated as atomic, it is placed in a "signal group". A signal group has to be placed always in a single I-PDU. Thus, atomicity is established. When all signals have

[7] An AUTOSAR signal that is a primitive data type contains exactly one one primitive element whereas a signal that is a composite data type one or more primitive elements.

been updated, the RTE initiates transmission of the signal group by using the COM API call Com_SendSignalGroup.

As would be expected, the receiver side is the exact reverse of the transmission side: the RTE must first call Com_ReceiveSignalGroup precisely once for the signal group and then call Com_ReceiveSignal (if parameter RteUseComShadowSignalApi is FALSE) or Com_ReceiveShadowSignal (if parameter RteUseComShadowSignalApi is TRUE) to extract the value of each signal within the signal group.

A signal group has the additional property that COM guarantees to inform the receiver by invoking a call-back about its arrival only after all signals belonging to the signal group have been unpacked into a shadow buffer.

4.3.1.11.3 Fan-out

Fan-out can be divided into two scenarios; *PDU fanout* where the same I-PDU is sent to multiple destinations and *signal fan-out* where the same signal, i.e. data element is sent in different I-PDUs to multiple receivers.

For Inter-ECU communication, the RTE does not perform PDU fan-out. Instead, the RTE invokes Com_SendSignal once for a primitive data element and expects the fan-out to multiple PDU destinations to occur lower down in the AUTOSAR communication stack. However, it is necessary for the RTE to support *signal fan-out* since this cannot be performed by any lower level layer of the AUTOSAR communication stack.

The data mapping in the System Template[8] is based on the SystemSignal and SystemSignalGroup. The COM module however uses the ISignal and ISignalGroup counterparts (ComSignal, ComSignalGroup, ComGroupSignal) to define the COM API. The RTE Generator needs to identify whether there are several ISignal or ISignalGroup elements defined for the SystemSignal or SystemSignalGroup and implement the fan-out accordingly. Then the corresponding elements in the COM ecu configuration (ComSignal, ComSignalGroup, ComGroupSignal) are required to establish the interaction between Rte and COM.

[rte_sws_6023]⌈ For inter-ECU transmission of a primitive data type, the RTE shall invoke Com_SendSignal for each ISignal to which the primitive data element is mapped. ⌋*(RTE00019, RTE00028)*

For the invocation the ComHandleId from the ComSignal of COM's ecu configuration shall be used (see rte_sws_4505).

If the data element is typed by a composite data type, RTE invokes Com_SendSignal (if parameter RteUseComShadowSignalApi is FALSE) or Com_UpdateShadowSignal (if parameter RteUseComShadowSignalApi is TRUE) for each primitive element (ISignal) in the composite data type and each COM signal to which that primitive

element is mapped, and Com_SendSignalGroup for each ISignalGroup to which the data element is mapped.

[rte_sws_4526]⌈ For inter-ECU transmission of composite data type, the RTE shall invoke Com_SendSignal (if parameter RteUseComShadowSignalApi is FALSE) or Com_UpdateShadowSignal (if parameter RteUseComShadowSignalApi is TRUE) for each ISignal to which an element in the composite data type is mapped and Com_SendSignalGroup for each ISignalGroup to which the composite data element is mapped. ⌋*(RTE00019, RTE00028)*

For the invocation the ComHandleId from the ComGroupSignal and ComSignalGroup of COM's ecu configuration shall be used (see rte_sws_5173 and rte_sws_5081).

For intra-ECU transmission of data elements, the situation is slightly different; the RTE handles the communication (the lower layers of the AUTOSAR communication stack are not used) and therefore must ensure that the data elements are routed to all receivers. For inter-partition communication, RTE may use the IOC.

[rte_sws_6024]⌈ For inter-partition transmission of data elements, the RTE shall perform the fan-out to each receiver. ⌋*(RTE00019, RTE00028)*

When the fan-out is performed at the PDU level by the PDU Router, no transmission acknowledgment are routed to the upper layers. In order to rely on the Rte_Feedback return values in case of fan-out, the fan-out performed by the RTE at the signal level should be used.

4.3.1.11.4 Fan-in

When receiving data from multiple senders in inter-ECU communication, either the RTE on the receiver side has to collect data received in different COM signals or COM signal groups and pass it to one receiver or the RTE on the sender side has to provide shared access to a COM signal or COM signal group to multiple senders. The receiver RTE, which has to handle multiple COM signals or signal groups, is notified about incoming data for each COM signal or COM signal group separately but has to ensure data consistency when passing the data to the receiver. The sender RTE, which has to handle multiple senders sharing COM signals or signal groups, has to ensure consistent access to the COM API, since COM API calls for the same signal are not reentrant.

[rte_sws_3760]⌈ If multiple senders use different COM signals or signal groups for inter-ECU transmission of a data element prototype with swImplPolicy different from 'queued' to a receiver, the RTE on the receiver side has to pass the last received value to the receiver component while ensuring data consistency. ⌋*(RTE00019, RTE00131)*

[rte_sws_3761]⌈ If multiple senders use different COM signals or signal groups for inter-ECU transmission of a data element prototype with event semantics to a receiver,

the RTE on the receiver side has to queue all incoming values while ensuring data consistency. ⌋*(RTE00019, RTE00131)*

[rte_sws_3762]⌈ If multiple senders share COM signals or signal groups for inter-ECU transmission of a data element prototype to a receiver, the RTE on the sender side shall ensure that the COM API for those signals is not invoked concurrently. ⌋*(RTE00019, RTE00131)*

4.3.1.11.5 Sequence diagrams of Sender Receiver communication

Figure 4.34 shows a sequence diagram of how Sender Receiver communication for data transmission and non-blocking reception may be implemented by RTE. The sequence diagram also shows the `Rte_Read` API behavior if an `initValue` is specified.

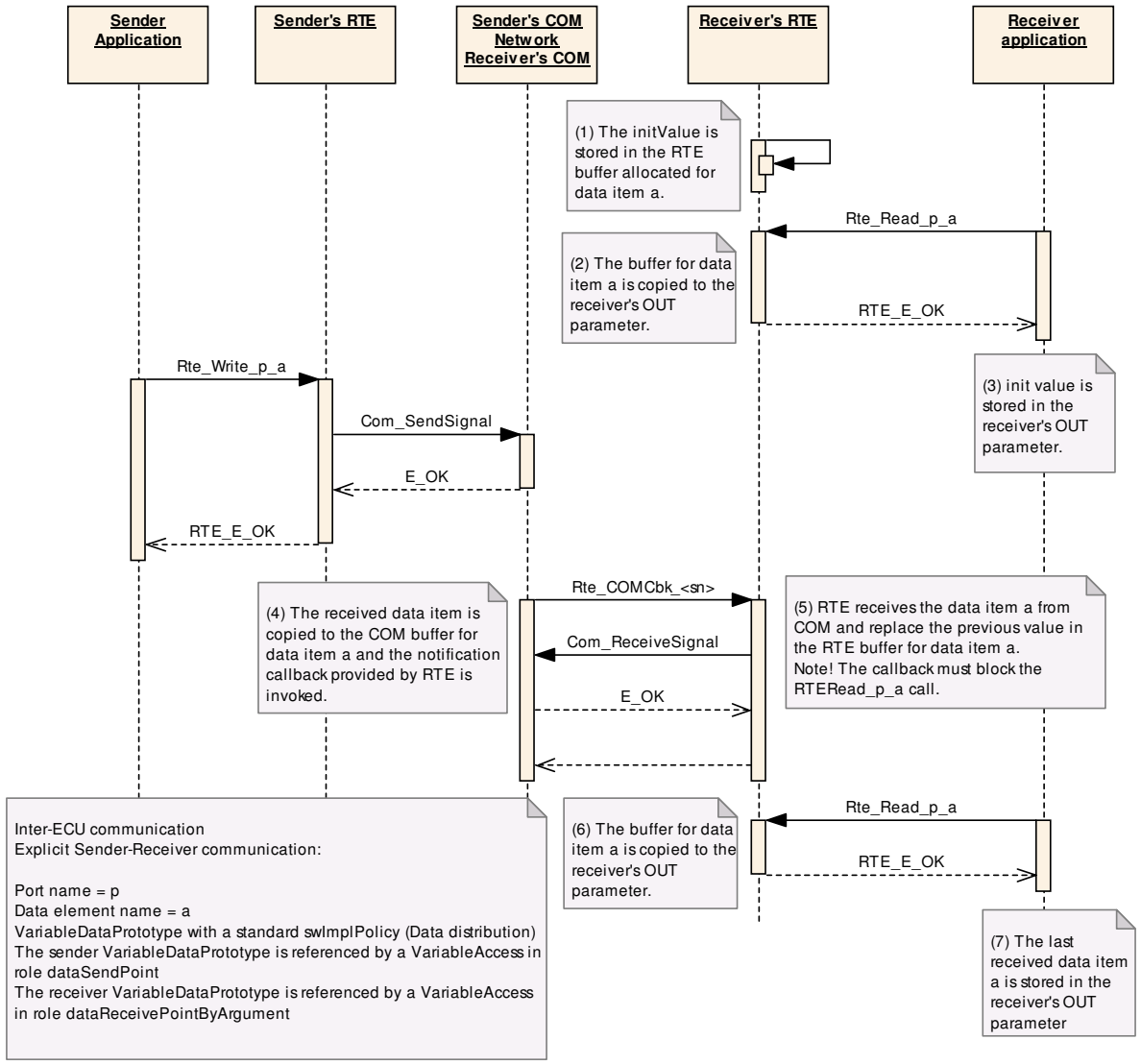


Figure 4.34: Sender Receiver communication with data semantics and dataReceivePointByArgument as reception mechanism

Figure 4.35 shows a sequence diagram of how Sender Receiver communication for event transmission and non-blocking reception may be implemented by RTE. The sequence diagram shows the `Rte_Receive` API behavior when the queue is empty.

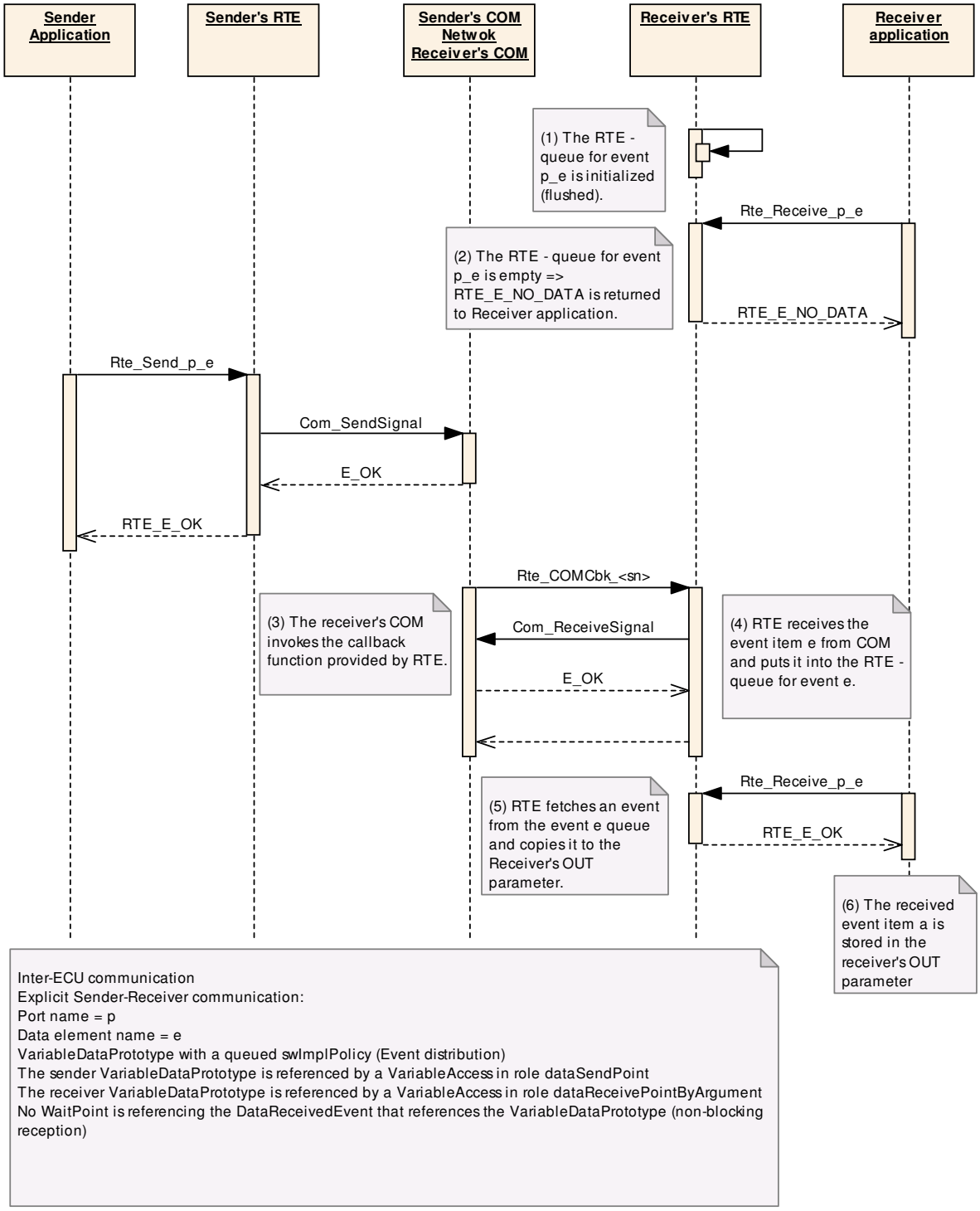


Figure 4.35: Sender Receiver communication with event semantics and dataReceivePointByArgument as reception mechanism

Figure 4.36 shows a sequence diagram of how Sender Receiver communication for event transmission and activation of runnable entity on the receiver side may be implemented by RTE.

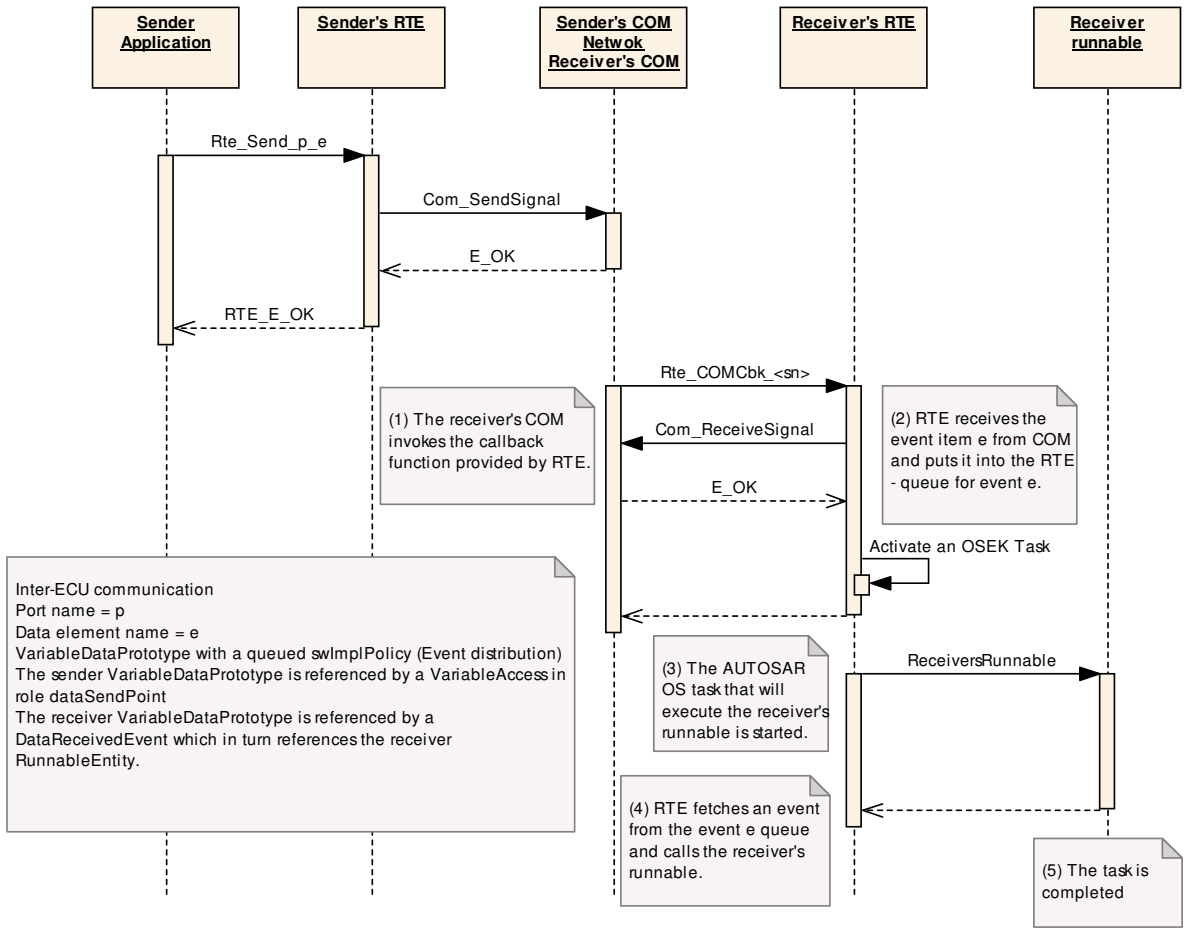


Figure 4.36: Sender Receiver communication with event semantics and activation of runnable entity as reception mechanism

4.3.1.12 "Never received status" for Data Element

The Software Component template allows specifying whether an unqueued data, defined in an AUTOSAR Interface, has been updated since system start (or partition

restart) or not. This additional optional status establishes the possibility to check whether a data element has been changed since system start (or partition restart).

[rte_sws_7381]⌈ On receiver side the `handleNeverReceived` attribute of the `NonqueuedReceiverComSpec` shall specify the handling of the never received status. ⌋*(RTE00184)*

[rte_sws_7382]⌈ The initial status of the data elements with the attribute `handleNeverReceived` set to TRUE shall be `RTE_E_NEVER_RECEIVED`. ⌋*(RTE00184)*

[rte_sws_7383]⌈ The initial status of the data elements with the attribute `handleNeverReceived` set to `TRUE` shall be cleared when the first reception occurs. ⌋*(RTE00184)*

[rte_sws_7645]⌈ The status of data elements shall be reset on the receiver side to `RTE_E_NEVER_RECEIVED` when the receiver's partition is restarted. ⌋*(RTE00184, RTE00224)*

4.3.1.13 "Update flag" for Data Element

The Software Component template allows specifying whether an unqueued data, defined in an AUTOSAR Interface, has been updated since last read or not. This additional optional status establishes the possibility to check, whether a data element has been updated since last read.

[rte_sws_7385]⌈ On receiver side the "`enableUpdate`" attribute of the `NonqueuedReceiverComSpec` shall activate the handling of the update flag. ⌋*(RTE00179)*

[rte_sws_7386]⌈ The update flag of the data elements configured with the "`enableUpdate`" attribute shall be set by receiving new data from COM or from a local software-conponent. ⌋*(RTE00179)*

[rte_sws_7387]⌈ The update flag shall be cleared after each read by `Rte_Read` or `Rte_DRead` of the data element. ⌋*(RTE00179)*

[rte_sws_7689]⌈ The update flag shall be cleared when the RTE is started or when the partition of the software-component is restarted. ⌋*(RTE00179)*

The update flag can be queried by the `Rte_IsUpdated` API, see 5.6.34.

4.3.1.14 Dynamic data type

Dynamic data are data whose length varies at runtime.

This includes:

- arrays with variable number of elements
- structures including arrays with variable number of elements

This excludes:

- structures including variable number of elements

The length of the dynamic data is accessed thanks to the optional parameter `<length>` of the RTE APIs for communication. See chapter 5 for more information.

In case of inter-ECU communication, dynamic data are mapped to dynamic signals and received/transmitted through the TP by the COM stack.

With the current release of SWS_COM, COM limits the dynamic signals to the `ComSignalType` UINT_8DYN (see the requirement COM569).

In order to respect the VFB concept the capability of inter-ECU and intra-ECU communication should be equal. So it has been decided to extend these limitation from COM also to the intra-ECU communication. As a consequence the only one dynamic data type supported by the RTE is the type uint8[n] whatever the communication is intra or inter-ECU. See rte_sws_7810.

4.3.1.15 Inter-ECU communication through TP

Inter-ECU communication can be configured in COM to be supported by the TP. This is especially necessary if:

- Size of the signal exceed the size of the L-PDU (large signals)
- Size of the signal group exceed the size of the L-PDU
- Size of the signal varies at runtime (dynamic signals)

In the current release of SWS_COM, COM APIs to access signal values might return the error code COM_BUSY for the signals mapped to N-PDU. This error code indicates that the access to the signal value has failed (internally rejected by COM) and should be retried later. This situation might only be possible when the transmission or the reception of the corresponding PDU is in progress in COM at the time the access to the signal value is requested.

This is a problem for the handling of data with data semantic (last is best behavior) because:

- "COM_BUSY like" errors are not compatible with real time systems that should have predictable response time.
- Forwarding this error code to the application implies that every applications should handle it (implement a retry) even if it will never comes (data is not be mapped to N-PDU).
- Error code can not be forwarded to the application in case of direct read or implicit write.

This is not a problem for the handling of data with event semantic (last is best behavior) because:

- The COM_BUSY error should not be possible during the execution of COM call-backs (Rx indication and Tx confirmation) that can be used by the RTE to handle the queue.
- Data are queued internally by RTE and accessible at any time by the application.

Note: First point is especially true if the `ComIPduSignalProcessing` is configured as `IMMEDIATE`. But if the `ComIPduSignalProcessing` is configured as `DEFFERED` and 2 events are closely received, it is possible that at the time the RTE tries to access the corresponding COM signal the second event reception has already started. In this case the RTE will received COM_BUSY and the event will be lost but it is more a problem of configuration than a limitation from COM.

As a consequence it has been decided to limit the data mapped to N-PDU to the event semantic (queued behavior). See rte_sws_7811.

Note: As the data mapping is not mandatory for the RTE contract phase, it is possible that a configuration is accepted at contract phase but rejected at generation phase when the data mapping is known.

Dynamic data are always mapped to N-PDU in case of inter-ECU communication. So in order to avoid such situation (late rejection at generation phase) and in order to respect the VFB concept (intra and inter-ECU should be equal) it has been decided to extend this limitation to every dynamic data whatever the communication is intra or inter-ECU. See rte_sws_7812.

4.3.1.16 Inter-ECU communication of arrays of bytes

Generally the communication of arrays in the case of inter-ECU communication must make use of the signal group mechanisms to send an array to COM. This implies sending each array element to a shadow buffer in COM (with `Com_SendSignal()` API, if parameter `RteUseComShadowSignalApi` is `FALSE` or `Com_UpdateShadowSignal()` API, if parameter `RteUseComShadowSignalApi` is `TRUE`), and in the end send the signal group (with `Com_SendSignalGroup()` API).

An exception to this general rule is for arrays of bytes. In this case, the RTE shall use the native COM interface to send directly the data.

[rte_sws_7408]⌈ The RTE shall use the `Com_SendSignal` or `Com_ReceiveSignal` APIs to send or receive fixed-length arrays of bytes. ⌋*(RTE00231)*

[rte_sws_7817]⌈ The RTE shall use the `Com_SendDynSignal` or `Com_ReceiveDynSignal` APIs to send or receive variable-length arrays of bytes. ⌋*(RTE00231)*

4.3.1.17 Handling of acknowledgment events

As a general rule, the acknowledgment events `DataWriteCompletedEvent` and `DataSendCompletedEvent` shall be raised immediately after the sending to all receivers has been performed and in case of Inter-ECU communication all acknowledgments from COM have been received. As part of the implementation detailed rules for the following communication scenarios have to be considered:

Intra-Partition communication

[rte_sws_8017]⌈ For intra-partition communication with implicit `dataWriteAccess` the `DataWriteCompletedEvent` shall be fired if and only if a task terminates and the write-back copy actions to the global RTE-buffer are completed. ⌋*(RTE00122)*

[rte_sws_8043]⌈ For intra-partition communication with incoherent implicit `dataWriteAccess` no write-back copy actions to a global RTE-buffer will be performed, if the involved runnables are all running in one preemption area. In this case the `DataWriteCompletedEvent` shall be fired after the termination of the last sending runnable in the sending task. ⌋*(RTE00122)*

[rte_sws_8018]⌈ For intra-partition communication with explicit `dataSendPoint` the `DataSendCompletedEvent` shall be fired if and only if the sending to all receivers has been performed. ⌋*(RTE00122)*

Inter-Partition communication

[rte_sws_8020]⌈ For inter-partition communication with implicit `dataWriteAccess` the `DataWriteCompletedEvent` shall be fired if and only if a task terminates and the write-back copy actions to the global RTE-buffer are completed. In addition the execution of the data write operations at the data receiver partitions must have taken place. Thereby the return status of the `IOC` for the different write operations can be neglected. ⌋*(RTE00122)*

[rte_sws_8044]⌈ For inter-partition communication with incoherent implicit `dataWriteAccess` no write-back copy actions to a global RTE-buffer will be performed, if the involved runnables are all running in one preemption area. In this case the `DataWriteCompletedEvent` shall be fired after the termination of the last sending runnable in the sending task and after the execution of the data write operations at the data receiver partitions have taken place. Thereby the return status of the `IOC` for the different write operations can be neglected. ⌋*(RTE00122)*

[rte_sws_8021]⌈ For inter-partition communication with explicit `dataSendPoint` the `DataSendCompletedEvent` shall be fired if and only if the sending to all receivers has been performed and the execution of the data write operations at the data receiver partitions have taken place. Thereby the return status of the `IOC` for the different write operations can be neglected. ⌋*(RTE00122)*

Inter-ECU communication

[rte_sws_8022]⌈ For inter-ECU communication with implicit `dataWriteAccess` the `DataWriteCompletedEvent` shall be fired if and only if a task terminates and the

write-back copy actions to the global RTE-buffer are completed. In addition the transmission acknowledgment from COM must be complete, i.e. the acknowledgment has been received and in case of RTE-fanout all acknowledgments have been received. ⌋*(RTE00122)*

[rte_sws_8045]⌈ For inter-ECU communication with incoherent implicit `dataWriteAccess` no write-back copy actions to a global RTE-buffer will be performed, if the involved runnables are all running in one preemption area. In this case the `DataWriteCompletedEvent` shall be fired after the termination of the last sending runnable in the sending task and after the transmission acknowledgment from COM is complete, i.e. the acknowledgment has been received and in case of RTE-fanout all acknowledgments have been received. ⌋*(RTE00122)*

[rte_sws_8023]⌈ For inter-ECU communication with explicit `dataSendPoint` the `DataSendCompletedEvent` shall be fired if and only if the sending to all receivers has been performed and the transmission acknowledgment from COM is complete, i.e. the acknowledgment has been received and in case of RTE-fanout all acknowledgments have been received. ⌋*(RTE00122)*

4.3.2 Client-Server

4.3.2.1 Introduction

Client-server communication involves two entities, the client which is the requirer (or user) of a service and the server that provides the service.

[rte_sws_5110]⌈ A client is defined as one `ClientServerOperation` in one `RPortPrototype` of one Software Component instance. ⌋*(RTE00029)*

For the definition of the client in rte_sws_5110 neither the number of `ServerCallPoint`s nor `RunnableEntity` accesses to the `ServerCallPoint` are relevant. A Software Component instance can appear as several clients to the same server if it defines `ServerCallPoint`s for several `PortPrototype`s of the same `PortInterface`'s `ClientServerOperation`.

[rte_sws_5163]⌈ A server is defined as one `RunnableEntity` which is the target of an `OperationInvokedEvent`. Serialization is on activation of `RunnableEntity`. ⌋*(RTE00029)*

The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server, in the form of the RTE, waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client's request. So, the direction of initiation is used to categorize whether a AUTOSAR software-component is a client or a server.

A single component can be both a client and a server depending on the software realization.

The invocation of a server is performed by the RTE itself when a request is made by a client. The invocation occurs synchronously with respect to the RTE (typically via a function call) however the client's invocation can be either synchronous (wait for server to complete) or asynchronous with respect to the server.

Note: servers which have an asynchronous operation (i.e. they accept a request an another provide a feedback by invoking a server of the caller) should be avoided as the RTE does not know the link between these 2 client-server communications. In particular, the server should have no OUT (or INOUT) parameters because the RTE cannot perform the copy of the result in the caller's environment when the request was processed.

[rte_sws_6019]⌈ The only mechanism through which a server can be invoked is through a client-server invocation request from a client. ⌋*(RTE00029)*

The above requirement means that *direct invocation* of the function implementing the server outside the scope of the RTE is not permitted.

A server has a dedicated provide port and a client has a dedicated require port. To be able to connect a client and a server, both ports must be categorized by the same interface.

The client can be blocked (synchronous communication) respectively non-blocked (asynchronous communication) after the service request is initiated until the response of the server is received.

A server implemented by a RunnableEntity with attribute canBeInvokedConcurrently set to FALSE is not allowed to be invoked concurrently and since a server can have one or more clients the server may have to handle concurrent service calls (n:1 communication) the RTE must ensure that concurrent calls do not interfere.

[rte_sws_4515]⌈ It is the responsibility of the RTE to ensure that serialization[8] of the operation is enforced when the server runnable attribute `canBeInvokedConcurrently` is FALSE. ⌋*(RTE00019, RTE00033)*

Note that the same server may be called using both synchronous and asynchronous communication.

Note also that even when `canBeInvokedConcurrently` is FALSE, an `AtomicSwComponentType` might be instantiated multiple times. In this case, the implementation of the `RunnableEntity` can still be invoked concurrently from several tasks. However, there will be no concurrent invocations of the implementation with the same instance handle.

[rte_sws_4516]⌈ The RTE's implementation of the client-server communication has to ensure that a service result is dispatched to the correct client if more than one client uses a service. ⌋*(RTE00019, RTE00080)*

The result of the client/server operation can be collected using "wake up of wait point", "explicit data read access" or "activation of runnable entity".

[rte_sws_7409]⌈ The RTE generator shall support the optimization of a client-server call to a direct function call without interaction with the RTE or the communication services, at least when the following conditions are true:

- the server runnable's property `canBeInvokedConcurrently` is set to TRUE
- the client and server execute in the same partition, i.e. `intra-partition` Client-Server communication
- the `ServerCallPoint` is Synchronous
- the `OperationInvokedEvent` is not mapped to an `OsTask`

⌋*()*

[8] Serialization ensures at most one thread of control is executing an instance of a runnable entity at any one time. An AUTOSAR software-component can have multiple instances (and therefore a runnable entity can also have multiple instances). Each instance represents a different server and can be executed in parallel by different threads of control thus serialization only applies to an individual instance of a runnable entity – multiple runnable entities within the same component instance may also be executed in parallel.

Note: In case the conditions in rte_sws_4522 are fulfilled the RTE Generator may implement a client-server call with a direct function call, even when the server runnable's property `canBeInvokedConcurrently` is set to FALSE.

Since the communication occurs conceptually via the RTE (it is initiated via an RTE API call) the optimization does not violate the requirement that servers are only invoked via client-server requests (see Sect. 5.6.13).

[rte_sws_7662]⌈ The RTE Generator shall reject configurations where an `ClientServerOperation` has an `ArgumentDataPrototype` whose `ImplementationDataType` is of category `DATA_REFERENCE` and whose `direction` is `OUT` or `INOUT`. ⌋*(RTE00018, RTE00019)*

4.3.2.2 Multiplicity

Client-server interfaces contain two dimensions of multiplicity; multiple clients invoking a single server and multiple operations within a client-server interface.

4.3.2.2.1 Multiple Clients Single Server

Client-server communication involves an AUTOSAR software-component invoking a defined "server" operation in another AUTOSAR software-component which may or may not return a reply.

[rte_sws_4519]⌈ The RTE shall support multiple clients invoking the same server operation ('n:1' communication where n $\geq$ 1). ⌋*(RTE00029)*

4.3.2.2.2 Multiple operations

A client-server interface contains one or more operations. A port of a AUTOSAR software-component that *requires* an AUTOSAR client-server interface to the component can independently invoke any of the operations defined in the interface [RTE00089].

[rte_sws_4517]⌈ The RTE API shall support independent access to operations in a client-server interface. ⌋*(RTE00029)*

Example 4.5

Consider a client-server interface that has two operations, `op1` and `op2` and that an AUTOSAR software-component definition requires a port typed by the interface. As a result, the RTE generator will create two API calls; one to invoke `op1` and another to invoke `op2`. The calls can invoke the server operations either synchronously or asynchronously depending on the configuration.

Recall that each data element in a sender-receiver interface is transmitted independently (see Section 4.3.1.3) and that the coherent transmission of multiple data items is achieved through combining multiple items into a single composite data type. The transmission of the parameters of an operation in a client-server interface is similar to a record since the RTE guarantees that all parameters are handled atomically [RTE00073].

[rte_sws_4518]⌈ The RTE shall treat the parameters (and results) of a client-server operation atomically. ⌋*(RTE00033)*

However, unlike a sender-receiver interface, there is no facility to combine multiple client-server operations so that they are invoked as a group.

4.3.2.2.3 Single Client Multiple Server

The RTE is *not* required to support multiple server operations invoked by a single client component request ('1:n' communication where n $>$ 1).

4.3.2.2.4 Serialization

Each client can invoke the server simultaneously and therefore the RTE is required to support multiple requests of servers. If the server requires serialization, the RTE has to ensure it.

[rte_sws_4520]⌈ The RTE shall support simultaneous invocation requests of a server operation. ⌋*(RTE00019, RTE00080)*

[rte_sws_4522]⌈ The RTE shall ensure that the `RunnableEntity` implementing a server operation has completed the processing of a request before it begins processing the next request, if serialization is required by the server operation, i.e `canBeInvokedConcurrently` attribute of the server is set to FALSE and client `RunnableEntity`s to `OsTask` mapping (`RteEventToTaskMapping`) may lead to concurrent invocations of the server. ⌋*(RTE00019, RTE00033)*

When this requirement is met the operation is said to be "serialized". A serialized server only accepts and processes requests atomically and thus avoids the potential for conflicting concurrent access.

Client requests that cannot be serviced immediately due to a server operation being "busy" are required to be queued pending processing. The presence and depth of the queue is configurable.

If the `RunnableEntity` implementing the server operation is reentrant , i.e. `canBeInvokedConcurrently` attribute set to TRUE, no serialization is necessary. This allows to implement invocations of reentrant server operations as direct function calls without involving the RTE.

But even when the canBeInvokedConcurrently attribute is set to FALSE the RTE Generator still can utilize a direct function call, if the mapping of client the RunnableEntitys to OsTasks will not imply a concurrent execution of the server.

[rte_sws_8001]⌈ If two operations are mapped to the same RunnableEntity, and rte_sws_4522 requires a serialization, then the operation invoked events shall be mapped to same task and they shall have the same position in task. Otherwise the RTE Generator shall reject configuration. ⌋*(RTE00019, RTE00033)*

[rte_sws_8002]⌈ If two operations are mapped to the same RunnableEntity, and rte_sws_4522 requires a serialization, then a single queue is implemented for invocations coming from any of the operations. ⌋*(RTE00019, RTE00033)*

4.3.2.3 Communication Time-out

The ServerCallPoint allows to specify a timeout so that the client can be notified that the server is not responding and can react accordingly. If the client invokes the server synchronously, the RTE API call to invoke the server reports the timeout. If the client invokes the server asynchronously, the timeout notification is passed to the client by the RTE as a return value of the API call that collects the result of the server operation.

[rte_sws_3763]⌈ The RTE shall ensure that timeout monitoring is performed for client-server communication, regardless of the receive mode for the result. ⌋*(RTE00069, RTE00029)*

If the server is invoked asynchronously and a WaitPoint is specified to collect the result, two timeout values have to be specified, one for the ServerCallPoint and one for the WaitPoint.

[rte_sws_3764]⌈ If different timeout values are specified for the AsynchronousServerCallPoint and for the WaitPoint associated with the AsynchronousServerCallReturnsEvent for this AsynchronousServerCallPoint, the configuration shall be rejected by the RTE generator. ⌋*(RTE00018)*

In asynchronous client-server communication the AsynchronousServerCallReturnsEvent associated with the AsynchronousServerCallPoint for an ClientServerOperation shall indicate that the server communication is finished or that a timeout occurred. The status information about the success of the server operation shall be available as the return value of the RTE API call generated to collect the result.

[rte_sws_3765]⌈ For each asynchronous invocation of an operation prototype only one AsynchronousServerCallReturnsEvent shall be passed to the client component by the RTE. The AsynchronousServerCallReturnsEvent shall indicate

either that the transmission was successful or that the transmission was not successful. ⌋*(RTE00079)*

[rte_sws_3766]⌈ The status information about the success or failure of the asynchronous server invocation shall be available as the return value of the RTE API call to retrieve the result. ⌋*(RTE00079)*

After a timeout was detected, no result shall be passed to the client.

[rte_sws_3770]⌈ If a timeout was detected by the RTE, no result shall be passed back to the client. ⌋*(RTE00069, RTE00029)*

Since an asynchronous client can have only one outstanding server invocation at a time, the RTE has to monitor when the server can be safely invoked again. In normal operation, the server can be invoked again when the result of the previous invocation was collected by the client.

[rte_sws_3773]⌈ If a server is invoked asynchronously and no timeout occurred, the RTE shall ensure that the server can be invoked again by the same client, after the result was successfully passed to the client. ⌋*(RTE00069)*

In intra-partition client-server communication, the RTE can determine whether the server runnable is still running or not.

[rte_sws_3771]⌈ If a timeout was detected in asynchronous intra-partition client-server communication, the RTE shall ensure that the server is not invoked again by the same client until the server runnable has terminated. ⌋*(RTE00069, RTE00079)*

In inter-ECU communication, the client RTE has no knowledge about the actual status of the server. The response of the server could have been lost because of a communication error or because the server itself did not respond. Since the client-side RTE cannot distinguish the two cases, the client must be able to invoke the server again after a timeout expired. As partitions in one ECU are decoupled in a similar way like separate ECUs, and can be restarted separately, client server communication should behave similar for inter-ECU and intra-partition communication.

[rte_sws_3772]⌈ If a timeout was detected in asynchronous `inter-ECU` or `inter-partition` client-server communication, the RTE shall ensure that the server can be invoked again by the same client after the timeout notification was passed to the client. ⌋*(RTE00069, RTE00079)*

Note that this might lead to client and server running out of sync, i.e. the response of the server belongs to the previous, timed-out invocation of the client. The application has to handle the synchronization of client and server after a timeout occurred.

[rte_sws_3767]⌈ If the timeout value of the ServerCallPoint is 0, no timeout monitoring shall be performed. ⌋*(RTE00069, RTE00029)*

[rte_sws_3768]⌈ If the canBeInvokedConcurrently attribute of the server runnable is set to TRUE, no timeout monitoring shall be performed if the RTE API call to invoke the server is implemented as a direct function call. ⌋*(RTE00069, RTE00029)*

[rte_sws_2709]⌈ In case of inter partition communication, if the partition of the server is stopped or restarting at the invocation time of the server call or during the operation of the server call, the client shall immediately receive a timeout. ⌋*()*

Note: In case of inter-ECU or interpartition client-server communication it is recommended to always specify a timeout>0. Otherwise in case of a full server queue the client would wait for the server response infinitely.

4.3.2.4 Port-Defined argument values

Port-defined argument values exist in order to support interaction between Application Software Components and Basic Software Modules.

Several Basic Software Modules use an integer identifier to represent an object that should be acted upon. For instance, the NVRAM Manager uses an integer identifier to represent the NVRAM block to access. This identifier is not known to the client, as the client must be location independent, and the NVRAM block to access for a given application software component cannot be identified until components have been mapped onto ECUs.

There is therefore a mismatch between the information available to the client and that required by the server. Port-defined argument values bridge that gap.

The required port-defined arguments (the fact that they are required, their data type and their values) are specified within the input to the RTE generator.

[rte_sws_1360]⌈ When invoking the runnable entity specified for an OperationInvokedEvent, the RTE must include the port-defined argument values between the instance handle (if it is included) and the operation-specific parameters, in the order they are given in the template. ⌋*(RTE00152)*

Requirement rte_sws_1360 means that a client will make a request for an operation on a require (Client-Server) port including only its instance handle (if required) and the explicit operation parameters, yet the server will be passed the implicit parameters as it requires.

Note that the values of implicit parameters are constant for a particular server runnable entity; it is therefore expected that using port-defined argument values imposes no RAM overhead (beyond any extra stack required to store the additional parameters).

4.3.2.5 Buffering

Client-Server-Communication is a two-way-communication. A request is sent from the client to the server and a response is sent back.

Unless a server call is implemented as direct function call, the RTE shall store or buffer the communication on the corresponding receiving sides, requests on server side and responses on client side, respectively:

- **[rte_sws_2527]**⌈ Unless a server call is implemented as a direct function call, the RTE shall buffer a request on the server side in a first-in-first-out queue as described in chapter 4.3.1.10.2 for queued data elements.

  Note: The data that shall be buffered is implementation specific but at least RTE should store the IN parameters, the IN/OUT parameters and a client identifer. ⌋*(RTE00019, RTE00033, RTE00110)*

- **[rte_sws_2528]**⌈ Unless a server call is implemented as a direct function call, RTE shall keep the response on the client side in a queue with queue length 1.

  Note: The data that shall be buffered is implementation specific but at least RTE should store the IN/OUT parameters, the OUT parameters and the error code. ⌋*(RTE00019, RTE00033)*

For the server side, the `ServerComSpec.queueLength` attribute specifies the length of the queue.

[rte_sws_2529]⌈ The RTE generator shall reject a `queueLength` attribute of a `ServerComSpec` with a queue length ≤ 0. ⌋*(RTE00033, RTE00110, RTE00018)*

[rte_sws_2530]⌈The RTE shall use the queue of requests to serialise access to a server. ⌋*(RTE00033, RTE00110)*

A buffer overflow of the server is not reported to the client. The client will receive a time out.

[rte_sws_7008]⌈ If a server call is implemented by direct function call the RTE shall not create any copies for parameters passed by reference. Therefore, it is the responsibility of the application to provide consistency mechanisms for referenced parameters if necessary. ⌋*(RTE00033, RTE00110)*

4.3.2.6 Inter-ECU and Inter-Partition Response to Request Mapping

RTE is responsible to map a response to the corresponding request. With this mapping, RTE can activate or resume the corresponding runnable and provide the response to the correct client. The following situations can be distinguished:

- Mapping of a response to the correct request within one ECU. In general, this is solved already by the call stack. The details are implementation specific and will not be discussed in this document.
- Mapping of a response coming from a different partition or a different ECU.

The problem of request to response mapping in inter-ECU and inter-Partition communication can be split into:

- Mapping of a response to the correct client. This is discussed in 4.3.2.6.1.
- Mapping of a response to the correct request within of one client. This is discussed in 4.3.2.6.2.

The general approach for the inter-ECU and inter-Partition request response mapping is to use transaction handles.

[rte_sws_2649]⌈ In case of inter-ECU client-server communication, the transaction handle shall contain two optional parts of unsigned integer type with configurable size,

- the client identifier
- and a sequence counter.

⌋*(RTE00082)*

The presence of an part of the transaction handle is an input to the RTE Generator and up to the system design.

[rte_sws_7346]⌈ In case of inter-Partition client-server communication, no response shall be communicated by the RTE to the client if the client is part of a partition that was restarted since the request was sent. ⌋*(RTE00082)*

rte_sws_7346 could be implemented with a transaction handle that contains a sequence counter.

[rte_sws_2651]⌈ In case of inter-ECU client-server communication, the optional transaction handle shall be used for the identification of client server transactions communicated via COM. ⌋*(RTE00082)*

[rte_sws_2652]⌈ If configured: in case of inter-ECU client-server communication, the transaction handle shall be bundled with the parameters of a request or response in the same `SystemSignalGroup`. ⌋*(RTE00082)*

[rte_sws_2653]⌈ The RTE on the server side shall return the transaction handle of the request without modification together with the response. ⌋*(RTE00082)*

Since there is always at most one open request per client (see rte_sws_2658), the transaction handle can be kept within the RTE and does not have to be exposed to the SW-C.

4.3.2.6.1 Client Identifier

In case of a server on one ECU with clients on other ECUs, the inter-ECU client-server communication shall use different unique `SystemSignal`s and `SystemSignalGroup`s for each client-ECU to allow the identification of the client-ECU associated with each client call.

[rte_sws_2579]⌈ The RTE Generator shall reject configurations where there is inter-ECU client-server communication from several client-ECUs using the same `SystemSignal`s and/or `SystemSignalGroup`s. ⌋*(RTE00029, RTE00082, RTE00018)*

With this mechanism, the server-side RTE must handle the fan-in. This is done in the same way as for sender-receiver communication.

However it is allowed to have several clients in one client-ECU communicating using inter-ECU client-server communication with a server on a different ECU, if the client identifier is used to distinguish the different clients.

[rte_sws_5111]⌈ The RTE Generator shall reject configurations where there is inter-ECU client-server communication from several clients on the same client-ECU and no client identifiers are configured for all of these inter-ECU client-server communications. ⌋*(RTE00018)*

[rte_sws_3769]⌈ If multiple clients have access to one server, the RTE on the server side has to queue all incoming server invocations while ensuring data consistency. ⌋*(RTE00019, RTE00029)*

[rte_sws_5066]⌈ The data type used to hold the client identifier shall be derived from the system template's [8] `length` attribute of the corresponding `SystemSignal` referenced by the `ClientIdMapping`. ⌋*(RTE00082)*

The structure is shown in figure 4.37.

4.3.2.6.2 SequenceCounter

The purpose of sequence counters is to map a response to the correct request of a known client.

[rte_sws_2658]⌈ In case of inter-ECU and inter-Partition communication, RTE shall allow only one request per client and server operation at any time. ⌋*(RTE00079)*

rte_sws_2658 does not apply to intra-partition communication because there can be several `execution-instances`.

rte_sws_2658 implies under normal operation that a response can be mapped to the previous request. But, when a request or response is lost or delayed, this order can get out of phase. To allow a recovery from lost or delayed signals, a sequence counter is used. The sequence counter can also be used to detect stale responses after a restart of the client side RTE and SW-C.

[rte_sws_2654]⌈ RTE shall support a sequence counter for the inter ECU client server connection where configured in the input information. ⌋*(RTE00082)*

[rte_sws_2655]⌈ RTE shall initialize all sequence counters with zero during `Rte_Start`. ⌋*(RTE00082)*

[rte_sws_2656]⌈ RTE shall increase each sequence counter in a cyclic manner after a client server operation has finished successfully or with a timeout. ⌋*(RTE00082)*

[rte_sws_2657]⌈ RTE shall ignore incoming responses that do not match the sequence counter. ⌋*(RTE00082)*

[rte_sws_5067]⌈ The data type used to hold the sequence counter shall be derived from the system template's [8] `length` attribute of the corresponding `SystemSignal` referenced by the `SequenceCounterMapping`. ⌋*(RTE00082)*

The structure is shown in figure 4.37.

4.3.2.7 Operation

4.3.2.7.1 Inter-ECU Mapping

The client server protocol defines how a client call and the server response are mapped onto the communication infrastructure of AUTOSAR is case of inter-ECU communication. This allows RTE implementations from different vendors to interpret the client server communication in the same way.

The AUTOSAR System Template [8] does specify a protocol for the client server communication in AUTOSAR. A short overview of the major elements is provided in this section.

The structure in figure 4.37 describes the client server protocol as defined in the AUTOSAR System Template [8].

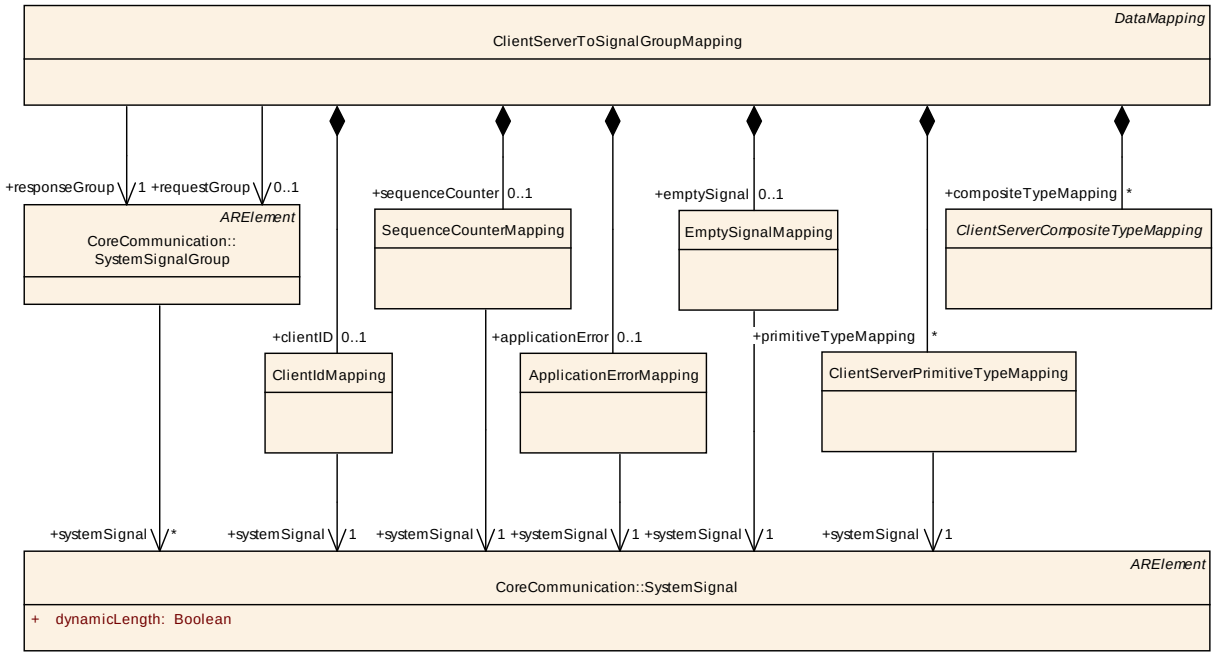


Figure 4.37: Standardized client server protocol

For each `ClientServerOperation` defined at a `PortPrototype` of one Software Component instance one `ClientServerToSignalGroupMapping` object has to be defined representing the server call and the response (with references to the request and response `SystemSignalGroup`s) of this specific client.

[rte_sws_5054]⌈ The RTE Generator shall reject an input configuration where for any configured inter-ECU client-server communication (comprised of the `ClientServer-Operation` of a `PortPrototype` of one Software Component instance) there is not one and only one `ClientServerToSignalGroupMapping` defined. ⌋*(RTE00082, RTE00018)*

[rte_sws_5055]⌈ The RTE Generator shall use the `ClientServerToSignal-GroupMapping` information to establish the configuration with the lower layers of AUTOSAR (e.g. COM). ⌋*(RTE00082)*

[rte_sws_6028]⌈ The arguments, application errors, client identifier, and sequence counter of an operation shall be mapped to two dedicated composite data items; one for the request and one for the response. ⌋*(RTE00082)*

Each `ClientServerToSignalGroupMapping` references a unique `SystemSignalGroup` which holds all the signals related to the call or response.

For each ArgumentDataPrototype either a ClientServerPrimitiveTypeMapping or a ClientServerCompositeTypeMapping is defined which maps the operation arguments to SystemSignal elements.

[rte_sws_5056]⌈ If a ClientIdMapping element is configured it references the SystemSignal which holds the client identifier (see section 4.3.2.6.1). The RTE Generator shall utilize this SystemSignal as the client identifier. ⌋*(RTE00082)*

[rte_sws_5057]⌈ If a SequenceCounterMapping element is configured it references the SystemSignal which holds the Sequence Counter (see section 4.3.2.6.2). The RTE Generator shall utilize this SystemSignal as the SequenceCounter. ⌋*(RTE00082)*

[rte_sws_5058]⌈ If an ApplicationErrorMapping element is configured it references the SystemSignal which holds the ApplicationErrors (see section 5.2.6.8). The RTE Generator shall utilize this SystemSignal to transmit the ApplicationErrors. ⌋*(RTE00082)*

There might be configuration where no actual data is transferred between the client and the server (or vice versa). In this case a SystemSignalGroup shall be used with an update bit defined in System Description. In this case at least one SystemSignal is required to be present in the SystemSignalGroup.

[rte_sws_5059]⌈ If no actual data is configured for a client server communication i.e. the applicable ClientServerToSignalGroupMapping owns only an emptySignal, the RTE shall send the SignalGroup to initiate the communication. ⌋*(RTE00082)*

4.3.2.7.2 Atomicity

The requirements for atomicity from Section 4.3.1.11.2 also apply for the composite data types described in Section 4.3.2.7.1.

4.3.2.7.3 Fault detection and reporting

Client Server communication may encounter interruption like:

- Buffer overflow at the server side.
- Communication interruption.
- Server might be inaccessible for some reason.

The client specifies a timeout that will expire in case the server or communication fails to complete within the specified time. The reporting method of an expired timeout depends on the communication attributes:

- If the C/S communication is synchronous the RTE returns `RTE_E_TIMEOUT` on the `Rte_Call` function (see chapter 5.6.13).
- If the C/S communication is asynchronous the RTE returns `RTE_E_TIMEOUT` on the `Rte_Result` function (see chapter 5.6.14).

In the case that RTE detects that the COM service is not available when forwarding signals to COM, the RTE returns `RTE_E_COM_STOPPED` on the `Rte_Call` (see chapter 5.6.13).

If the client still has an outstanding server invocation when the server is invoked again, the RTE returns `RTE_E_LIMIT` on the `Rte_Call` (see chapter 5.6.13).

In the absence of structural errors, application errors will be reported if present.

4.3.2.7.4 Asynchronous Client Server communication

Figure 4.38 shows a sequence diagram of how asynchronous client server communication may be implemented by RTE.

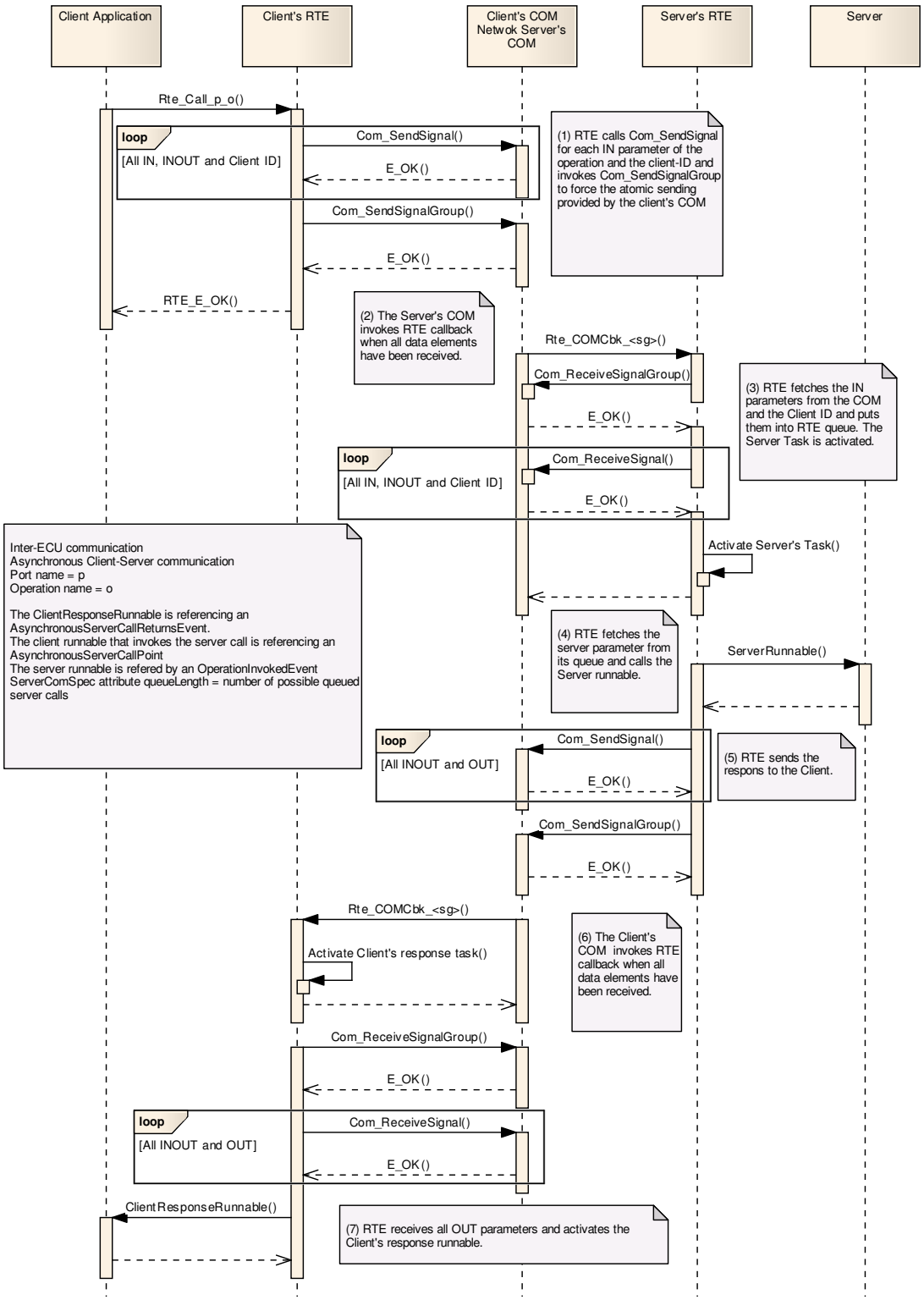


Figure 4.38: Client Server asynchronous

4.3.2.7.5 Synchronous Client Server communication

Figure 4.39 shows a sequence diagram of how synchronous client server communication may be implemented by RTE.

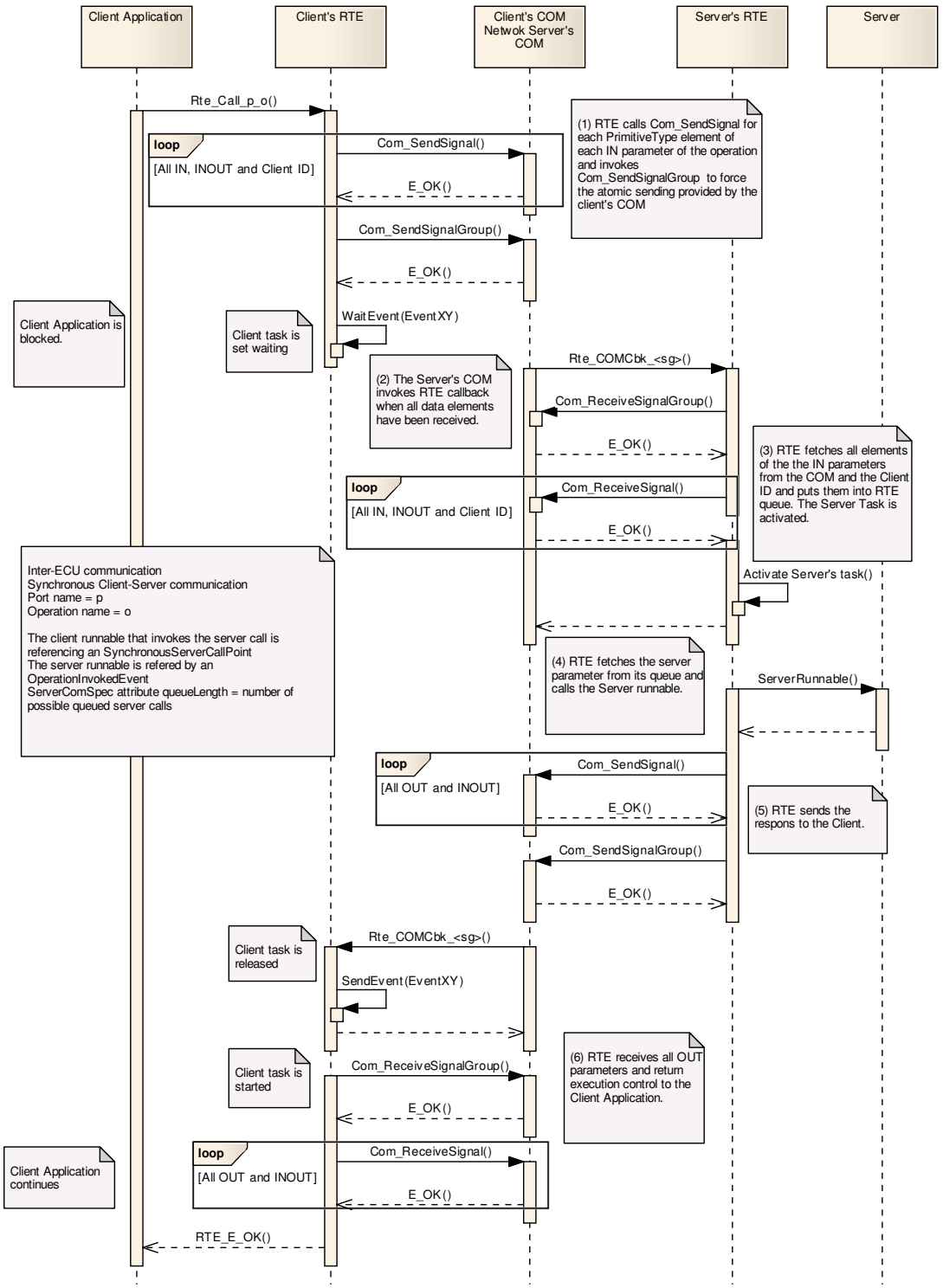


Figure 4.39: Client Server synchronous

4.3.3 SWC internal communication

4.3.3.1 Inter Runnable Variables

Sender/Receiver and Client/Server communication through AUTOSAR ports are the model for communication between AUTOSAR SW-Cs.

For communication between Runnables inside of an AUTOSAR SW-C the AUTOSAR SW-C Template [2] establishes a separate mechanism. Non-composite AUTOSAR SW-C can reserve InterRunnableVariables which can only be accessed by the Runnables of this one AUTOSAR SW-C. The Runnables might be running in the same or in different task contexts. Read and write accesses are possible.

[rte_sws_3589]⌈ The RTE has to support *Inter Runnable Variable*s for single and multiple instances of AUTOSAR SW-Cs. ⌋*(RTE00142)*

[rte_sws_7187]⌈ The generated RTE shall initialize a defined `implicitInterRunnableVariable` and `explicitInterRunnableVariable` according the `ValueSpecification` of the `VariableDataPrototype` defining the `implicitInterRunnableVariable` respectively `explicitInterRunnableVariable` if the general initialization conditions in rte_sws_7046 are fulfilled. ⌋*(RTE00142)*

InterRunnableVariables have a behavior corresponding to Sender/Receiver communication *between* AUTOSAR SW-Cs (or rather between Runnables of different AUTOSAR SW-Cs).

But why not use Sender/Receiver communication directly instead? Purpose is data encapsulation / data hiding. Access to InterRunnableVariables of an AUTOSAR SW-C from other AUTOSAR SWCs is not possible and not supported by RTE. InterRunnableVariable content stays SW-C internal and so no other SW-C can use. Especially not misuse it without understanding how the data behaves.

Like in Sender/Receiver (S/R) communication between AUTOSAR SW-Cs two different behaviors exist:

1. *Inter Runnable Variable*s with *implicit* behavior (`implicitInterRunnableVariable`)
   This behavior corresponds with `VariableAccess`es in the `dataReadAccess` and `dataWriteAccess` roles of Sender/Receiver communication and is supported by *implicit S/R API* in this specification.

   Note:
   If a `VariableAccess` in the `writtenLocalVariable` role referring to a `VariableDataPrototype` in the `implicitInterRunnableVariable` role is specified for a certain interrunnable variable, but no RTE API for implicit write of this interrunnable variable is called during an execution of the runnable, an undefined value is written back when the runnable terminates.

   For more details see section 4.2.5.6.1.
   For APIs see sections 5.6.23 and 5.6.24.

Note 2:
As for the Implicit Sender/Receiver communication, the implicit concept for InterRunnableVariables implies that the runnable does terminate. For runnable entities of category 2, the behavior is guaranteed only if it has a finite execution time. A category 2 runnable that runs forever will not have its data updated.

2. *Inter Runnable Variable*s with *explicit* behavior (`explicitInterRunnableVariable`)
   This behavior corresponds with `VariableAccess`es in the `dataSendPoint`, `dataReceivePointByValue`, or `dataReceivePointByArgument` roles of Sender/Receiver communication and is supported by *explicit S/R API* in this specification.

   For more details see section 4.2.5.6.2
   For APIs see sections 5.6.25 and 5.6.26.

4.3.4 Inter-Partition communication

Partitions are used to decompose an ECU into functional units. Partitions can contain both SW-Cs and BSW modules. The partitioning is done to protect the software contained in the partitions against each other or to increase the performance by running the partitions on different cores of a multi core controller.

Since the partitions may be separated by core boundaries or memory boundaries and since the partitions can be stopped and restarted independently, the observable behavior to the SW-Cs for the communication between different partitions is rather similar to the inter ECU communication than to the intra partition communication. The RTE needs to use special mechanisms to communicate from one partition to another.

Like for the inter ECU communication, inter partition communication uses the connectionless communication paradigm. This means, that a send operation is successful for the sender, even if the receiving partition is stopped. A receiver will only, by means of a timeout, be notified if the partition of the sender is stopped.

Unlike most basic software, the RTE does not have a main processing function. The execution logic of the RTE is contained in the generated task bodies, the wrapper code around the runnables whose execution RTE manages.

As the tasks that contain the SW-Cs runnables are uniquely assigned to partitions (see page 11EER of [16]), the execution logic of the RTE is split among the partitions. It can not be expected that the RTE generated wrapper code running in one partition can directly access the memory objects assigned to the RTE part of another partition.

In this sense, there is one RTE per partition, that contains runnable entities.

Still, RTE is responsible to support the communication between SW-Cs allocated to the different partitions. According to the AUTOSAR software layered architecture [], RTE shall be independent of the micro controller architecture. AUTOSAR supports a wide variety of multi core and memory protection architectures.

[rte_sws_2734]⌈ The RTE generator shall support a mode in which the generated code is independent of the micro controller. ⌋*(BSW161)*

It can not be generally assumed that a cache coherent, shared memory is available for the communication between partitions. Direct memory access and function calls across partition boundaries are generally not possible. In the extreme case, communication might even be limited to a message passing interface.

To allow memory protection and multi core support in spite of rte_sws_2734, the AUTOSAR OS provides a list of mechanisms, that can be used for the communication across cores (see [12]). Especially, the IOC has been designed to support the communication needs of RTE in a way that should not introduce additional run time overhead.

The following sections describe the use of some OS mechanisms that are designed for inter partition communication.

4.3.4.1 Inter partition data communication using IOC

The general idea to allow the data communication between partitions in a most efficient way and still be independent of the micro controller implementation is to take the buffers and queues from the intra partition communication case and replace them with so called IOC communication objects in the inter partition communication case.

In the ideal case, the access macros to the IOC communication object resolve to a direct access to shared memory.

The IOC (Inter OS-Application Communication) is a feature of the AUTOSAR OS, which provides a data oriented communication mechanism between partitions. The IOC provides communication buffers, queues, and protected access functions/macros to these buffers that can be used from any pre-configured partitions concurrently.

The IOC offers communication of data to another core or between memory protected partitions with guarantee of data consistency.

All data communications including the passing of parameters and return values in client server communication, can be implemented by using the IOC. The basic principle for using the IOC is to replace the RTE internal communication buffers by IOC buffers.

The IOC supports 1:1 and N:1 communication. For 1:N communication, N IOC communication objects have to be used. The IOC is configured and provides generated APIs for each IOC communication object. In case of N:1 communication, each sender has a separate API.

The IOC API is not reentrant.

[rte_sws_2737]⌈ RTE shall prevent concurrent access to the same IOC API from different `ExecutableEntity` `execution-instances`. ⌋*()*

The IOC will use the appropriate mechanism to communicate between the partitions, whether it requires communicating with another core, communicating with a partition with a different level of trust, or communicating with another memory partition.

The IOC channels are configured in the OS Configuration. Their configurations shall be provided as inputs for the RTE generator when the external configuration switch `strictConfigurationCheck` rte_sws_5148 is set to true, and can be provided by the RTE Generator or RTE Configuration Editor when `strictConfigurationCheck` is set to false (see rte_sws_5150).

The IOC APIs use:

1. types declared by user on input to RTE (sender-receiver communication across OsApplication boudaries).
2. types created by RTE to collect client-server operation arguments into single data structure.

For the second item, RTE uses internal types that have to be described as ImplementationDataTypes (see rte_sws_8400).

The signaling between partitions is not covered by the IOC. The callbacks of IOC are in interrupt context and are mainly intended for direct use by BSW. For the signaling between partitions, RTE can use the activation of tasks or setting of events, see section 4.3.4.3.

[rte_sws_2736]⌈ The RTE shall not execute ExecutableEntities in the context of IOC callbacks. ⌋*()*

This is necessary to ensure that ExecutableEntities will not be executed in interrupt context or when a partition is terminated or restarted.

4.3.4.2 Accessing COM from slave core in multicore configuration

In case of a multi core configuration, if a software component on the slave core wants to send data to a software component on another ECU, the RTE has to send data from the slave core through the IOC to the master core which in turn calls the send API of COM. The same behavior is required for receive case where the master core is responsible for forwarding received COM data to slave core through IOC.

[rte_sws_8306]⌈ It is the RTEs responsibility to interact with COM whenever it is needed. ⌋*()*

This requires some special handling by the RTE since it implies, at least in the send case, the need of a scheduable entity to do the actual call of COM send API.

[rte_sws_8307]⌈ The RTE shall generate two (*BswSchedulableEntity*'s):

- `Rte_ComSendSignalProxyPeriodic`.
- `Rte_ComSendSignalProxyImmediate`.

`Rte_ComSendSignalProxyPeriodic` shall handle the sending of periodic signals and `Rte_ComSendSignalProxyImmediate` shall handle the sending of immediate signals. ⌋*()*

[rte_sws_8308]⌈ It shall be a possible to configure whether the return value of RTE APIs is based on RTE-IOC interaction or RTE-COM interaction using the configuration parameter `RteIocInteractionReturnValue`. A warning should preferably be issued in case RTE-COM interaction return value is chosen since that will cause major performance decrease. ⌋*()*

4.3.4.2.1 Example sequence diagrams of accessing COM from Slave core

Figure 4.40 shows a sequence diagram of how receive data through COM from slave core may be implemented by RTE.

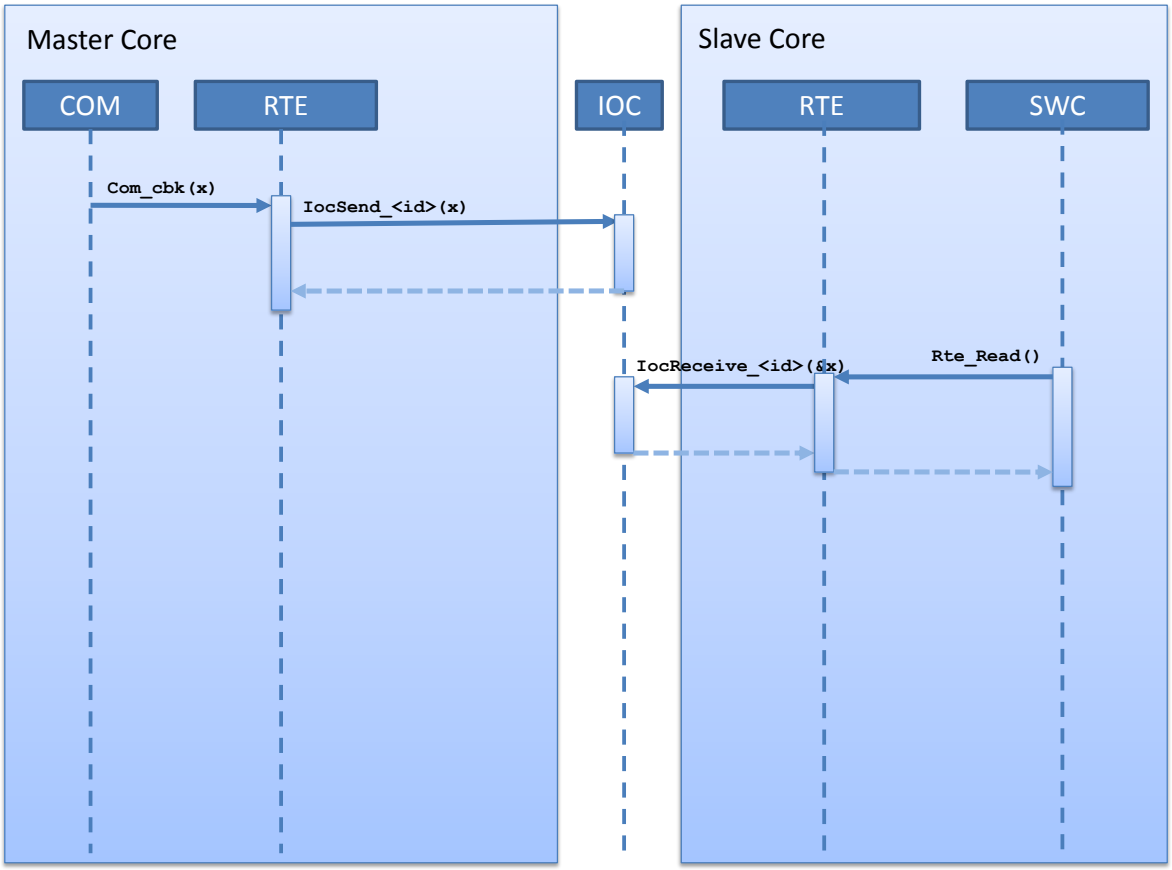


Figure 4.40: Receive data through COM from slave core

Figure 4.41 shows a sequence diagram of how send from COM to slave core may be implemented by RTE.

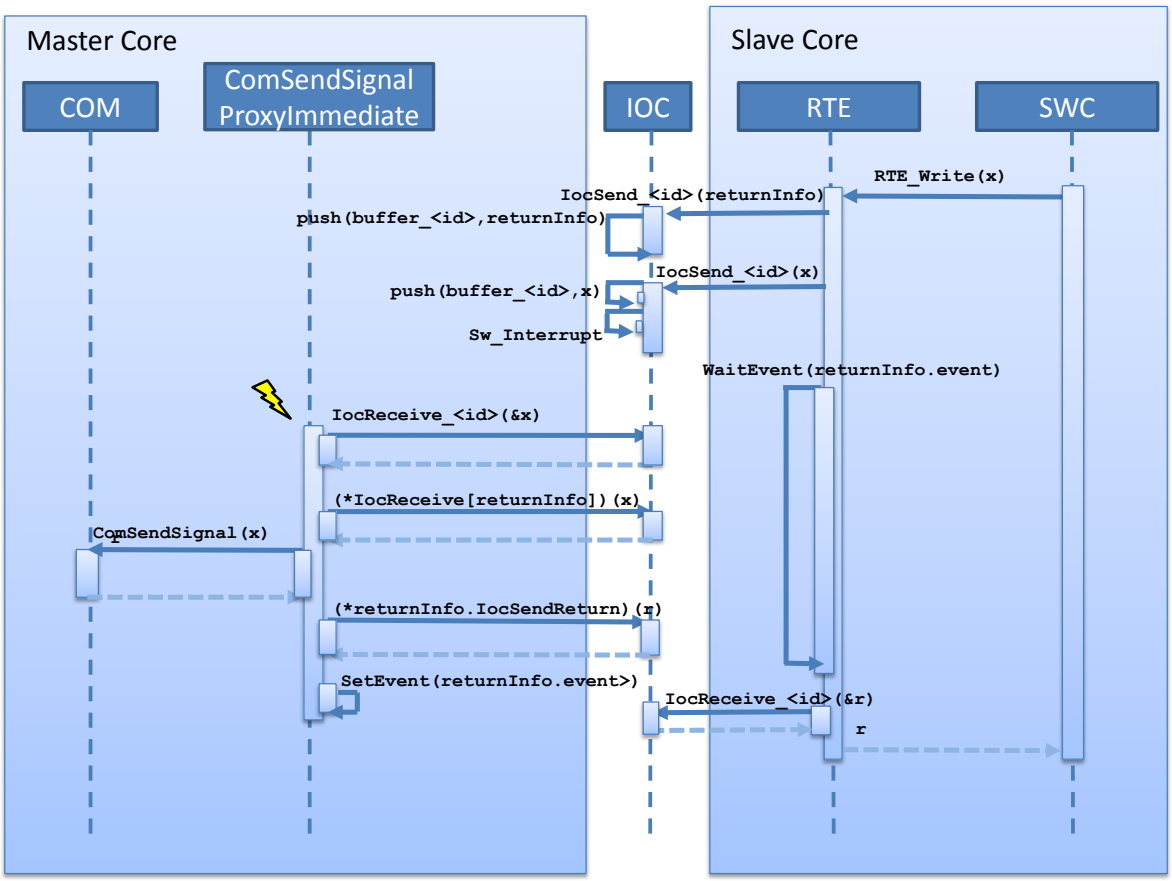


Figure 4.41: Send data through COM from slave core

Figure 4.42 shows a sequence diagram of how send from COM to slave core using return value based on RTE-IOC interaction may be implemented by RTE.

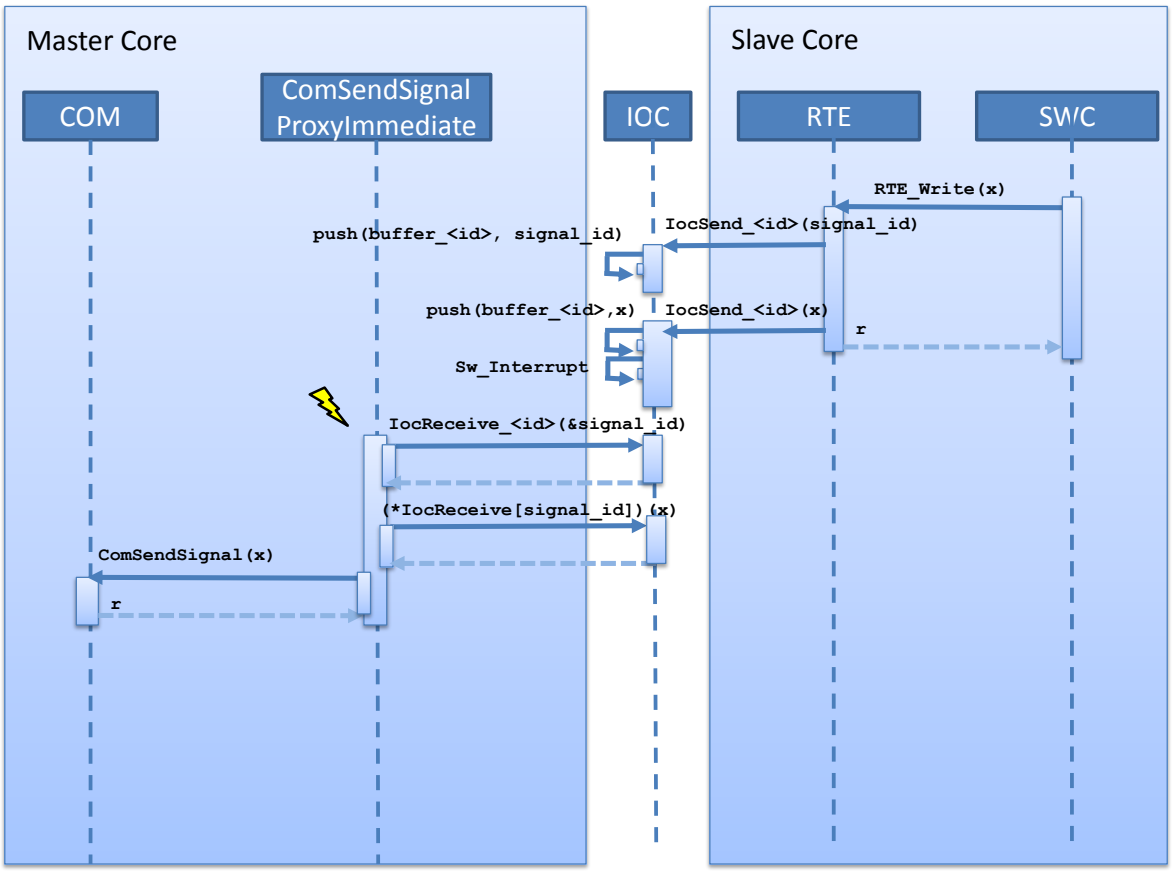


Figure 4.42: Send data through COM from slave core using return value based on RTE-IOC interaction

4.3.4.3 Signaling and control flow support for inter partition communication

The OS representation of a partition is an OS Application.

This is a (non-exhaustive) summary of OS features that can be used for signaling and control flow across partition boundaries:

- activation of tasks
- start and stop of schedule tables
- event signaling
- alarms
- spin locks (for inter core synchronization)

The following are not available for inter core signaling:

- OS Resource
- DisableAllInterrupts

For inter core synchronization, spin locks are provided. But, for efficiency reasons they should be used with care.

4.3.4.4 Trusted Functions

The call-trusted-function mechanism of AUTOSAR OS can be used in a memory protected controller to implement a function call from an untrusted to a trusted partition.

This Trusted Partition is a partition that has full access to the OS objects of other partitions on the same core. The Basic Software is assumed to reside in a trusted partition. It is assumed that the trusted partition cannot be terminated or restarted.

The typical use case for the call-trusted-function mechanism are AUTOSAR services which are usually provided by a client/server interface where the service side resides together with the basic software in the trusted partition.

Beware that this mechanism can not be used between two untrusted partitions or between cores.

The trusted functions are configured in the OS Configuration. Their configurations shall be provided as inputs for the RTE generator when the external configuration switch `strictConfigurationCheck` rte_sws_5148 is set to true, and can be provided by the RTE Generator or RTE Configuration Editor when `strictConfigurationCheck` is set to false (see rte_sws_5150).

[rte_sws_7606]⌈ Direct start of an ExecutableEntity execution-instance by the mean of a trusted function shall only be used for the start of an ExecutableEntity in the Trusted Partition. ⌋*(RTE00195, RTE00210)*

The OS ensures that the partition of the caller is not terminated or restarted when a trusted function is executed. If needed, the termination or restart of the caller's partition is delayed after the trusted function returns.

RTE has to ensure, that the OS does not kill an RTE-generated task due to stopping or restarting a partition while this task is executing a function call to BSW or to the software component of another partition when this call is not a pure function.

For this purpose, RTE can use either the OS mechanism of trusted function call, or it can allocate the server to a different task than the client.

[rte_sws_2761]⌈ In a partitioned system that supports stop or restart of partitions, the RTE shall not use a direct function call (without use of OS call trusted function) from a task of an untrusted partition to BSW or to the SW-C of another partition unless this is a pure function. ⌋*(RTE00196)*

Please note that rte_sws_2761 might require the use of OS call trusted function for a partitioned system even without memory protection.

4.3.4.5 Memory Protection and Pointer Type Parameters in RTE API

In a memory protected ECU, a SW-C from an untrusted partition might misuse the transition to the trusted context to modify memory in another partition. This can occur when a pointer to a different memory partition is passed from the untrusted partition to the trusted context. The RTE shall avoid this misuse by at least checking the validity of the address of the pointer, and, where possible, also checking the integrity of the associated memory object.

[rte_sws_2752]⌈ When a SW-C in an untrusted partition receives (OUT parameter) or provides (IN parameter with composite data type) an ArgumentDataPrototype or VariableDataPrototype, it hands over a pointer to a memory object to an RTE API. The RTE shall only forward this pointer to a trusted SW-C after it has checked that the whole memory object is owned by the caller's partition. ⌋*(RTE00210)*

[rte_sws_2753]⌈ When a SW-C in an untrusted partition passes an ArgumentDataPrototype or VariableDataPrototype, as a reference type to a SW-C in a trusted partition (`DATA_REFERENCE` as an IN parameter), the RTE shall only check that the caller's partition owns the start address of the referenced memory. ⌋*(RTE00210)*

Note to rte_sws_2753: The RTE only checks whether the start address referenced directly by the DataPrototypes belongs to the calling partition. Because the RTE is not aware of the semantic of the pointed reference, it cannot check if the referenced object is completely contained in the calling partition (e.g. the RTE does not know the size and does not know if the referenced object also contains references to other objects). The BSW is responsible to make sure that the referenced memory object does not cross memory section boundaries.

The OS API CheckTaskMemoryAccess can be used to fulfill rte_sws_2752 and rte_sws_2753.

4.3.5 PortInterface Element Mapping and Data Conversion

AUTOSAR supports the connection of an R-port to a P-port with an interface that is not compatible in the sense of the AUTOSAR compatibility rules. In addition, for sender-receiver communication it is possible to specify how data elements are represented given that the communication requires the usage of a dedicated communication bus. In these cases the generated RTE has to support the conversion and re-scaling of data.

4.3.5.1 PortInterface Element Mapping

Per default the `shortName`s of `PortInterface` elements are used to identify the matching element pairs of connected ports. In case of non fitting names — might be caused due to distributed development, off-the-shelf development, or re-use of software components — it is required to explicitly specify which `PortInterface` elements shall correlate. This is modelled with `PortInterfaceMapping`s. A connection of two ports can be associated with a set of `PortInterfaceMapping`s. If two ports are connected and a `PortInterfaceMapping` for the pair of interfaces of the two ports is associated with the connection, the interface elements are mapped and converted as specified in the `PortInterfaceMapping`. If no `PortInterfaceMapping` for the respective pair of interfaces is associated with the connection, the ordinary interface compatibility rules are applied.

The general approach is to perform the data conversion in the RTE of the ECU implementing the R-port. The reason for this design decision is that in case of 1:n sender-receiver communication it is inefficient to perform all the data conversions for the multiple receivers on the sender side and then send multiple sets of the same data just in different representations over the communication bus.

[rte_sws_3815]⌈ The RTE shall support the mapping of sender-receiver interfaces, parameter interfaces and non-volatile data interface elements. ⌋*(RTE00182)*

[rte_sws_3816]⌈ If a P-port specified by a `SenderReceiverInterface` or `NvDataInterface` is connected to an R-port with an incompatible interface and a `VariableAndParameterInterfaceMapping` for both interfaces is associated with the connection, the RTE of the ECU implementing the R-port shall map and convert the data elements of the sender's interface to the data elements of the receiver's interface. ⌋*(RTE00182)*

[rte_sws_7091]⌈ The RTE shall support the *Mapping of elements of composite data types* in the context of a mapping of `SenderReceiverInterface`, `NvDataInterface` or `ParameterInterface` elements. ⌋*(RTE00182, RTE00234)*

[rte_sws_7092]⌈ The RTE of the ECU implementing the R-port shall map and convert the composite data type elements of `DataPrototoype`s of the sender's interface to the composite data type elements of `DataPrototoype`s of the receiver's interface according the `SubElementMapping`

if a P-port specified by a SenderReceiverInterface, NvDataInterface or ParameterInterface is connected to an R-port with an incompatible interface and a VariableAndParameterInterfaceMapping exists for both interfaces and is associated with the connection and
the SubElementMapping maps composite data type elements of the provided interface to composite data type elements of the required interface. ⌋*(RTE00182, RTE00234)*

[rte_sws_7099]⌈ The RTE of the ECU implementing the R-port shall map and convert the composite data type elements of DataPrototoype of the sender's interface to the primitive DataPrototoype of the receiver's interface according the SubElementMapping
if a P-port specified by a SenderReceiverInterface, NvDataInterface or ParameterInterface is connected to a R-port with an incompatible interface and a VariableAndParameterInterfaceMapping exists for both interfaces and is associated with the connection and the SubElementMapping exclusively maps one composite data type element of the provided interface ⌋*(RTE00182, RTE00234)*

Please note that the DataPrototoypes of the provide port and DataPrototoypes of the require port might use exclusively ApplicationDataTypes, exclusively ImplementationDataTypes or both kinds of AutosarDataTypes in a mixed manner.

[rte_sws_3817]⌈ If a P-port specified by a SenderReceiverInterface or NvDataInterface is connected to an R-port with an incompatible interface and no VariableAndParameterInterfaceMapping for this pair of interfaces is associated with the connection, the RTE generator shall reject the input as an invalid configuration. ⌋*(RTE00182, RTE00018)*

[rte_sws_3818]⌈ The RTE shall support the mapping of client-server interface elements. ⌋*(RTE00182)*

[rte_sws_3819]⌈ If a P-port specified by a ClientServerInterface is connected to an R-port with an incompatible interface and a ClientServerInterfaceMapping for both interfaces is associated with the connection, the RTE of the ECU implementing the R-port, i. e. the client, shall map the operation and map and convert the operation arguments of the client's interface to the operation arguments of the server's interface. ⌋*(RTE00182)*

[rte_sws_3820]⌈ If a P-port specified by a ClientServerInterface is connected to an R-port with an incompatible interface and no ClientServerInterfaceMapping for this pair of interfaces is associated with the connection, the RTE generator shall reject the input as an invalid configuration. ⌋*(RTE00182, RTE00018)*

[rte_sws_3821]⌈ The RTE shall support the mapping of ModeSwitchInterface elements. ⌋*(RTE00182)*

[rte_sws_3822]⌈ If a P-port specified by a ModeSwitchInterface is connected to an R-port with an incompatible interface and a ModeInterfaceMapping for both interfaces is associated with the connection, the RTE of the ECU implementing the

R-port shall map and convert the mode elements of the sender's interface to the mode elements of the receiver's interface. ⌋*(RTE00182)*

[rte_sws_3823]⌈ If a P-port specified by a `ModeSwitchInterface` is connected to an R-port with an incompatible interface and no `ModeInterfaceMapping` for this pair of interfaces is associated with the connection, the RTE generator shall reject the input as an invalid configuration. ⌋*(RTE00182, RTE00018)*

[rte_sws_3824]⌈ The RTE shall support the mapping of trigger interface elements. ⌋*()*

[rte_sws_3825]⌈ If a P-port specified by a `TriggerInterface` is connected to an R-port with an incompatible interface and a `TriggerInterfaceMapping` for both interfaces is associated with the connection, the RTE of the ECU implementing the R-port shall map the trigger of the sender's interface to the trigger of the receiver's interface. ⌋*()*

[rte_sws_3826]⌈ If a P-port specified by a `TriggerInterface` is connected to an R-port with an incompatible interface and no `TriggerInterfaceMapping` for this pair of interfaces is associated with the connection, the RTE generator shall reject the input as an invalid configuration. ⌋*(RTE00018)*

In order to generate the RTE for the ECU implementing the R-ports, the RTE generator has to know the interfaces of the P-ports that are connected over the bus. This information is provided in the ECU extract via the `networkRepresentationProps` (see section 4.3.5.2) specified at the `ISignal` representing the data element.

4.3.5.2 Network Representation

For sender-receiver communication it is possible to specify how data elements are represented given that the communication requires the usage of a dedicated communication bus. For this purpose `networkRepresentationProps` can be specified at the `ISignal`, describing the representation of the data element on the communication bus via the attributes `compuMethod` and `baseType`.

[rte_sws_3827]⌈ If a network representation is specified for a data element of a sender-receiver P-port, the RTE of the transmitting ECU shall perform the conversion of the data element that has to be sent over a communication bus to the representation specified by the `baseType` and `compuMethod` of the `networkRepresentationProps` of the respective `ISignal`. The converted data shall be passed to COM. ⌋*(RTE00181)*

[rte_sws_3828]⌈ If a network representation is specified for a data element of a sender-receiver R-port, the RTE of the receiving ECU shall perform the conversion of the data element that is received over a communication bus from the representation specified by the `baseType` and `compuMethod` of the `networkRepresentationProps` of the respective `ISignal` to the data element's application data type. In this case rte_sws_3816 shall not be applied. ⌋*(RTE00181)*

4.3.5.3 Data Conversion

[rte_sws_3829]⌈ The RTE shall support the conversion of an identical or linear scaled data representation to another identical or linear scaled data representation. In this context, the term "linear scaled data representation" also includes floating-point data representations. ⌋*(RTE00182)*

[rte_sws_3830]⌈ The RTE shall support the conversion of a texttable data representation (enumeration) to another texttable data representation. ⌋*(RTE00182)*

[rte_sws_3855]⌈ The RTE shall support the conversion of a mixed linear scaled and texttable data representation to another mixed linear scaled and texttable data representation. ⌋*(RTE00182)*

[rte_sws_3856]⌈ The RTE shall support the conversion between a texttable data representation (enumeration) and a mixed linear scaled and texttable data representation. In this case only the enumeration part of the data representation shall be converted, the linear scaled part shall be handled as out of range data. ⌋*(RTE00182)*

[rte_sws_3857]⌈ The RTE shall support the conversion between an identical or linear scaled data representation and a mixed linear scaled and texttable data representation. A scale with a `COMPU-CONST` shall be handled as out of range data if the mapping to a value is not defined by a `TextTableMapping`. ⌋*(RTE00182)*

[rte_sws_3860]⌈ The RTE shall support the conversion of composite data representations. In this case, the respective requirements rte_sws_3829, rte_sws_3830, rte_sws_3855, rte_sws_3856, rte_sws_3857, rte_sws_3831, rte_sws_3832, and rte_sws_3833 are applicable to the individual composite elements. ⌋*(RTE00182)*

[rte_sws_3831]⌈ The RTE generator shall reject any input that requires a conversion which is not supported according to rte_sws_3829, rte_sws_3830, rte_sws_3855, rte_sws_3856, or rte_sws_3860 as an invalid configuration. ⌋*(RTE00182, RTE00018)*

[rte_sws_3832]⌈ For the conversion between two data representations with linear scaling described either by an `ApplicationDataType` or a combination of `BaseType` and `CompuMethod` (used for the specification of the network representation at the `ComSpec` respectively the `ISignal`) the RTE generator shall derive the data conversion code automatically from the referred `CompuMethod`s of the two representations. In this context the scaling of a data representation is linear if the referred `CompuMethod` is of category `IDENTICAL`, `LINEAR`, or `SCALE_LINEAR_AND_TEXTTABLE`. In case of a `CompuMethod` of category `SCALE_LINEAR_AND_TEXTTABLE` this requirement applies to the linear scaled part only. The conversion shall only be performed if both `CompuMethod`s either refer to compatible `Unit`s or to `Unit`s referring to identical definitions of `PhysicalDimension` (i. e. all `PhysicalDimension` attributes are identical). ⌋*(RTE00182)*

For a linear conversion the linear conversion factor can be calculated out of the `factorSiToUnit` and `offsetSiToUnit` attributes of the referred Units and the `CompuRationalCoeffs` of a `compuInternalToPhys` of the referred `CompuMethod`s.

Example 4.6

A software component `SwcA` on an ECU `EcuA` sends a data element `u` of an uint16 type `t_VoltageAtSender` via its port `SenderPort`. The referenced `CompuMethod` is `cm_VoltageAtSender`, describing a fixpoint representation with offset 0 and LSB $\frac{1}{4} = 2^{-2}$. The port `SenderPort` is connected to the port `ReceiverPort` of a software component `SwcB` that is deployed on a different ECU `EcuB`. The sent data element `u` is mapped to a data element `u` of an uint16 type `t_VoltageAtReceiver` on the receiving side that references a `CompuMethod` named `cm_VoltageAtReceiver`. `cm_VoltageAtReceiver` describes a fixpoint representation with offset $\frac{16}{8} = 2$ and LSB $\frac{1}{8} = 2^{-3}$. For transportation over the bus a `networkRepresentation` that references an uint8 type `t_VoltageOnNetwork` is specified, using a fixpoint representation described by the `CompuMethod cm_VoltageOnNetwork` with offset $\frac{1}{2} = 0.5$ and LSB $\frac{1}{2} = 2^{-1}$.

Definition of the `CompuMethod`s in *XML*:

```
<COMPU-METHOD>
  <SHORT-NAME>cm_VoltageAtSender</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR><V>0</V><V>1</V></COMPU-NUMERATOR>
          <COMPU-DENOMINATOR><V>4</V></COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>

<COMPU-METHOD>
  <SHORT-NAME>cm_VoltageAtReceiver</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR><V>16</V><V>1</V></COMPU-NUMERATOR>
          <COMPU-DENOMINATOR><V>8</V></COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
```

```
</COMPU-METHOD>

<COMPU-METHOD>
  <SHORT-NAME>cm_VoltageOnNetwork</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR><V>1</V><V>1</V></COMPU-NUMERATOR>
          <COMPU-DENOMINATOR><V>2</V></COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

Implementation of `Rte_Send` on the sending ECU `EcuA`:

```
Std_ReturnType
Rte_Send_SwcA_SenderPort_u(t_voltageAtSender u)
{
  ...
  /*
  u_NetworkRepresentation
   = ((u * LSB_sender + off_sender) - off_network) / LSB_network
   = ((u / 4          + 0         ) - 0.5         ) * 2
   =  (u / 2                      ) - 1
  */
  u_NetworkRepresentation = (uint8) ((u >> 1) - 1);
  ...
}
```

Implementation of `Rte_Receive` on the receiving ECU `EcuB`:

```
Std_ReturnType
Rte_Receive_SwcB_ReceiverPort_u(t_voltageAtReceiver * u)
{
  ...
  /*
  *u
  *u = ((u_NetworkRepresentation * LSB_network + off_network)
        - off_receiver) / LSB_receiver
     = ((u_NetworkRepresentation / 2           + 0.5         )
        - 2           ) * 8
     =  (u_NetworkRepresentation * 4           + 4           )
        - 16
     =   u_NetworkRepresentation * 4           - 12
  */
  *u = (uint16) ((u_NetworkRepresentation << 2) - 12);
```

```
16      ...
17  }
```

The intention of this specification is not to describe any mechanism that supports the generation of identical conversion code for each implementation of an RTE generator. Even if the generated C code for the conversion would be the same, the numerical result of the conversion still depends on the microcontroller target and the compiler.

Strategies how to handle the conversion of values that are out of range of the target representation are described in section 4.3.5.4.

[rte_sws_3833]⌈ For the conversion between two texttable data representations (enumerations) described either by an `ApplicationDataType` or an `ImplementationDataType` (used for the specification of the network representation) the RTE generator shall generate the data conversion code according to the `TextTableMapping`. This requirement also applies to the texttable part of a mixed linear scaled and texttable data representation. ⌋*(RTE00182)*

4.3.5.4 Range Checks during Runtime

A software component might try to send a value that is outside the range that is specified at a `dataElement` or `ISignal`. In case of different ranges the result of a data conversion might also be a value that is out of range of the target representation. For a safe handling of these use cases the RTE provides range checks during runtime. For an overview see figure 4.43.

[rte_sws_8024]⌈ Range checks during runtime shall occur after data invalidation, i.e. first the handleNeverReceived check, then the invalidation check and lastly the range check shall be effected. ⌋*(RTE00180)*

[rte_sws_3861]⌈ The range check is intended to be performed according to the following rule: If a upper/lower limit is specified at the `DataConstr`, this value shall be taken for the range check. If it is not specified at the `DataConstr`, the highest/lowest representable value of the datatype shall be used. ⌋*(RTE00180)*

Whether a range check is required is specified in case of intra ECU communication at the `handleOutOfRange` attribute of the respective `SenderComSpec` or `ReceiverComSpec` and in case of inter ECU communication at the `handleOutOfRange` attribute of `ISignalProps` of the sending or receiving `ISignal`.

Range checks at sender's side

Range checks during runtime for intra ECU communication at the sender's side are described in the following requirements:

[rte_sws_8026]⌈ The RTE shall implement a range check of sent data in the sending path of a particular component if the `handleOutOfRange` is defined at the `SenderComSpec` and has any value other than `none`. In this case all receivers receive the value after the range check was applied. ⌋*(RTE00180)*

[rte_sws_8039]⌈ The RTE shall use the preceding limits (rte_sws_7196) from the `DataPrototype` in the `pPort` for the range check of sent data in the sending path of a particular component if the `handleOutOfRange` is defined at the `SenderComSpec`. ⌋*(RTE00180)*

[rte_sws_3839]⌈ If for a `dataElement` to be sent a `SenderComSpec` with `handleOutOfRange=ignore` is provided, a range check shall be implemented in the sending component. If the value is out of bounds, the sending of the `dataElement` shall not be propagated. This means for a non-queued communication that the last valid value will be propagated and for a queued communication that no value will be enqueued.

In case of a composite datatype the sending of the whole `dataElement` shall not be propagated, if any of the composite elements is out of bounds. ⌋*(RTE00180)*

[rte_sws_3840]⌈ If for a `dataElement` to be sent a `SenderComSpec` with `handleOutOfRange=saturate` is provided, a range check shall be implemented in the sending component. If the value is out of bounds, the value actually sent shall be set to the lower respectively the upper limit.

In case of a composite datatype each composite element whose actual value is out of bounds shall be saturated. ⌋*(RTE00180)*

[rte_sws_3841]⌈ If for a `dataElement` to be sent a `NonqueuedSenderComSpec` with `handleOutOfRange=default` is provided, a range check shall be implemented in the sending component. If the value is out of bounds and the `initValue` is not equal to the `invalidValue`, the value actually sent shall be set to the `initValue`.

In case of a composite datatype each composite element whose actual value is out of bounds shall be set to the `initValue`. ⌋*(RTE00180)*

[rte_sws_3842]⌈ If for a `dataElement` to be sent a `NonqueuedSenderComSpec` with `handleOutOfRange=invalid` is provided, a range check shall be implemented in the sending component. If the value is out of bounds, the value actually sent shall be set to the `invalidValue`.

In case of a composite datatype each composite element whose actual value is out of bounds shall be set to the `invalidValue`. ⌋*(RTE00180)*

[rte_sws_3843]⌈ If for a `dataElement` to be sent a `QueuedSenderComSpec` with `handleOutOfRange` set to `default` or `invalid` is provided, the RTE generator shall reject the input as an invalid configuration, since for a `QueuedSenderComSpec`

the attribute initValue is not defined (see SW-C Template [2]) and data invalidation is not supported (see rte_sws_5033). ⌋*(RTE00180)*

Range checks during runtime for inter ECU communication at the sender's side are described in the following requirements:

[rte_sws_8027]⌈ The RTE shall implement a range check of sent data in the sending path of a particular signal if the handleOutOfRange is defined at the ISignalProps and has any value other than none. In this case only receivers of the specific ISignal receive the value after the range check was applied. ⌋*(RTE00180)*

[rte_sws_8040]⌈ The RTE shall use the limits from the ISignal for the range check of sent data in the sending path of a particular signal if the handleOutOfRange is defined at the ISignalProps. ⌋*(RTE00180)*

[rte_sws_8030]⌈ If for an ISignal to be sent an ISignalProps with handleOutOfRange=ignore is provided, a range check shall be implemented in the sending signal. If the value is out of bounds, the sending of the ISignal shall not be propagated. In this case the RTE shall behave as if no sending occurred. ⌋*(RTE00180)*

[rte_sws_8031]⌈ If for an ISignal to be sent an ISignalProps with handleOutOfRange=saturate is provided, a range check shall be implemented in the sending signal. If the value is out of bounds, the value actually sent shall be set to the lower respectively the upper limit. ⌋*(RTE00180)*

[rte_sws_8032]⌈ If for an ISignal to be sent an ISignalProps with handleOutOfRange=default is provided, a range check shall be implemented in the sending signal. If the value is out of bounds and the initValue is not equal to the invalidValue, the value actually sent shall be set to the initValue. ⌋*(RTE00180)*

[rte_sws_8033]⌈ If for an ISignal to be sent an ISignalProps with handleOutOfRange=invalid is provided, a range check shall be implemented in the sending signal. If the value is out of bounds, the value actually sent shall be set to the invalidValue. ⌋*(RTE00180)*

Range checks at receiver's side

Range checks during runtime for intra ECU communication at the receiver's side are described in the following requirements:

[rte_sws_8028]⌈ The RTE shall implement a range check in the receiving path of a particular component if the handleOutOfRange is defined at the ReceiverComSpec and has any value other than none. In this case the range check applies only for data received by the particular component. ⌋*(RTE00180)*

[rte_sws_8041]⌈ The RTE shall use the preceding limits (rte_sws_7196) from the DataPrototype in the rPort for the range check of received data in the receiv-

ing path of a particular component if the handleOutOfRange is defined at the ReceiverComSpec. ⌋*(RTE00180)*

[rte_sws_3845]⌈ If for a dataElement to be received a ReceiverComSpec with handleOutOfRange=ignore is provided, a range check shall be implemented in the receiving component. If the value is out of bounds, the reception of the dataElement shall not be propagated. This means for a non-queued communication that the last valid value will be propagated and for a queued communication that no value will be enqueued.

If the value of the received dataElement is out of bounds and a NonqueuedReceiverComSpec with handleOutOfRangeStatus=indicate is provided, the return value of the RTE shall be RTE_E_OUT_OF_RANGE.

In case of a composite datatype the reception of the whole dataElement shall not be propagated, if any of the composite elements is out of bounds. If the handleOutOfRangeStatus attribute is set to indicate, the return value of the RTE shall be RTE_E_OUT_OF_RANGE. ⌋*(RTE00180)*

[rte_sws_3846]⌈ If for a dataElement to be received a ReceiverComSpec with handleOutOfRange=saturate is provided, a range check shall be implemented in the receiving component. If the value is out of bounds, the value actually received shall be set to the lower respectively the upper limit.

If the value of the received dataElement is out of bounds and a NonqueuedReceiverComSpec with handleOutOfRangeStatus=indicate is provided, the return value of the RTE shall be RTE_E_OUT_OF_RANGE.

In case of a composite datatype each composite element whose actual value is out of bounds shall be saturated. If the handleOutOfRangeStatus attribute is set to indicate, the return value of the RTE shall be RTE_E_OUT_OF_RANGE, if any of the composite elements is out of bounds. ⌋*(RTE00180)*

[rte_sws_3847]⌈ If for a dataElement to be received a NonqueuedReceiverComSpec with handleOutOfRange=default is provided, a range check shall be implemented in the receiving component. If the value is out of bounds and the initValue is not equal to the invalidValue, the value actually received shall be set to the initValue.

If the value of the received dataElement is out of bounds and a NonqueuedReceiverComSpec with handleOutOfRangeStatus=indicate is provided, the return value of the RTE shall be RTE_E_OUT_OF_RANGE.

In case of a composite datatype each composite element whose actual value is out of bounds shall be set to the initValue. If the handleOutOfRangeStatus attribute is set to indicate, the return value of the RTE shall be RTE_E_OUT_OF_RANGE, if any of the composite elements is out of bounds. ⌋*(RTE00180)*

[rte_sws_3848]⌈ If for a dataElement to be received a NonqueuedReceiverComSpec with handleOutOfRange=invalid is provided, a range check shall be imple-

mented in the receiving component. If the value is out of bounds, the value actually received shall be set to the invalidValue.

If the value of the received dataElement is out of bounds and a ReceiverComSpec with handleOutOfRangeStatus=indicate is provided, the return value of the RTE shall be RTE_E_INVALID.

In case of a composite datatype each composite element whose actual value is out of bounds shall be set to the invalidValue. If the handleOutOfRangeStatus attribute is set to indicate, the return value of the RTE shall be RTE_E_INVALID, if any of the composite elements is out of bounds. ⌋*(RTE00180)*

[rte_sws_8016]⌈ If for a dataElement to be received a ReceiverComSpec with handleOutOfRange=externalReplacement is provided, a range check shall be implemented in the receiving component. If the value is out of bounds, the value actually received shall be replaced by the value sourced from the ReceiverComSpec.externalReplacement (e.g. constant, NVRAM parameter).

If the value of the received dataElement is out of bounds and a NonqueuedReceiverComSpec with handleOutOfRangeStatus=indicate is provided, the return value of the RTE shall be RTE_E_OUT_OF_RANGE.

In case of a composite datatype the value actually received shall be completely replaced by the external value, if any of the composite elements is out of bounds. If the handleOutOfRangeStatus attribute is set to indicate, the return value of the RTE shall be RTE_E_OUT_OF_RANGE. ⌋*(RTE00180)*

[rte_sws_3849]⌈ If for a dataElement to be received a QueuedReceiverComSpec with handleOutOfRange set to default or invalid is provided, the RTE generator shall reject the input as an invalid configuration, since for a QueuedReceiverComSpec the attribute initValue is not defined (see SW-C Template [2]) and data invalidation is not supported (see rte_sws_5033). ⌋*(RTE00180)*

[rte_sws_8025]⌈ If for a dataElement to be received a QueuedReceiverComSpec is provided and the handleOutOfRangeStatus attribute is set to indicate, the RTE generator shall reject the input as an invalid configuration. ⌋*(RTE00180)*

Range checks during runtime for inter ECU communication at the receiver's side are described in the following requirements:

[rte_sws_8029]⌈ The RTE shall implement a range check in the receiving path of a particular signal if the handleOutOfRange is defined at the ISignalProps and has any value other than none. In this case all receivers of the specific ISignal on that ECU receive the value after the range check was applied. ⌋*(RTE00180)*

[rte_sws_8042]⌈ The RTE shall use the limits from the ISignal for the range check of received data in the receiving path of a particular signal if the handleOutOfRange is defined at the ISignalProps. ⌋*(RTE00180)*

[rte_sws_8034]⌈ If for an ISignal to be received an ISignalProps with handleOutOfRange=ignore is provided, a range check shall be implemented in the receiv-

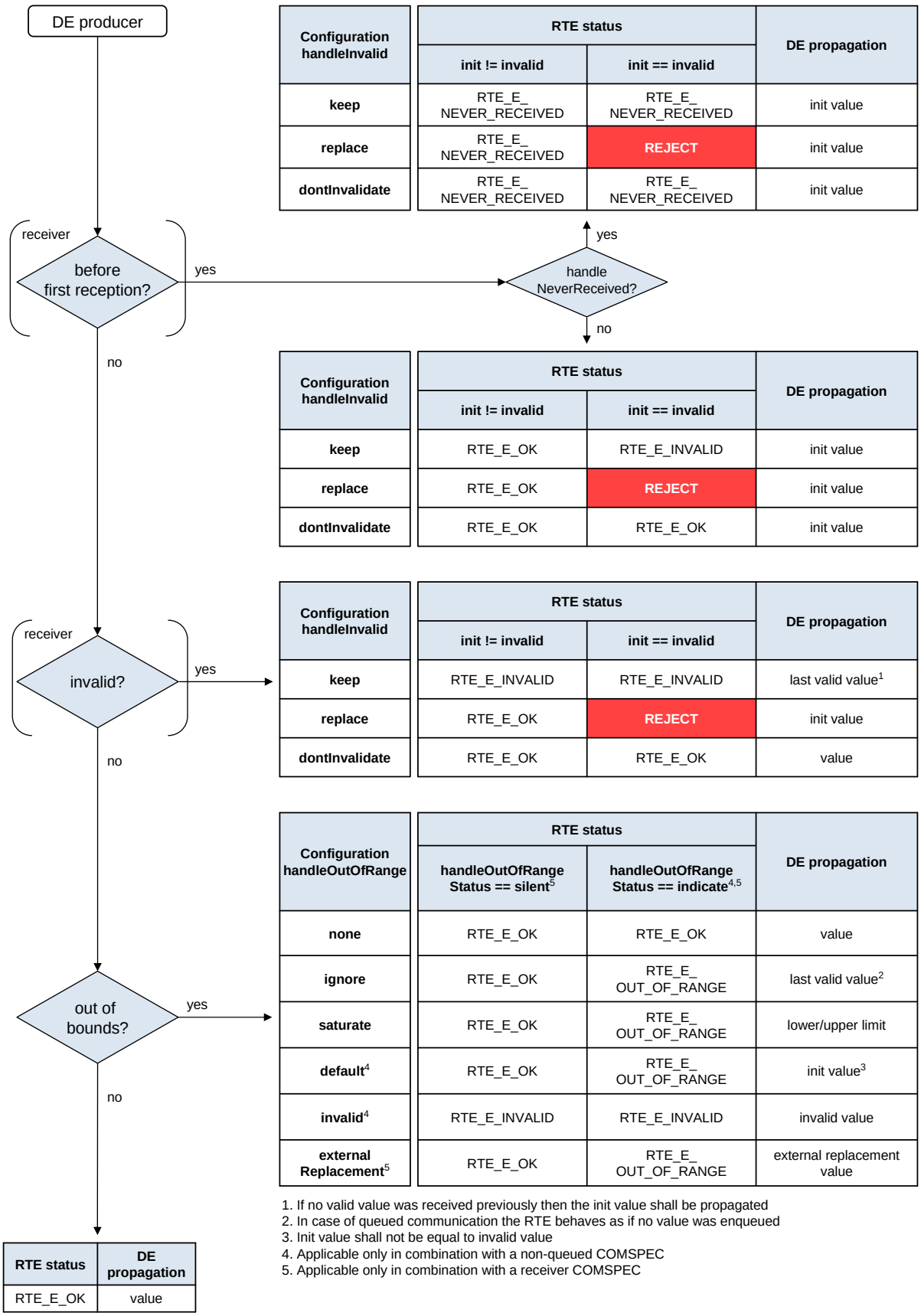
ing signal. If the value is out of bounds, the reception of the ISignal shall not be propagated. In this case the RTE shall behave as if no reception occurred. ⌋*(RTE00180)*

[rte_sws_8035]⌈ If for an ISignal to be received an ISignalProps with handleOutOfRange=saturate is provided, a range check shall be implemented in the receiving signal. If the value is out of bounds, the value actually received shall be set to the lower respectively the upper limit. ⌋*(RTE00180)*

[rte_sws_8036]⌈ If for an ISignal to be received an ISignalProps with handleOutOfRange=default is provided, a range check shall be implemented in the receiving signal. If the value is out of bounds and the initValue is not equal to the invalidValue, the value actually received shall be set to the initValue. ⌋*(RTE00180)*

[rte_sws_8037]⌈ If for an ISignal to be received an ISignalProps with handleOutOfRange=invalid is provided, a range check shall be implemented in the receiving signal. If the value is out of bounds, the value actually received shall be set to the invalidValue. ⌋*(RTE00180)*

[rte_sws_8038]⌈ If for an ISignal to be received an ISignalProps with handleOutOfRange=externalReplacement is provided, a range check shall be implemented in the receiving signal. If the value is out of bounds, the value actually received shall be replaced by the value sourced from the ReceiverComSpec.externalReplacement (e.g. constant, NVRAM parameter). ⌋*(RTE00180)*



Configuration handleInvalid	RTE status		DE propagation
	init != invalid	init == invalid	
keep	RTE_E_ NEVER_RECEIVED	RTE_E_ NEVER_RECEIVED	init value
replace	RTE_E_ NEVER_RECEIVED	REJECT	init value
dontInvalidate	RTE_E_ NEVER_RECEIVED	RTE_E_ NEVER_RECEIVED	init value

Configuration handleInvalid	RTE status		DE propagation
	init != invalid	init == invalid	
keep	RTE_E_OK	RTE_E_INVALID	init value
replace	RTE_E_OK	REJECT	init value
dontInvalidate	RTE_E_OK	RTE_E_OK	init value

Configuration handleInvalid	RTE status		DE propagation
	init != invalid	init == invalid	
keep	RTE_E_INVALID	RTE_E_INVALID	last valid value[1]
replace	RTE_E_OK	REJECT	init value
dontInvalidate	RTE_E_OK	RTE_E_OK	value

Configuration handleOutOfRange	RTE status		DE propagation
	handleOutOfRange Status == silent[5]	handleOutOfRange Status == indicate[4,5]	
none	RTE_E_OK	RTE_E_OK	value
ignore	RTE_E_OK	RTE_E_ OUT_OF_RANGE	last valid value[2]
saturate	RTE_E_OK	RTE_E_ OUT_OF_RANGE	lower/upper limit
default[4]	RTE_E_OK	RTE_E_ OUT_OF_RANGE	init value[3]
invalid[4]	RTE_E_INVALID	RTE_E_INVALID	invalid value
external Replacement[5]	RTE_E_OK	RTE_E_ OUT_OF_RANGE	external replacement value

1. If no valid value was received previously then the init value shall be propagated
2. In case of queued communication the RTE behaves as if no value was enqueued
3. Init value shall not be equal to invalid value
4. Applicable only in combination with a non-queued COMSPEC
5. Applicable only in combination with a receiver COMSPEC

RTE status	DE propagation
RTE_E_OK	value

Figure 4.43: Overview for data invalidation and range checks

4.4 Modes

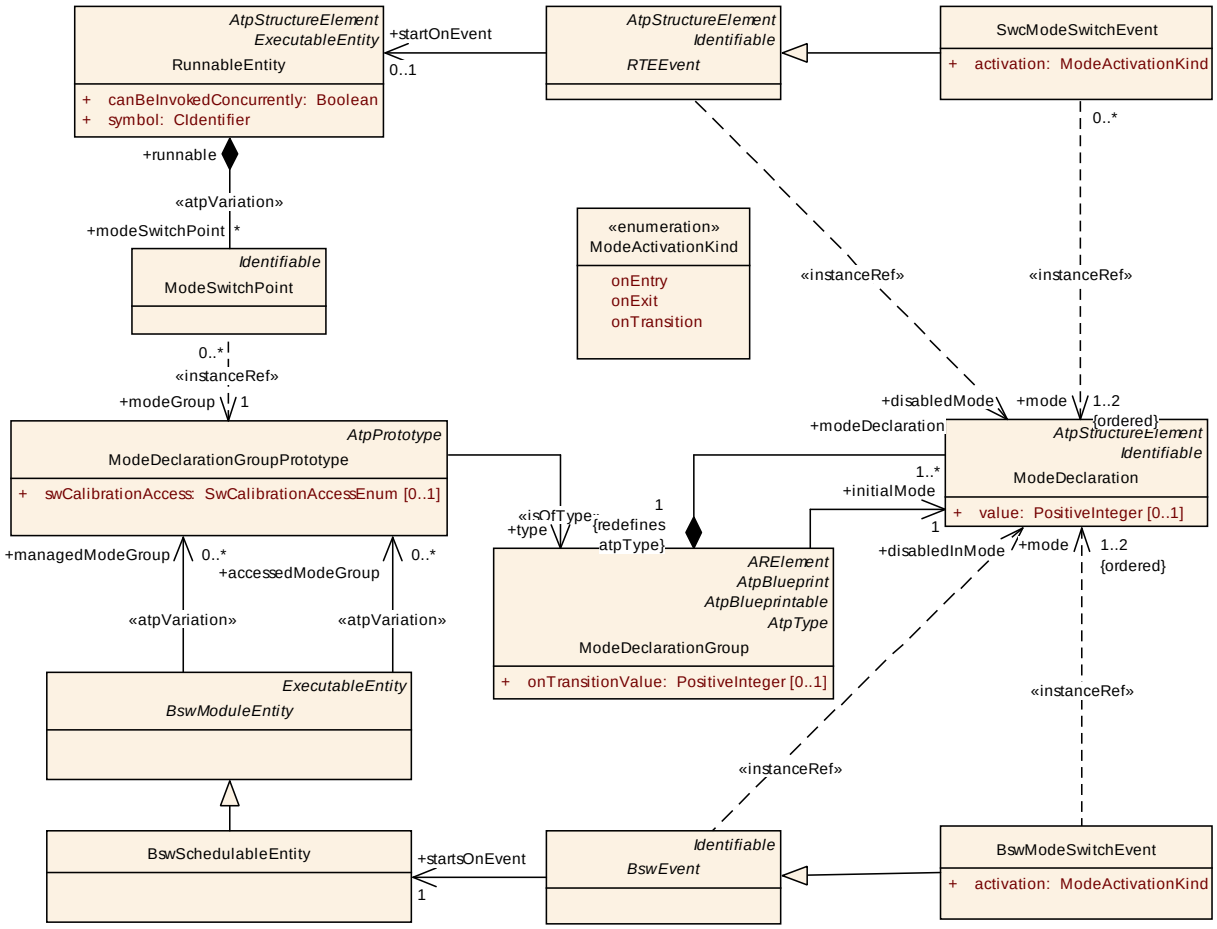


Figure 4.44: Summary of the use of ModeDeclarations by an AUTOSAR software-components and Basic Software Modules as defined in the *Software Component Template Specification* [2] and *Specification of BSW Module Description Template* [9].

The purpose of modes is to start *Runnable Entities* and *Basic Software Schedulable Entities* on the transition between modes and to disable (/enable) specified triggers of *Runnable Entities* and *Basic Software Schedulable Entities* in certain modes. Here, we use the specification of modes from the *Software Component Template Specification* [2]. Further on the document *Specification of BSW Module Description Template* [9] describes how modes are described for *Basic Software Modules*.

The first subsection 4.4.1 describes how modes can be used by an AUTOSAR software-component or *Basic Software Module* `mode user`(). The role of the `mode manager` who initiates mode switches is described in section 4.4.2. How ModeDeclarations are connected to a state machine is described in subsection 4.4.3. The behaviour of the RTE and *Basic Software Scheduler* regarding mode switches is detailed in subsection 4.4.4.

One usecase of modes is described in section 4.6.2 for the initialization and finalization of AUTOSAR software-components. Modes can be used for handling of communica-

tion states as well as for specific application purposes. The specific definition of modes and their use is not in the scope of this document.

The status of the modes will be notified to the AUTOSAR software-component `mode user` by mode communication - `mode switch notifications` - as described in the subsection 4.4.7. The port for receiving (or sending) a `mode switch notification` is called `mode switch port`.

A *Basic Software Module* `mode users` and the *Basic Software Module* `mode manager` are not necessarily using ports. *Basic Software Modules* without *AUTOSAR Interfaces* are connected via the configuration of the *Basic Software Scheduler*.

4.4.1 Mode User

To use modes, an AUTOSAR software-component (`mode user`) has to reference a ModeDeclarationGroup by a ModeDeclarationGroupPrototype of a require `mode switch port`, see section 4.4.7. The ModeDeclarationGroup contains the required modes.

An *Basic Software Module* (`mode user`) has to define a *requiredModeGroup ModeDeclarationGroupPrototype*.The *ModeDeclarationGroup* referred by these *ModeDeclarationGroupPrototype* contains the required modes.

The *ModeDeclarations* can be used in two ways by the `mode user` (see also figure 4.44):

1. Modes can be used to trigger runnables: The `SwcInternalBehavior` of the AUTOSAR SW-C or the `BswInternalBehavior` of the BSW module can define a `SwcModeSwitchEvent` respectively a `BswModeSwitchEvent` referencing the required `ModeDeclaration`. This *SwcModeSwitchEvent* or `BswModeSwitchEvent` can then be used as trigger for a *Runnable Entity* / *Basic Software Schedulable Entity*. Both `SwcModeSwitchEvent` and `BswModeSwitchEvent` carry an attribute `ModeActivationKind` which can be ‘exit’, ‘entry’, or ‘transition’.

   A *Runnable Entity* or *Basic Software Schedulable Entity* that is triggered by a `SwcModeSwitchEvent` or a `BswModeSwitchEvent` with `ModeActivationKind` ‘exit’ is triggered on exiting the mode. For simplicity it will be called `OnExit ExecutableEntity`. Correspondingly, an `OnTransition ExecutableEntity` is triggered by a `SwcModeSwitchEvent` or a `BswModeSwitchEvent` with `ModeActivationKind` ‘transition’ and will be executed during the transition between two modes, and an `OnEntry ExecutableEntity` is triggered by a `SwcModeSwitchEvent` or a `BswModeSwitchEvent` with `ModeActivationKind` ‘entry’ and will be executed when the mode is entered.

   Since a *Runnable Entity* as well as a *Basic Software Schedulable Entity* can be triggered by multiple `RTEEvent`s respectively `BswEvent`s, both can be an *OnExit-*, *OnTransition* and *OnEntry ExecutableEntity* at the same time.

RTE does not support a WaitPoint for a SwcModeSwitchEvent (see rte_sws_1358).

2. An *RTEEvent* and *BswEvent* that starts a *Runnable Entity* respectively a *Basic Software Schedulable Entity* can contain a *disabledInMode* association which *references* a ModeDeclaration. This association is called ModeDisablingDependency in this document.

   [rte_sws_2503]⌈ If a *Runnable Entity r* is referenced with *startOnEvent* by an *RTEEvent e* that has a ModeDisablingDependency on a mode *m*, then RTE shall not activate runnable *r* on any occurrence of *e* while the mode *m* is active. ⌋*(RTE00143, RTE00052)*

   [rte_sws_7530]⌈ If a *Basic Software Schedulable Entity r* is referenced with *startOnEvent* by an *BswEvent e* that has a ModeDisablingDependency on a mode *m*, then *Basic Software Scheduler* shall not activate Basic Software Schedulable Entitys *r* on any occurrence of *e* while the mode *m* is active. ⌋*(RTE00213)*

   Note: As a consequence of rte_sws_2503 and rte_sws_7530 in combination with rte_sws_2661, RTE or *Basic Software Scheduler* will not start runnable or BswSchedulableEntity *r* on any occurrence of *e* while the mode *m* is active.

   The mode disabling is active during the transition to a mode, during the mode itself and during the transition for exiting the mode. For a precise definition see section 4.4.4.

   The existence of a ModeDisablingDependency prevents the RTE to start the mode disabling dependent ExecutableEntity by the disabled *RTEEvent* / *BswEvent* during the mode, referenced by the ModeDisablingDependency, and during the transitions from and to that mode. ModeDisablingDependencys override any activation of a *Runnable Entity* and *Basic Software Schedulable Entity* by the disabled RTEEvents / BswEvents. This is also true for the SwcModeSwitchEvent and BswModeSwitchEvent.

   A *Runnable Entity* as well as a *Basic Software Schedulable Entity* can not be 'enabled' explicitly. *Runnable Entities* are *Basic Software Schedulable Entities* are only 'enabled' by the absence of any active ModeDisablingDependencys.

   Note that ModeDisablingDependencys do not prevent the wake up from a WaitPoint by the 'disabled' RTEEvent. This allows the wake-uped Runnable Entity to run until completion if a transition occurred during the Runnable Entity's execution.

   [rte_sws_2504]⌈ The existence of a ModeDisablingDependency shall not instruct the RTE to kill a running runnable at a mode switch. ⌋*(RTE00143)*

   [rte_sws_7531]⌈ The existence of a ModeDisablingDependency shall not instruct the *Basic Software Scheduler* to kill a running *Basic Software Schedulable Entity* at a mode switch. ⌋*(RTE00213)*

The RTE and the *Basic Software Scheduler* can be configured to switch schedule tables to implement mode disabling dependencies for cyclic triggers of *Runnable Entities* or *Basic Software Schedulable Entities*. Sets of mutual exclusive modes can be mapped to different schedule tables. The RTE shall implement the switch between schedule tables according to the mapping of modes to schedule tables in `RteModeScheduleTableRef`, see rte_sws_5146.

The mode user can specify in the *ModeSwitchReceiverComSpec* (software components) or *BswModeReceiverPolicy* (BSW modules) that it is able to deal with asynchronous mode switch behavior (`supportsAsynchronousModeSwitch == TRUE`). If all `mode users` connected to the same *ModeDeclarationGroupPrototype* of the `mode manager` support the asynchronous mode switch behavior, the related `mode machine instance` can be implemented with the asynchronous mode switching procedure. Otherwise, the synchronous mode switching procedure has to be applied (see rte_sws_7150).

4.4.2 Mode Manager

Entering and leaving modes is initiated by a `mode manager`. A `mode manager` might be a basic software module, for example the Basic Software Mode Manager (BswM), the communication manager (ComM), or the ECU state manager (EcuM). The `mode manager` may also be an AUTOSAR SW-C. In this case, it is called an `application mode manager`.

The `mode manager` contains the master state machine to represent the modes.

To provide modes, an AUTOSAR software-component (`mode manager`) has to reference a *ModeDeclarationGroup* by a *ModeDeclarationGroupPrototype* of a provide `mode switch port`, see section 4.4.7. The *ModeDeclarationGroup* contains the provided modes.

An *Basic Software Module* (`mode manager`) has to define a *providedModeGroup ModeDeclarationGroupPrototype*. The *ModeDeclarationGroup* referred by these *ModeDeclarationGroupPrototype* contains the provided modes.

The RTE / *Basic Software Scheduler* will take the actions necessary to switch between the modes. This includes the termination and execution of several *ExecutableEntities* from all `mode users` that are connected to the same *ModeDeclarationGroupPrototype* of the `mode manager`. To do so, the RTE / *Basic Software Scheduler* needs a state machine to keep track of the currently active modes and transitions initiated by the `mode manager`. The RTE's / *Basic Software Scheduler*'s mode machine is called `mode machine instance`. There is exactly one `mode machine instance` for each *ModeDeclarationGroupPrototype* of a `mode manager`'s provide `mode switch port` respectively *providedModeGroup ModeDeclarationGroupPrototype*.

It is the responsibility of the `mode manager` to advance the RTE's / *Basic Software Scheduler*'s `mode machine instance` by sending `mode switch notifications` to the `mode users`. The `mode switch notifications` are imple-

mented by a non blocking API (see 5.6.6 / 6.5.3). So, the mode switch notifications alone provide only a loose coupling between the state machine of the mode manager and the mode machine instance of the RTE / *Basic Software Scheduler*. To prevent, that the mode machine instance lags behind and the states of the mode manager and the RTE / *Basic Software Scheduler* get out of phase, the mode manager can use acknowledgment feedback for the mode switch notification. RTE / *Basic Software Scheduler* can be configured to send an acknowledgment of the mode switch notification to the mode manager when the requested transition is completed.

At the mode manager, the acknowledgment results in an *ModeSwitchedAckEvent*. As with *DataSendCompletedEvent*s, this event can be picked up with the polling or blocking Rte_SwitchAck API. And the event can be used to trigger a mode switch acknowledge ExecutableEntity to pick up the status. Note: The *Basic Software Scheduler* do not support WaitPoints. Therefore the SchM_SwitchAck never blocks.

Some possible usage patterns for the acknowledgement are:

- The most straight forward method is to use a sequence of Rte_Switch and a blocking Rte_SwitchAck to send the mode switch notification and wait for the completion. This requires the use of an extended task.
- Another possibility is to have a cyclic *Runnable Entity* / *Basic Software Schedulable Entity* (maybe the same that switches the modes via Rte_Switch / SchM_Switch) to poll for the acknowledgement using Rte_SwitchAck / SchM_SwitchAck.
- The acknowledgement can also be polled from a *Runnable Entity* or *Basic Software Schedulable Entity* that is started by the *ModeSwitchedAckEvent*.

The mode manager can also use the Rte_Mode / SchM_Mode API to read the currently active mode from the RTE's / *Basic Software Scheduler*'s perspective.

4.4.3 Refinement of the semantics of *ModeDeclarations* and *ModeDeclarationGroups*

To implement the logic of mode switches, the RTE / *Basic Software Scheduler* needs some basic information about the available modes. For this reason, RTE / *Basic Software Scheduler* will make the following additional assumptions about the modes of one *ModeDeclarationGroup*:

1. **[rte_sws_ext_2542]** Whenever any *Runnable Entity* or *Basic Software Schedulable Entity* is running, there shall always be exactly one mode or one mode transition active of each ModeDeclarationGroupPrototype.
2. Immediately after initialization of a mode machine instance, RTE / *Basic Software Scheduler* will execute a transition to the initial mode of each ModeDeclarationGroupPrototype (see rte_sws_2544).

RTE / *Basic Software Scheduler* will enforce the `mode disablings` of the initial modes and trigger the `OnEntry ExecutableEntitys` (if any defined) of the initial modes of every `ModeDeclarationGroupPrototype` immediately after initialization of the RTE / *Basic Software Scheduler*.

In other words, RTE / *Basic Software Scheduler* assumes, that the modes of one `ModeDeclarationGroupPrototype` belong to exactly one state machine without nested states. The state machines cover the whole lifetime of the atomic AUTOSAR SW-Cs[9] and mode dependent AUTOSAR Basic Software Modules [10].

4.4.4 Order of actions taken by the RTE / *Basic Software Scheduler* upon interception of a mode switch notification

This section describes what the 'communication' of a mode switch to a `mode user` actually does. What does the RTE *Basic Software Scheduler* do to switch a mode and especially in which order.

Mode switch procedures

Depending on the needs of mode users for synchronicity, the mode machine instance can be implemented with two different realizations.

- synchronous mode switching procedure
- asynchronous mode switching procedure

The differences between these two realizations are the omitted waiting conditions in case of asynchronous mode switching procedure. For instance with asynchronous behavior a software component can not rely that all `mode disabling dependent ExecutableEntitys` of the previous mode are terminated before `OnEntry ExecutableEntitys` and `OnExit ExecutableEntitys` are started. On one hand this might put some effort to the software component designer to enable the components implementation to support this kind of scheduling but on the other hand it enables fast and lean mode switching.

[rte_sws_7150]⌈ The RTE generator shall use the synchronous mode switching procedure if at least one `mode user` of the `mode machine instance` does not support the asynchronous mode switch behavior. ⌋*(RTE00143, RTE00213)*

[rte_sws_7151]⌈ The RTE generator shall apply the asynchronous mode switch behavior, if all `mode users` support the asynchronous mode switch behavior and if it is configured for the related `mode machine instance`. ⌋*(RTE00143, RTE00213)*

Typical usage of modes to protect resources

[9] The lifetime of an atomic AUTOSAR SW-C is considered to be the time span in which the SW-C's runnables are being executed.

[10] The lifetime of an mode dependent AUTOSAR Basic Software Module is considered to be the time span in which the *Basic Software Schedulable Entities* are being executed.

RTE / *Basic Software Scheduler* can start and prevent the execution of *Runnable Entities* and `BswSchedulableEntity`. In the context of mode switches,

- RTE / *Basic Software Scheduler* starts `OnExit ExecutableEntitys` for leaving the previous mode. This is typically used by 'clean up ExecutableEntities' to free resources that were used during the previous mode.
- RTE / *Basic Software Scheduler* starts `OnEntry ExecutableEntitys` for entering the next mode. This is typically used by 'initialization ExecutableEntities' to allocate resources that are used in the next mode.
- And RTE / *Basic Software Scheduler* can prevent the execution of `mode disabling dependent ExecutableEntitys` within a mode. This is typically used with time triggered 'work ExecutableEntity' that use a resource which is not available in a certain mode.

According to this use case, during the execution of 'clean up ExecutableEntity' and 'initialization ExecutableEntity' the 'work ExecutableEntity' should be disabled to protect the resource. Also, if the same resource is used (by different SW-C's) in two successive modes, the 'clean up ExecutableEntity' should be safely terminated before the 'initialization ExecutableEntity' of the next mode are executed (synchronous mode switching procedure). In summary, this would lead to the following sequence of actions by the RTE / *Basic Software Scheduler* upon reception of the `mode switch notification`:

1. activate `mode disablings` for the next mode
2. wait for the newly disabled ExecutableEntities to terminate in case of synchronous mode switching procedure
3. execute 'clean up ExecutableEntities'
4. wait for the 'clean up ExecutableEntities' to terminate in case of synchronous mode switching procedure
5. execute 'initialization ExecutableEntities'
6. wait for the 'initialization ExecutableEntities' to terminate in case of synchronous mode switching procedure
7. deactivate `mode disablings` for the previous modes and enable ExecutableEntities that have been disabled in the previous mode.

RTE / *Basic Software Scheduler* can also start `OnTransition ExecutableEntitys` on a transition between two modes which is not shown in this use case example.

Often, only a fraction of the SW-Cs, Runnable Entities, Basic Software modules and Basic Software Schedulable Entities of one ECU depends on the modes that are switched. Consequently, it should be possible to design the system in a way, that the mode switch does not influence the performance of the remaining software.

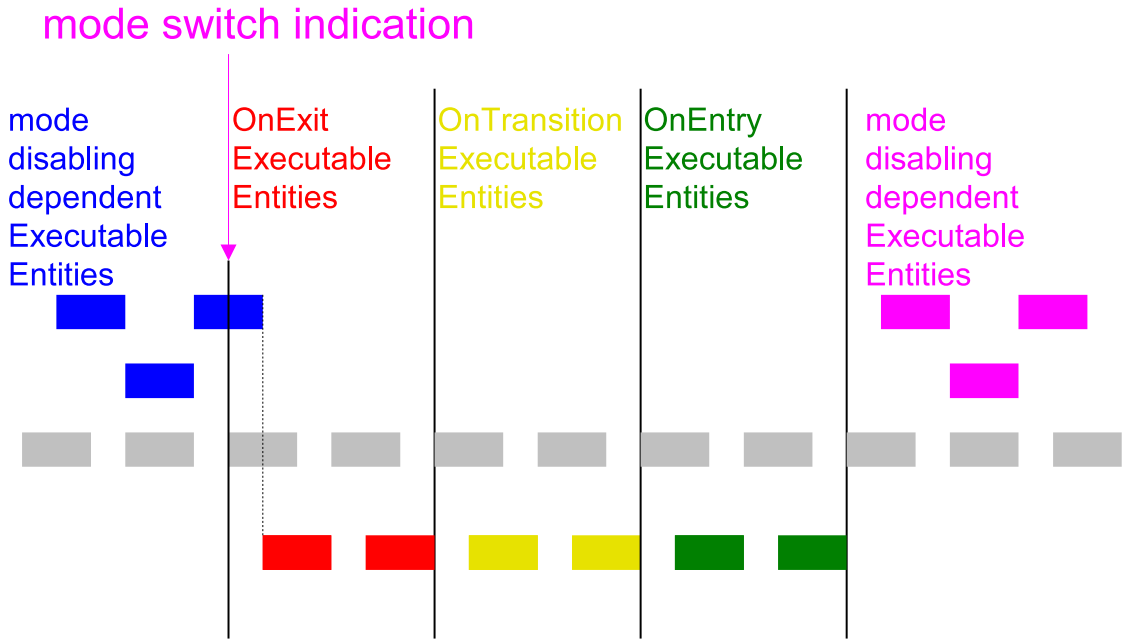


Figure 4.45: This figure shall illustrate what kind of ExecutableEntities will run in what order during a synchronous mode transition. The boxes indicate activated ExecutableEntities. Mode disabling dependant ExecutableEntities are printed in blue (old mode) and pink (new mode). OnExit, OnTransition, and OnEntry ExecutableEntity are printed in red, yellow, and green.

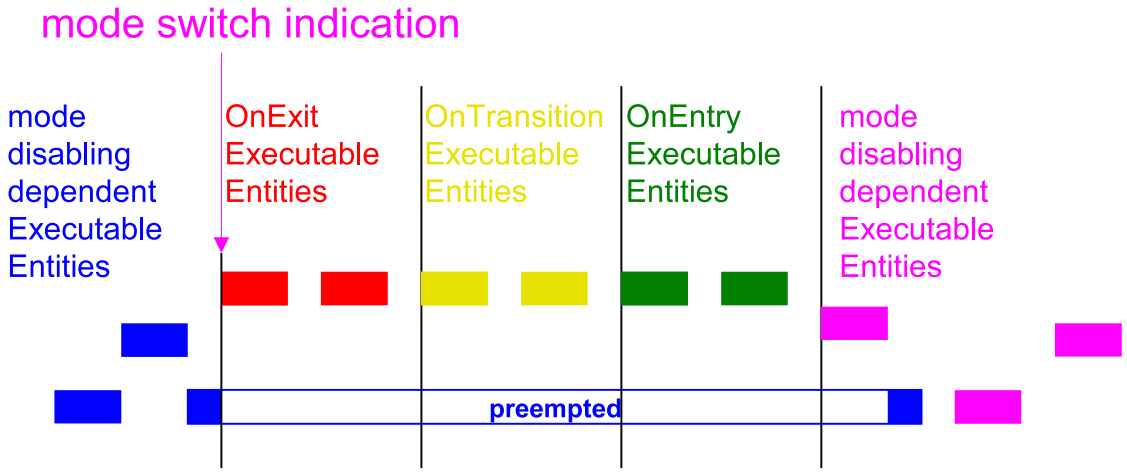


Figure 4.46: This figure shall illustrate what kind of ExecutableEntity will run in what order during an asynchronous mode transition where the ExecutableEntities are triggered on a mode change are mapped to a higher priority task than the Mode Dependent ExecutableEntity. The boxes indicate activated ExecutableEntity. Mode disabling dependant ExecutableEntity are printed in blue (old mode) and pink (new mode). OnExit, OnTransition, and OnEntry ExecutableEntity are printed in red, yellow, and green.

The remainder of this section lists the requirements that guarantee the behavior described above.

All runnables with dependencies on modes have to be executed or terminated during mode transitions. Restriction rte_sws_2500 requires these runnables to be of category 1 to guarantee finite execution time.

For simplicity of the implementation to guarantee the order of runnable executions, the following restriction is made:

All `OnEntry ExecutableEntitys`, `OnTransition ExecutableEntitys`, and `OnExit ExecutableEntitys` of the same `mode machine instance` should be mapped to the same task in the execution order following OnExit, OnTransition, OnEntry (see rte_sws_2662).

A `mode machine instance` implementing an asynchronous mode switch procedure might be fully implemented inside the `Rte_Switch` or `SchM_Switch` API. In this case the `OnEntry ExecutableEntitys`, `OnTransition ExecutableEntitys`, `OnExit ExecutableEntitys` and `mode switch acknowledge ExecutableEntitys` are not mapped to tasks as described in chapter 7.6.1.

[rte_sws_7173]⌈ The RTE generator shall support invocation of `OnEntry ExecutableEntitys`, `OnTransition ExecutableEntitys`, `OnExit ExecutableEntitys` and `mode switch acknowledge ExecutableEntitys` via direct function call, if all following conditions are fulfilled:

- if the asynchronous mode switch behavior is configured (see rte_sws_7151)
- the `OnEntry ExecutableEntitys`, `OnTransition ExecutableEntitys`, `OnExit ExecutableEntitys` and `mode switch acknowledge ExecutableEntitys` do not define a 'minimum start distance'
- the `mode manager` and `mode user` are in the same Partition
- if the preconditions of table 4.5 are fulfilled

⌋*(RTE00143, RTE00213)*

Further on the requirements rte_sws_5083, rte_sws_7155 and rte_sws_7157 has to be considered.

[rte_sws_2667]⌈ Within the mode manager's `Rte_Switch` / `SchM_Switch` API call to indicate a mode switch, one of the following shall be done:

1. If the corresponding `mode machine instance` is in a transition, and the queue for `mode switch notifications` is full, `Rte_Switch` / `SchM_Switch` shall return an error immediately.
2. If the corresponding `mode machine instance` is in a transition, and the queue for `mode switch notifications` is not full, the `mode switch notification` shall be queued.
3. If the `mode machine instance` is not in a transition, `Rte_Switch` / `SchM_Switch` shall activate the `mode disablings` (see rte_sws_2661) of

the next mode, and initiate the transition as described by the sequence in rte_sws_2665.

⌋*(RTE00143, RTE00213)*

The following list holds the requirements for the steps of a mode transition.

- **[rte_sws_2661]**⌈ At the beginning of a transition of a `mode machine instance`, the RTE / *Basic Software Scheduler* shall activate the `mode disablings` of the next mode (see also rte_sws_2503), if any `ModeDisablingDependencys` for that mode are defined. ⌋*(RTE00143, RTE00213)*
- **[rte_sws_7152]**⌈ If any `ModeDisablingDependencys` for the next mode are defined (as specified by rte_sws_2661), the RTE / *Basic Software Scheduler* shall wait until the newly disabled *Runnable Entities* and *Basic Software Schedulable Entities* are terminated, in case of synchronous mode switching procedure. ⌋*(RTE00143, RTE00213)*

  Note: To guarantee in case of synchronous mode switching all activated `mode disabling dependent ExecutableEntitys` of this `mode machine instance` have terminated before the start of the `OnExit ExecutableEntitys` of the transition, RTE generator can exploit the restriction rte_sws_2663 that `mode disabling dependent ExecutableEntitys` run with higher or equal priority than the `OnExit ExecutableEntitys` and the `OnEntry ExecutableEntitys`.
- **[rte_sws_2562]**⌈ RTE / *Basic Software Scheduler* shall execute the `OnExit ExecutableEntitys` of the previous mode. ⌋*(RTE00143, RTE00052, RTE00213)*
- **[rte_sws_7153]**⌈ If any `OnExit ExecutableEntity` is configured the RTE / *Basic Software Scheduler* shall wait after its execution (rte_sws_2562) until all `OnExit ExecutableEntitys` are terminated in case of synchronous mode switching procedure. ⌋*(RTE00143, RTE00213)*
- **[rte_sws_2707]**⌈ RTE / *Basic Software Scheduler* shall execute the `OnTransition ExecutableEntitys` of the next mode. ⌋*(RTE00143, RTE00052, RTE00213)*
- **[rte_sws_2708]**⌈ If any `OnTransition ExecutableEntity` is configured, the RTE / *Basic Software Scheduler* shall wait after its execution (rte_sws_2707) until all `OnTransition ExecutableEntitys` are terminated in case of synchronous mode switching procedure. ⌋*(RTE00143, RTE00213)*
- **[rte_sws_2564]**⌈ RTE / *Basic Software Scheduler* shall execute the `OnEntry ExecutableEntitys` of the next mode. ⌋*(RTE00143, RTE00052, RTE00213)*
- **[rte_sws_7154]**⌈ If any `OnEntry ExecutableEntity` is configured the RTE shall wait after its execution (rte_sws_2564) until all `OnEntry ExecutableEntitys` are terminated in case of synchronous mode switching procedure. ⌋*(RTE00143, RTE00213)*

- **[rte_sws_2563]**⌈ The RTE / *Basic Software Scheduler* shall deactivate the previous `mode disablings` and only keep the `mode disablings` of the next mode. ⌋*(RTE00143, RTE00213)*

  With this, the transition is completed.

- **[rte_sws_2587]**⌈ At the end of the transition, RTE / *Basic Software Scheduler* shall trigger the *ModeSwitchedAckEvent*s connected to the `mode manager`'s *ModeDeclarationGroupPrototype*. ⌋*(RTE00143, RTE00213)*

  This will result in an acknowledgment on the `mode manager`'s side which allows the `mode manager` to wait for the completion of the mode switch.

  The dequeuing of the mode switch notification shall also be done at the end of the transition, see rte_sws_2721.

[rte_sws_2665]⌈ During a transition of a `mode machine instance` each applicable of the steps

1. rte_sws_2661 (The transition is entered in parallel with this step),
2. rte_sws_7152,
3. rte_sws_2562,
4. rte_sws_7153,
5. rte_sws_2707,
6. rte_sws_2708,
7. rte_sws_2564,
8. rte_sws_7154,
9. rte_sws_2563 (The transition is completed with this step), and
10. immediately followed by rte_sws_2587

shall be executed in the order as listed. If a step is not applicable, the order of the remaining steps shall be unchanged. ⌋*(RTE00143, RTE00213)*

[rte_sws_2668]⌈ Immediately after the execution of a transition as described in rte_sws_2665, RTE / *Basic Software Scheduler* shall check the queue for pending `mode switch notifications` of this `mode machine instance`. If a `mode switch notification` can be dequeued, the `mode machine instance` shall enter the corresponding transition directly as described by the sequence in rte_sws_2665. ⌋*(RTE00143, RTE00213)*

In the case of a fast sequence of two mode switches, the `Rte_Mode` or `SchM_Mode` API will not indicate an intermediate mode, if a `mode switch notification` to the next mode is indicated before the transition to the intermediate mode is completed.

[rte_sws_2630]⌈ In case of synchronous mode switch procedure, the RTE shall execute all steps of a mode switch (see rte_sws_2665) synchronously for the whole `mode machine instance`. ⌋*(RTE00143, RTE00213)*

I.e., the mode transitions will be executed synchronously for all `mode users` that are connected to the same `mode manager`'s ModeDeclarationGroupPrototype.

[rte_sws_2669]⌈ If the next mode and the previous mode of a transition are the same, the transition shall still be executed. ⌋*(RTE00143, RTE00213)*

4.4.5 Assignment of mode machine instances to RTE and Basic Software Scheduler

[rte_sws_7533]⌈ A `mode machine instance` shall be assigned to the RTE if the correlating *ModeDeclarationGroupPrototype* is instantiated in a port of a software-component and if the *ModeDeclarationGroupPrototype* is not synchronized (*synchronizedModeGroup* of a *SwcBswMapping*) with a *providedModeGroup ModeDeclarationGroupPrototype* of a Basic Software Module instance. ⌋*(RTE00143)*

[rte_sws_7534]⌈ A `mode machine instance` shall be assigned to the *Basic Software Scheduler* if the correlating *ModeDeclarationGroupPrototype* is a *providedModeGroup ModeDeclarationGroupPrototype* of a Basic Software Module instance. ⌋*(RTE00213)*

[rte_sws_7535]⌈ The RTE Generator shall create only one `mode machine instance` if a *ModeDeclarationGroupPrototype* instantiated in a port of a software-component is synchronized (*synchronizedModeGroup* of a *SwcBswMapping*) with a *providedModeGroup ModeDeclarationGroupPrototype* of a Basic Software Module instance. The related `common mode machine instance` shall be assigned to the *Basic Software Scheduler*. ⌋*(RTE00143, RTE00213, RTE00214)*

In case of synchronized *ModeDeclarationGroupPrototype*s the correlating `common mode machine instance` is initialized during the execution of the `SchM_Init`. At this point of time the scheduling of *Runnable Entities* is not enabled due to the uninitialized RTE. Therefore situation occurs, that the *Runnable Entities* being `OnEntry ExecutableEntitys` are not called if the `mode machine instance` is initialized. Further on the current mode of such `mode machine instance` might be still switched

until the RTE gets initialized. Nevertheless the *OnEntry Runnables* of the current active mode are executed.

[rte_sws_7582]⌈ For `common mode machine instances` the *OnEntry Runnable Entities* of the current active mode are executed during the initialization of the RTE if the `common mode machine instance` is not in transition. ⌋*(RTE00214)*

[rte_sws_7583]⌈ A `common mode machine instances` is not allowed to enter transition phase during the RTE initialization if the `common mode machine instances` has *OnEntry Runnable Entities*, *OnTransition Runnable Entities* or *OnExit Runnable Entities* ⌋*(RTE00214)*

Note: rte_sws_7582 and rte_sws_7583 shall ensure a deterministic behavior that the software components receiving a Mode Switch Request from a `common mode machine instances` are receiving the current active mode during RTE initialization.

[rte_sws_7564]⌈ The RTE generator shall reject configurations where *ModeSwitchPoint*(s) referencing a *ModeDeclarationGroupPrototype* in a `mode switch port` and a *managedModeGroup* association(s) to a *providedModeGroup ModeDeclarationGroupPrototype* are not defined mutual exclusively to one of two synchronized *ModeDeclarationGroupPrototype*s. ⌋*(RTE00143, RTE00213, RTE00214, RTE00018)*

[rte_sws_ext_7565] Only one of two synchronized *ModeDeclarationGroupPrototype*s shall mutual exclusively be referenced by *ModeSwitchPoint*(s) or *managedModeGroup* association(s).

Note: rte_sws_ext_7565 shall ensure in the combination with the existence conditions of the `Rte_Switch`, `Rte_Mode`, `Rte_SwitchAck`, `SchM_Switch`, `SchM_Mode` and `SchM_SwitchAck` that either the port based RTE API or the *Basic Software Scheduler* API (rte_sws_7201 and rte_sws_7264) offered to the implementation of the `mode manager`.

4.4.6 Initialization of mode machine instances

[rte_sws_2544]⌈ RTE shall initiate the transition to the initial modes of each `mode machine instance` belonging to the RTE during `Rte_Start`. During the transition to the initial modes, the steps defined in the following requirements have to be omitted as no previous mode is defined:

- rte_sws_2562,
- rte_sws_7153,
- rte_sws_2707,
- rte_sws_2708,
- rte_sws_2563,
- rte_sws_2587

If applicable, the steps described by the following requirements still have to be executed for entering the initial mode:

- rte_sws_2661,
- rte_sws_2564

⌋*(RTE00143, RTE00144, RTE00116)*

[rte_sws_7532]⌈ *Basic Software Scheduler* shall initiate the transition to the initial modes of each `mode machine instance` belonging to the *Basic Software Scheduler* during `SchM_Init`. During the transition to the initial modes, the steps defined in the following requirements have to be omitted as no previous mode is defined:

- rte_sws_2562,
- rte_sws_7153,
- rte_sws_2707,
- rte_sws_2708,
- rte_sws_2563,
- rte_sws_2587

If applicable, the steps described by the following requirements still have to be executed for entering the initial mode:

- rte_sws_2661,
- rte_sws_2564

⌋*(RTE00213)*

4.4.7 Notification of mode switches

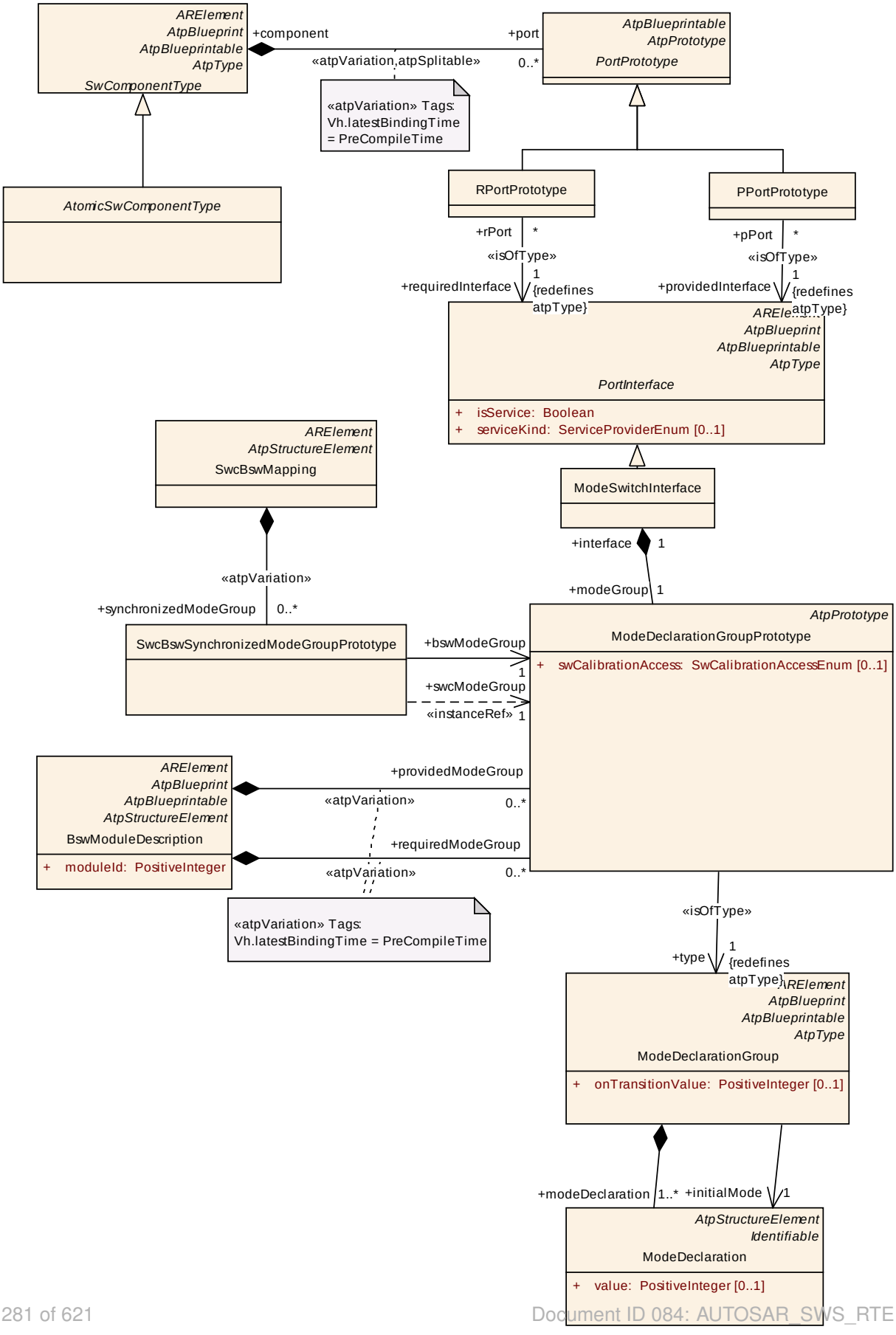


Figure 4.47: Definition of a ModeSwitchInterface.

- **[rte_sws_2549]**⌈ Mode switches shall be communicated via RTE by `ModeDeclarationGroupPrototype`s of a `ModeSwitchInterface` as defined in [2], see Fig. 4.47. ⌋*(RTE00144)*

  The `mode switch ports` of the `mode manager` and the `mode user` are of the type of a `ModeSwitchInterface`.

- **[rte_sws_7538]**⌈ Mode switches shall be communicated via *Basic Software Scheduler* via *providedModeGroup* and *requiredModeGroup ModeDeclarationGroupPrototype*s as defined in [9], see Fig. 4.47. Which *ModeDeclarationGroupPrototype*s are connected to each other is defined by the configuration of the *Basic Software Scheduler*. ⌋*(RTE00213)*

- RTE / *Basic Software Scheduler* only requires the notification of switches between modes.

- AUTOSAR does not support inter ECU communication of mode switch notifications.

  RTE does not support a configuration in which the `mode users` of one `mode machine instance` are distributed over several partitions, see rte_sws_2724.

  Rationale: Mode switch communication requires high synchronization effort. Such a high coupling should be avoided between ECUs and between partitions. This does not prevent distributed mode management.

  For the distributed mode management mode requests can be distributed via `ServiceProxySwComponent`s and the BswM of each target ECU to the `mode users` of the BswMs.

- **[rte_sws_2508]**⌈ A mode switch shall be notified asynchronously as indicated by the use of a *ModeSwitchInterface*. ⌋*(RTE00144)*

  Rationale: This simplifies the communication. Due to rte_sws_ext_2724 the communication is local and no handshake is required to guarantee reliable transmission.

  RTE offers the `Rte_Switch` API to the `mode manager` for this notification, see 5.6.6.

  Basic Software Scheduler offers the `SchM_Switch` API to the `mode manager` for this notification, see 6.5.3.

- The `mode manager` might still require a feedback to keep it's internal state machine synchronized with the RTE / *Basic Software Scheduler* view of active modes.

The RTE generator shall support an AcknowledgementRequest from the mode switch port / *providedModeGroup ModeDeclarationGroupPrototype* of a mode manager, see rte_sws_2587, to notify the mode manager of the completion of a mode switch.

- **[rte_sws_2566]**⌈ A ModeSwitchInterface shall support 1:n communication. ⌋*(RTE00144)*

  Rationale: This simplifies the configuration and the communication. One mode switch can be notified to all receivers simultaneously.

  A ModeSwitchInterface does not support n:1 communication, see rte_sws_2670.

- **[rte_sws_7539]**⌈ The connection of *providedModeGroup* and *requiredModeGroup ModeDeclarationGroupPrototype* shall support 1:n communication. ⌋*(RTE00213)*

- **[rte_sws_2624]**⌈ A mode switch shall be notified with event semantics, i.e., the mode switch notifications shall be buffered by RTE or *Basic Software Scheduler* to which the mode machine instance is assigned. ⌋*(RTE00144)*

  The queueing of mode switches (and SwcModeSwitchEvents) depends like that of DataReceivedEvents on the settings for the receiving port, see section 4.3.1.10.2.

- **[rte_sws_2567]**⌈ A ModeSwitchInterface shall only indicate the next mode of the transition. ⌋*(RTE00144)*

- **[rte_sws_7541]**⌈ A *providedModeGroup ModeDeclarationGroupPrototype* shall only indicate the next mode of the transition. ⌋*(RTE00213)*

  The API takes a single parameter (plus, optionally, the instance handle) that indicates the requested 'next mode'. For this purpose, RTE and *Basic Software Scheduler* will use identifiers of the modes as defined in rte_sws_2568 and rte_sws_7294.

- **[rte_sws_2546]**⌈ The RTE shall keep track of the active modes of a mode manager's ModeDeclarationGroupPrototypes (mode machine instances) which is assigned to the RTE. ⌋*(RTE00143, RTE00144)*

- **[rte_sws_7540]**⌈ The *Basic Software Scheduler* shall keep track of the active modes of a mode manager's ModeDeclarationGroupPrototypes (mode machine instances) which is assigned to the *Basic Software Scheduler*. ⌋*(RTE00213, RTE00144)*

  Rationale: This allows the RTE / *Basic Software Scheduler* to guarantee consistency between the timing for firing of SwcModeSwitchEvents / BswModeSwitchEvents and disabling the start of ExecutableEntities by ModeDisablingDependency without adding additional interfaces to a mode manager with fine grained substates on the transitions.

- The RTE offers an `Rte_Mode` API to the SW-C to get information about the active mode, see section 5.6.29.
- The *Basic Software Scheduler* offers an `SchM_Mode` API to the Basic Software Module to get information about the active mode, see section 6.5.4.
- In addition to the `mode switch ports`, the `mode manager` may offer an AUTOSAR interface for requesting and releasing modes as a means to keep modes alive like for ComM and EcuM.

4.4.8 Mode switch acknowledgment

In case of mode switch communication, the `mode manager` may specify a `ModeSwitchedAckEvent` or `BswModeSwitchedAckEvent` to receive a notification from the RTE that the mode transition has been completed, see rte_sws_2679 and rte_sws_7559.

[rte_sws_2679]⌈ If acknowledgement is enabled for a provided `ModeDeclarationGroupPrototype` and a `ModeSwitchedAckEvent` references a `RunnableEntity` as well as the `ModeDeclarationGroupPrototype`, the `RunnableEntity` shall be activated when the mode switch acknowledgment occurs or when the RTE detects that the partition to which the mode users are mapped was stopped or restarted. ⌋*(RTE00051, RTE00143)*

Note the constraint that all mode users are in the same partition (rte_sws_2724).

The related *Entry Point Prototype* is defined in rte_sws_2512.

[rte_sws_7559]⌈ If acknowledgement is enabled for a provided (`providedModeGroup`) `ModeDeclarationGroupPrototype` and a `BswModeSwitchedAckEvent` references a `BswSchedulableEntity` as well as the `ModeDeclarationGroupPrototype`, the `BswSchedulableEntity` shall be activated when the mode switch acknowledgment occurs or when a timeout was detected by the *Basic Software Scheduler*. rte_sws_2587. ⌋*(RTE00213, RTE00143)*

The related *Entry Point Prototype* is defined in rte_sws_7283.

Requirement rte_sws_2679 and rte_sws_7559 merely affects when the runnable is activated. The `Rte_SwitchAck` and `SchM_SwitchAck` shall still be created, according to requirement rte_sws_2678 and rte_sws_7558 to actually read the acknowledgment.

[rte_sws_2730]⌈ A `ModeSwitchedAckEvent` that references a `RunnableEntity` and is referenced by a `WaitPoint` shall be an invalid configuration which is rejected by the RTE generator. ⌋*(RTE00051, RTE00018, RTE00143)*

The attributes `ModeSwitchedAckRequest` and `BswModeSwitchedAckRequest` allow to specify a timeout.

[rte_sws_7056]⌈ If `ModeSwitchedAckRequest` or `BswModeSwitchedAckRequest` with a timeout greater than zero is specified, the RTE shall ensure that time-

out monitoring is performed, regardless of the receive mode of the acknowledgment. ⌋*(RTE00069, RTE00143)*

[rte_sws_7060]⌈ Regardless of an occurred timeout during a mode transition the RTE shall complete the transition of a `mode machine instance` as defined in rte_sws_2665. ⌋*(RTE00069, RTE00143)*

If a `WaitPoint` is specified to collect the acknowledgment, two timeout values have to be specified, one for the `ModeSwitchedAckRequest` and one for the `WaitPoint`.

[rte_sws_7057]⌈ If different timeout values are specified for the `ModeSwitchedAck-Request` for a `ModeDeclarationGroupPrototype` and for the `WaitPoint` associated with the `ModeSwitchedAckEvent` for the `ModeSwitchPoint`referring to that `ModeDeclarationGroupPrototype`, the configuration shall be rejected by the RTE generator. ⌋*(RTE00018, RTE00143)*

[rte_sws_7058]⌈ The status information about the success or failure of the mode transition shall be buffered with last-is-best semantics. When a new `mode switch notification` is sent or when the mode switch notification was completed after a timeout, the status information is overwritten. ⌋*(RTE00143)*

rte_sws_7058 implies that once the `ModeSwitchedAckEvent` or `BswModeSwitchedAckEvent` has occurred, repeated API calls (`Rte_SwitchAck` or `SchM_SwitchAck` to retrieve the acknowledgment can return different values.

[rte_sws_7059]⌈ If the `timeout` value of the `ModeSwitchedAckRequest` or `BswModeSwitchedAckRequest` is 0, no timeout monitoring shall be performed. ⌋*(RTE00069, RTE00143)*

4.5 External and Internal Trigger

4.5.1 External Trigger Event Communication

4.5.1.1 Introduction

With the mechanism of the trigger event communication a software component or a *Basic Software Module* acting as a `Trigger Source` is able to request the activation of *Runnable Entities* respectively *Basic Software Schedulable Entities* of connected `Trigger Sinks`. Typically but not necessarily these *Runnable Entities* and *Basic Software Schedulable Entities* are executed in a sequential order.

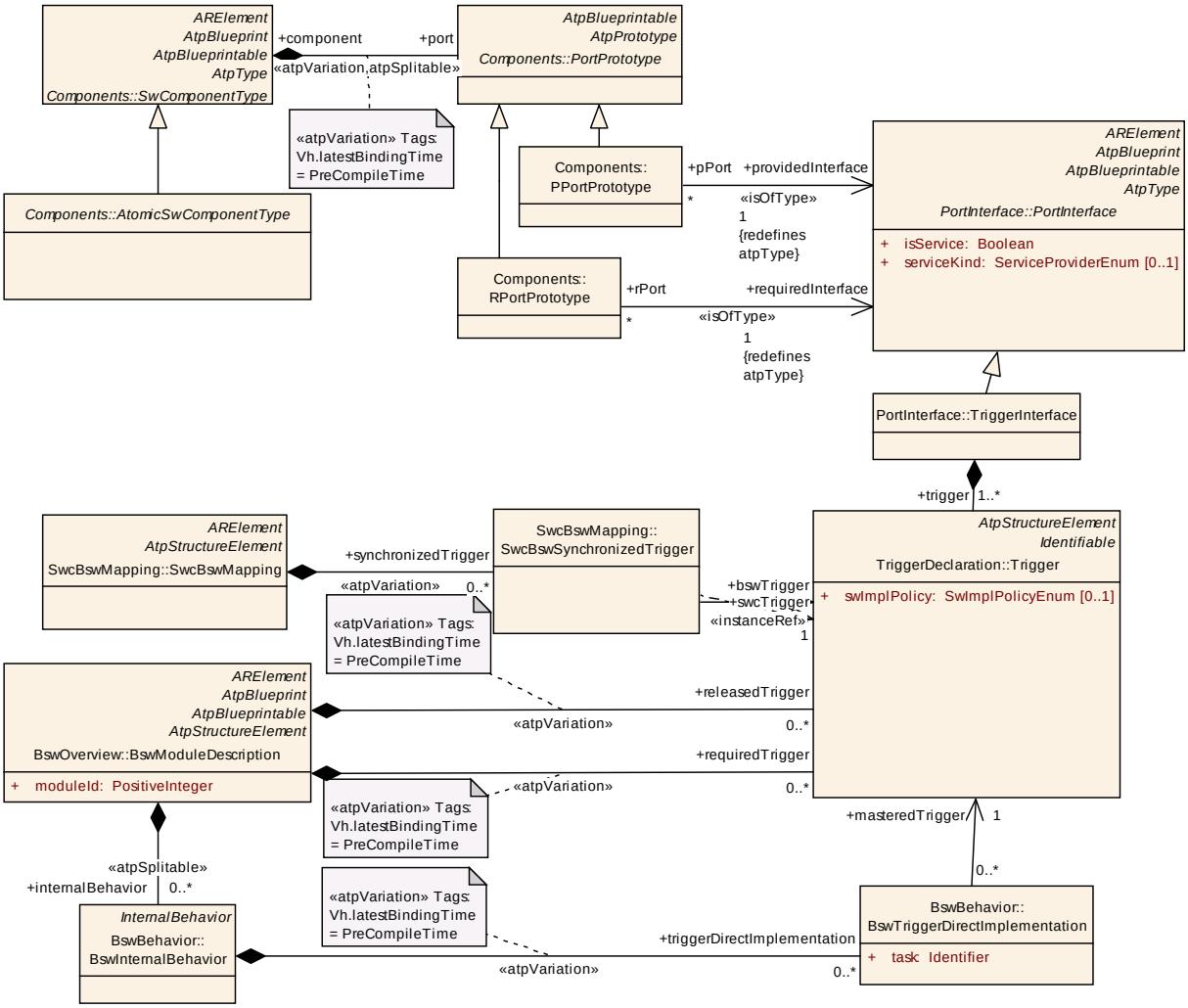


Figure 4.48: Summary of the use of Trigger by an AUTOSAR software-components and Basic Software Modules as defined in the *Software Component Template Specification* [2] and *Specification of BSW Module Description Template* [9].

[rte_sws_7212]⌈ The RTE shall support *External Trigger Event Communication*. ⌋*(RTE00162)*

[rte_sws_7542]⌈ The *Basic Software Scheduler* shall support the activation of *Basic Software Schedulable Entities* occurrence of External Trigger Events. ⌋*(RTE00216)*

4.5.1.2 Trigger Sink

A AUTOSAR software-component `Trigger Sink` has a dedicated require `trigger port`. The `trigger port` is typed by an *TriggerInterface* declaring one or more *Trigger*. See figure 4.48. The *Runnable Entities* of the of the software component are activated at the occurrence of the external event by the means of a `ExternalTriggerOccurredEvent`.

An *Basic Software Module* Trigger Sink has to define a *requiredTrigger Trigger*. The *Basic Software Schedulable Entities* of the of the *Basic Software Module* are activated at the occurrence of the external event by the means of a BswExternalTriggerOccurredEvent. See figure 4.48.

Basically there are two approaches to implement the activation of triggered ExecutableEntitys. In one case the triggered ExecutableEntitys of the Trigger Sinks triggered by one Trigger of the Trigger Source are mapped in one or more tasks. In this case the event communication can be implemented by the means of activating an Operating System Task.

[rte_sws_7213]⌈ The RTE generator shall support invocation of triggered ExecutableEntitys via OS Task. ⌋*(RTE00162, RTE00216)*

In the other case the Event Communication is mapped to a function call which means that the triggered ExecutableEntitys of the Trigger Sinks are executed in the Rte_Trigger API respectively SchM_Trigger API used to raise the trigger event in the Trigger Sinks.

[rte_sws_7214]⌈ The RTE generator shall support invocation of triggered ExecutableEntitys via direct function call, if all of the follwing conditions are fulfilled:

- the triggered ExecutableEntitys do not define a 'minimum start distance'
- the Trigger Sink and Trigger Source are in the same Partition
- if no *BswTriggerDirectImplementation* is defined.
- if the preconditions of table 4.5 are fulfilled
- no queuing for the Trigger Source is configured

⌋*(RTE00162, RTE00216)*

4.5.1.3 Trigger Source

An AUTOSAR software-component Trigger Source has a dedicated provide trigger port. The trigger port is typed by an *TriggerInterface* declaring one or more *Trigger*. See figure 4.48. To be able to connect a provide trigger port and a require trigger port, both ports must be categorized by the same or by compatible *TriggerInterface*(s).

An *Basic Software Module* Trigger Source has to define a *releasedTrigger Trigger*. See figure 4.48. The connection of *releasedTrigger* and *requiredTrigger Trigger* is defined by the ECU configuration of the *Basic Software Scheduler*.

To inform the RTE about an occurrence of the external trigger event the RTE provides the Rte_Trigger to an AUTOSAR software-component Trigger Source.

[rte_sws_7543]⌈ The call of the Rte_Trigger API shall activate all *Runnable Entities* that are activated by *ExternalTriggerOccurredEvent*s associated to a connected *Trig-*

ger of the `Trigger Source` if either no queuing for the `Trigger` is configured or if queuing for the `Trigger` is configured and the trigger queue is empty. ⌋*(RTE00162)*

For Basic Software Module `Trigger Source` are two options defined to interfaces with *Basic Software Scheduler*.

The first option is that the *Basic Software Module* `Trigger Source` inform the *Basic Software Scheduler* about an occurrence of the external trigger event by the call of the `SchM_Trigger` API.

[rte_sws_7544]⌈ The call of the `SchM_Trigger` API shall activate all `ExecutableEntity`s that are activated by *ExternalTriggerOccurredEvent*s associated to a connected *Trigger* of the `Trigger Source` if either no queuing for the `Trigger` is configured or if queuing for the `Trigger` is configured and the trigger queue is empty. ⌋*(RTE00216)*

The second option is that the *Basic Software Module* `Trigger Source` directly takes care about the activation of the particular OS task to which the *ExternalTriggerOccurredEvent*s of the `triggered ExecutableEntitys` are mapped. In this case the `Trigger Source` has to define a *BswTriggerDirectImplementation*. The name of the used OS tasks is annotated by the *task* attribute. If an *BswTriggerDirectImplementation* is defined no `SchM_Trigger` API is generated by the RTE generator. see rte_sws_7548 and rte_sws_7264.

[rte_sws_7545]⌈ The RTE generator shall reject configurations where a *BswTriggerDirectImplementation* is specified and an `ExecutableEntity` that is activated by an *ExternalTriggerOccurredEvent* associated to a connected *Trigger* of the `Trigger Source` is mapped to an OS task different from the one defined by the task attribute of the *BswTriggerDirectImplementation*. ⌋*(RTE00216, RTE00018)*

[rte_sws_7548]⌈ The RTE generator shall reject configurations where a *issuedTrigger* association and a *BswTriggerDirectImplementation* is defined for the same *releasedTrigger Trigger*. ⌋*(RTE00216, RTE00018)*

[rte_sws_ext_7547] A *releasedTrigger Trigger* shall not be referenced by both a *issuedTrigger* and a *BswTriggerDirectImplementation*.

Note: This shall ensure in the combination with the existence conditions (rte_sws_7264) of the `SchM_Trigger` that either the Trigger API or the direct task activation is offered to the implementation of the `Trigger Source`.

Note also that several OS tasks might be used to implement a `Trigger` (several `BswTriggerDirectImplementation` can be defined for a *releasedTrigger*).

If the `BswTriggerDirectImplementation` is defined for a *releasedTrigger* which `swImplPolicy` attribute is set to `queued` it is part of the `Trigger Source` to implement the queue or to use the means of the OS (OsTaskActivation > 1) to queue the number of raised triggers. (OsTaskActivation > 1). Further details about queuing of triggers is described in 4.5.4.

4.5.1.4 Multiplicity

4.5.1.4.1 Multiple Trigger

A trigger interface contains one or more `Trigger`. A port of an AUTOSAR software-component that provides an AUTOSAR trigger interface to the component can independently raise events related to each `Trigger` defined in the interface .

[rte_sws_7215]⌈ The RTE API shall support independent event raising for each `Trigger` in a trigger interface. ⌋*(RTE00162)*

Further on a *Basic Software Module* `Trigger Source` can define several *releasedTrigger Trigger* which can be independently raised.

[rte_sws_7546]⌈ The *Basic Software Scheduler* API shall support independent event raising for each *releasedTrigger* `Trigger`. ⌋*(RTE00216)*

4.5.1.4.2 Multiple Trigger Sinks Single Trigger Source

The concept of external event communication supports, that a `Trigger Source` activates one or more `triggered ExecutableEntitys` in one or more `Trigger Sinks`.

[rte_sws_7216]⌈ The RTE generator shall support `triggered ExecutableEntitys` triggered by the same `Trigger` of a `Trigger Source` ('1:n' communication where n $\geq$ 1). ⌋*(RTE00162, RTE00216)*

The execution order of the `triggered ExecutableEntitys` in the trigger sinks depends from the `RteEventToTaskMapping` described in chapter 7.6.1 and the configured priorities of the operating system.

4.5.1.4.3 Multiple Trigger Sources Single Trigger Sink

The RTE generator does support multiple `Trigger Sources` communicating events to the same `Trigger` in a `Trigger Sink` ('n:1' communication where n $\geq$ 1).

[rte_sws_7039]⌈ The RTE generator shall reject configurations where multiple `Trigger Sources` communicating events to the same `Trigger` in a `Trigger Sink` ('n:1' communication where n $\geq$ 1). ⌋*(RTE00018)*

[rte_sws_ext_7040] The same `Trigger` in a *Trigger Sink* must not be connected to multiple *Trigger Source*s.

4.5.1.5 Synchronized Trigger

If two *Trigger*s are synchronized by the definition of a *SwcBswSynchronizedTrigger* then the *Trigger* in the referenced provide `trigger port` and the referenced *releasedTrigger Trigger* are treated as one common *Trigger*. This means that all `ExecutableEntity`s activated by an *ExternalTriggerOccurredEvent* associated to one of the connected *Trigger*s are activated together.

[rte_sws_7218]⌈ The RTE and *Basic Software Scheduler* shall activate together all `ExecutableEntity`s that are activated by *ExternalTriggerOccurredEvent*s associated to a synchronized connected *Trigger*. ⌋*(RTE00162, RTE00216, RTE00217)*

[rte_sws_7549]⌈ The RTE generator shall reject configurations where a synchronized *Trigger* is referenced by more than one type of access method, where the type is one of the following:

1. *ExternalTriggeringPoint*
2. *issuedTrigger*
3. *BswTriggerDirectImplementation*

⌋*(RTE00216, RTE00217, RTE00018)*

[rte_sws_ext_7550] A synchronized *Trigger* shall only be referenced by either *ExternalTriggeringPoint*s, *issuedTrigger*s or *BswTriggerDirectImplementation*s.

Note: This shall ensure in the combination with the existence conditions of the `Rte_Trigger` and `SchM_Trigger` that only one kind of Trigger API (rte_sws_7201 and rte_sws_7264) or the direct task activation is offered to the implementation of the `Trigger Source`.

4.5.2 Inter Runnable Triggering

With the mechanism of *Inter Runnable Triggering* one *Runnable Entity* is able to request the activation of *Runnable Entities* of the same software-component instance.

[rte_sws_7220]⌈ The RTE shall support Inter Runnable Triggering. ⌋*(RTE00163)*

Similar to External Trigger Event Communication (described in chapter 4.5.1) the activation of triggered runnables can be implemented by means of activating an Operating System Task or by direct function call.

[rte_sws_7555]⌈ The call of the `Rte_IrTrigger` API shall activate all `triggered runnables` which *InternalTriggerOccurredEvent*s are associated with the related *InternalTriggeringPoint* of the same software-component instance if either no queuing

for the InternalTriggeringPoint is configured or if queuing for the InternalTriggeringPoint is configured and the trigger queue is empty. ⌋*(RTE00163)*

[rte_sws_7221]⌈ The RTE shall support for Inter Runnable Triggering that triggered runnables entities are invoked via OS Task activation. ⌋*(RTE00163)*

[rte_sws_7224]⌈ The RTE shall support for *Inter Runnable Triggering* that triggered runnables are invoked via direct function call if all of the following conditions are fulfilled:

- none of the triggered Basic Software Schedulable Entitys activated by this InternalTriggeringPoint define a 'minimum start distance'
- no queuing for the InternalTriggeringPointis configured

⌋*(RTE00163)*

4.5.2.1 Multiplicity

A InternalTriggeringPoint might be referenced by more than one InternalTriggerOccurredEvent. Therefore one RunnableEntity is able to request the activation of several RunnableEntity's with the mechanism of Inter Runnable Triggering contemporaneously.

[rte_sws_7223]⌈ The RTE shall support multiple RunnableEntity's triggered by the same InternalTriggeringPoint ('1:n' Inter Runnable Triggering where n $\geq$ 1). ⌋*(RTE00163)*

The execution order of the runnable entities in the trigger sinks depends from the Runnable Entity to task mapping described in chapter 7.6.1 and the configured priorities of the operating system.

4.5.3 Inter Basic Software Module Entity Triggering

The *Inter Basic Software Module Entity Triggering* is similar to the mechanism of *Inter Runnable Triggering* (see chapter 4.5.2) with the exception that it is used inside a Basic Software Module. It can be used to request the activation of a *Basic Software Schedulable Entity* by a *Basic Software Entity* of the same a *Basic Software Module* instance.

[rte_sws_7551]⌈ The *Basic Software Scheduler* shall support *Inter Basic Software Module Entity Triggering*. ⌋*(RTE00230)*

Similar to External Trigger Event Communication (described in chapter 4.5.1) the activation of triggered *Basic Software Schedulable Entity* can be implemented by means of activating an Operating System Task or by direct function call.

[rte_sws_7552]⌈ The call of the `SchM_ActMainFunction` API shall activate all `triggered Basic Software Schedulable Entitys` which *BswInternalTriggerOccurredEvent*s are associated by the related *activationPoint* of the same a *Basic Software Module* instance if either no queuing for the `BswInternalTriggeringPoint` is configured or if queuing for the `BswInternalTriggeringPoint` is configured and the trigger queue is empty.. ⌋*(RTE00230)*

[rte_sws_7553]⌈ The *Basic Software Scheduler* shall support for *Inter Basic Software Module Entity Triggering* that `triggered Basic Software Schedulable Entitys` are invoked via OS Task activation. ⌋*(RTE00230)*

[rte_sws_7554]⌈ The *Basic Software Scheduler* shall support for *Inter Basic Software Module Entity Triggering* that `triggered Basic Software Schedulable Entitys` are invoked via direct function call if

- the `triggered Basic Software Schedulable Entitys` do not define a 'minimum start distance'
- if the preconditions of table 4.5 are fulfilled
- no queuing for the `BswInternalTriggeringPoint`is configured

⌋*(RTE00230)*

Note: Typically the feature of *Inter Basic Software Module Entity Triggering* is used to decouple the execution context of *Basic Software Entities*. But if this decoupling is really required depends from the particular scheduling concept and microcontroller performance.

4.5.4 Queuing of Triggers

The queuing of triggers ensures that the number of executions of `triggered ExecutableEntitys` is equal to the number of released triggers. Further on it ensures that the number of activations of `triggered ExecutableEntitys` is equal for all associated `triggered ExecutableEntitys` of a `Trigger Emitter` if the associated `triggered ExecutableEntitys` are not activated by other `RTEEvents`. Therefore the trigger queue is rather a counter than a real queue.

[rte_sws_7087]⌈ The RTE shall support the queuing of triggers for

- *External Trigger Event Communication*
- *Inter Runnable Triggering*
- *Inter Basic Software Module Entity Triggering*

if the RteTriggerSourceQueueLength / RteBswTriggerSourceQueueLength is configured > 0. ⌋*(RTE00235)*

The attribute swImplPolicy specifies a queued or non queued processing of the Trigger Emitter. Since the setup of a queue might have other side effects on the dynamic behavior of the ECU its still an design decision of the ECU integrator to configure a trigger queue.
Therefore it is possible to configure a trigger queue regardless on the value of the attribute swImplPolicy of the Trigger Emitter.

[rte_sws_7088]⌈ The RTE shall enqueue a trigger when the RTE gets informed about the occurrence of a trigger by the call of the related API (Rte_IrTrigger, Rte_Trigger, SchM_Trigger, SchM_ActMainFunction) if queuing for this Trigger Emitter is configured and if the maximum queue length (RteTriggerSourceQueueLength / RteBswTriggerSourceQueueLength) is not exceeded. ⌋*(RTE00235)*

[rte_sws_7089]⌈ The RTE shall dequeue a trigger when the Trigger Emitter is informed about the end of execution of all triggered ExecutableEntitys which are triggered by this Trigger Emitter. ⌋*(RTE00235)*

[rte_sws_7090]⌈ The RTE shall activate all triggered ExecutableEntitys associated to a Trigger Emitter when it has successfully dequeued a trigger from the trigger queue of the Trigger Emitter except for the last dequeued trigger. ⌋*(RTE00235)*

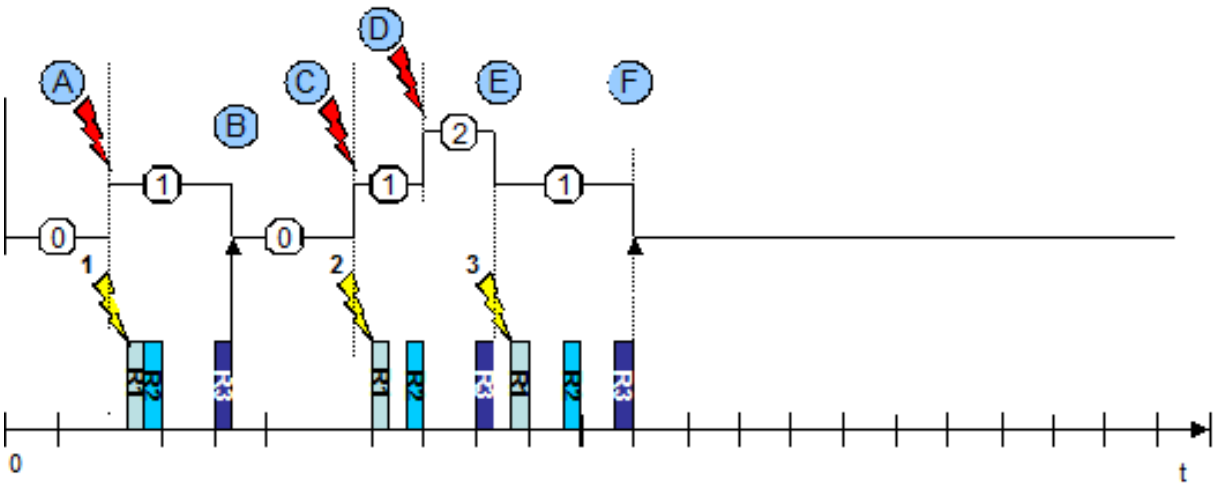


Figure 4.49: Queued activation of ExecutableEntitys

The figure 4.49 illustrates the basic behavior of a trigger queue.

- At "'A"' the RTE gets informed by the call of the API about the occurrence of a Trigger. Since no trigger is in the queue all associated triggered ExecutableEntitys are activated (rte_sws_7544, rte_sws_7555, rte_sws_7552) and the trigger is enqueued (rte_sws_7088).

- At "'B"' all `triggered ExecutableEntitys` which are triggered by this `Trigger Emitter` have terminated. The RTE dequeues the trigger but since it is the last dequeued trigger the associated `triggered ExecutableEntitys` are not activated again.
- At "'C"' the RTE gets informed by the call of the API about the occurrence of a Trigger. Enqueuing of triggers and activating of `triggered ExecutableEntitys` is done as in "'A"'
- At "'D"' the RTE gets informed again by occurrence of a trigger. Since a trigger is already in the queue the associated `triggered ExecutableEntitys` are not activated (rte_sws_7544, rte_sws_7555, rte_sws_7552). Nevertheless the trigger is enqueued (rte_sws_7088).
- At "'E"' all `triggered ExecutableEntitys` which are triggered by this `Trigger Emitter` have terminated. The RTE dequeues the trigger (rte_sws_7089) and activates all associated `triggered ExecutableEntitys` (rte_sws_7090).
- At "'E"' all `triggered ExecutableEntitys` which are triggered by this `Trigger Emitter` have terminated. Dequeuing of triggers is done as in "'B"'

Implementation hint:
One possible solution to implement the queue for the number of released triggers is to use the means of the operation systems which already can queue the activation requests for a OS task (`OsTaskActivation` > 1). This for sure is only possible if all `ExternalTriggerOccurredEvent`s, `InternalTriggerOccurredEvent`s, `BswExternalTriggerOccurredEvent` and `BswInternalTriggerOccurredEvent` connected to the same `Trigger Emitter` with configured queuing are mapped exclusively to one OS task.

4.5.5 Activation of triggered ExecutableEntities

The activation of `triggered ExecutableEntitys` is done like described in chapter 4.2.3. See also Fig. 4.15.

If the `triggered ExecutableEntitys` are activated synchronous or asynchronous depends how the *RTEEvent*s and *BswEvent*s are mapped to OS tasks.

If all *ExternalTriggerOccurredEvent*s of the `Trigger Sinks` which are associated to connected *Trigger* of the `Trigger Source`

- either are mapped to OS task(s) with higher priority as the OS task where the *Executable Entity* calling the `Rte_Trigger` respectively the `SchM_Trigger` API is mapped
- or are activated by direct function call

the triggering behaves synchronous. This means that all "triggered" *Executable Entities* of the `Trigger Sinks` are executed before the `Rte_Trigger` or `SchM_Trigger` API returns.

If any *ExternalTriggerOccurredEvent* of the `Trigger Sinks` which are associated to connected *Trigger* of the `Trigger Source`

are mapped to an OS task with lower priority as the OS task where the *Executable Entity* calling the `Rte_Trigger` respectively the `SchM_Trigger` API is mapped the triggering behaves asynchronous. This means that **not** all `triggered ExecutableEntitys` of the `Trigger Sinks` are executed before the `Rte_Trigger` or `SchM_Trigger` API returns.

4.6 Initialization and Finalization

4.6.1 Initialization and Finalization of the RTE

RTE and *Basic Software Scheduler* have a nested life cycle. It is only permitted to initialize the RTE if the *Basic Software Scheduler* is initialized (rte_sws_ext_7577). Further on it is only supported to finalize the *Basic Software Scheduler* after the RTE is finalized (rte_sws_ext_7576).

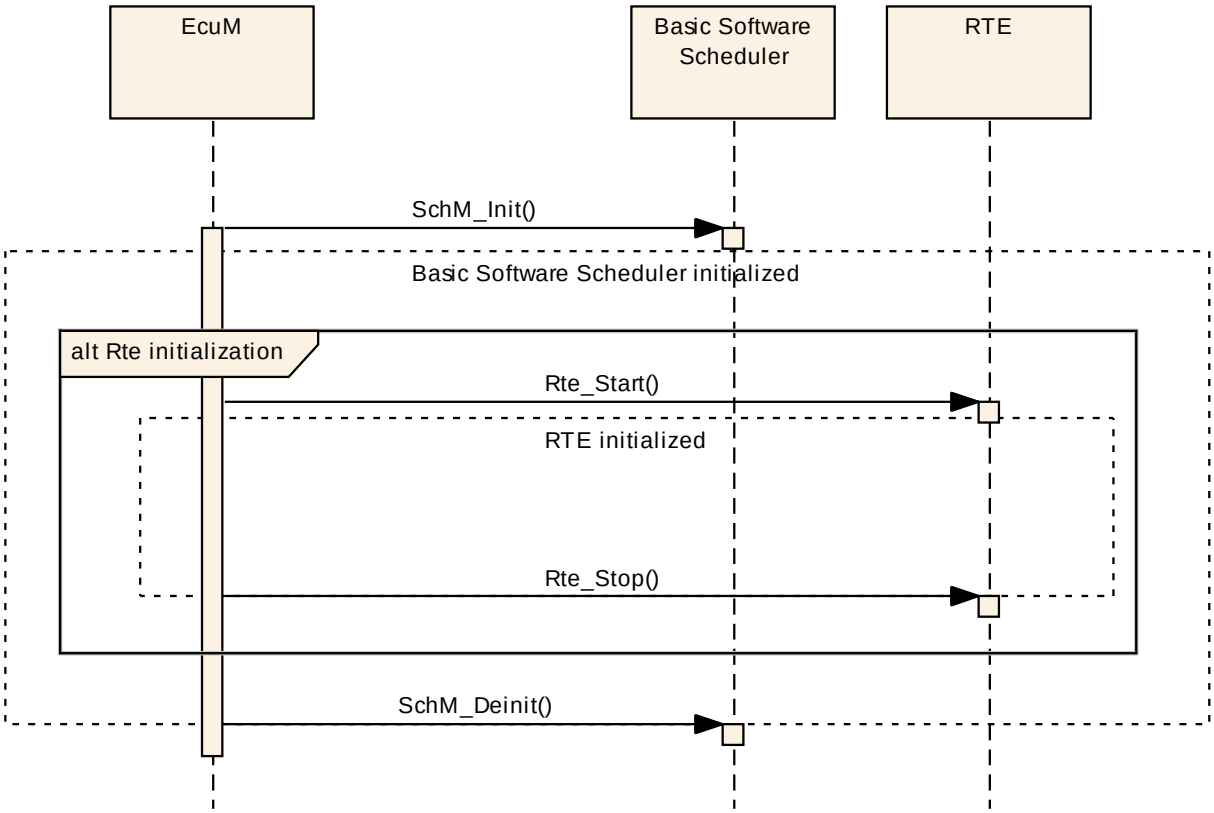


Figure 4.50: Nested life cycle of RTE and *Basic Software Scheduler*

4.6.1.1 Initialization of the Basic Software Scheduler

Before the *Basic Software Scheduler* is initialized only the API calls `SchM_Enter` and `SchM_Exit` are available (rte_sws_7578).

The ECU state manager calls the startup routine `SchM_Init` of the *Basic Software Scheduler* before any *Basic Software Module* needs to be scheduled.

The initialization routine of the *Basic Software Scheduler* will return within finite execution time (see rte_sws_7273).

The *Basic Software Scheduler* will initialize the `mode machine instances` (rte_sws_2544)assigned to the *Basic Software Scheduler*. This will activate the `mode disablings` of all initial modes during `SchM_Init` and trigger the execution of the

OnEntry ExecutableEntitys of the initial modes. After initialization of the *Basic Software Scheduler* internal data structure and mode machine instances the activation of *Basic Software Schedulable Entities* triggered by BswTimingEvents starts.

[rte_sws_7574]⌈ The call of SchM_Init shall start the activation of BswSchedulableEntitys triggered by BswTimingEvents. ⌋*(RTE00211)*

[rte_sws_7584]⌈ The call of SchM_Init shall start the activation of BswSchedulableEntitys triggered by BswBackgroundEvents. ⌋*(RTE00211)*

Note: In case of OS task where BswEvents and RTEEvents are mapped to the RTE Generator has to ensure, that RunnableEntitys are not activated before the RTE is initialized or after the RTE is finalized. See rte_sws_7580 and rte_sws_2538.

[rte_sws_7580]⌈ The *Basic Software Scheduler* has to prevent the activation of RunnableEntitys before the RTE is initialized. ⌋*(RTE00220)*

4.6.1.2 Initialization of the RTE

The ECU state manager calls the startup routine Rte_Start of the RTE at the end of startup phase II when the OS is available and all basic software modules are initialized.

The initialization routine of the RTE will return within finite execution time (see rte_sws_2585).

Before the RTE is initialized completely, there is only a limited capability of RTE to handle incoming data from COM:

The RTE will initialize the mode machine instances (rte_sws_2544) assigned to the RTE. This will activate the mode disablings of all initial modes during Rte_Start and trigger the execution of the OnEntry ExecutableEntitys of the initial modes. Further on for common mode machine instances the *OnEntry Runnable Entities* of the current active mode are executed during the initialization of the RTE (rte_sws_7582). common mode machine instances can not enter the transition phase during RTE initialization (rte_sws_7583).

[rte_sws_7575]⌈ The call of Rte_Start shall start the activation of RunnableEntitys triggered by TimingEvents. ⌋*(RTE00072)*

[rte_sws_7178]⌈ The call of Rte_Start shall start the activation of RunnableEntitys triggered by BackgroundEvents. ⌋*(RTE00072)*

[rte_sws_7615]⌈ The call of Rte_Start shall be executed on every core independently. ⌋*()*

[rte_sws_7616]⌈ The Rte_Start includes the partition specific startup activities of RTE for all partitions that are mapped to the core, from which the Rte_Start is called. ⌋*()*

4.6.1.3 Stop and restart of the RTE

Partitions of the ECU can be stopped and restarted. In a stopped or restarting partition, the OS has killed all running tasks. RTE has to react to stopping and restarting partitions.

The RTE does not execute ExecutableEntities of a terminated or restarting partition.

[rte_sws_7604]⌈ The RTE shall not activate, start or release `ExecutableEntity execution-instances` of a terminated or restarting partition. ⌋*(RTE00195)*

The RTE is notified of the termination (respectively, the beginning of restart) of a partition by the `Rte_PartitionTerminated` (respectively, `Rte_PartitionRestarting`) API. At this point in time, the tasks containing the runnables of this partition are already killed by the OS. In case of restart, RTE is notified by the `Rte_RestartPartition` API when the communication can be re-initialized and re-enabled.

rte_sws_7604 also applies to ExecutableEntities whose execution started before the notification to the RTE. RTE can rely on the OS functionality to stop or restart an OS application and all related OS objects.

When a partition is restarted, the RTE will restore an initial environment for its SW-Cs.

[rte_sws_2735]⌈ When the `Rte_RestartPartition` API for a partition is called, the RTE shall restore an initial environment for its SW-Cs on this partition. ⌋*()*

The SW-Cs themselves are responsible to restore their internal initial environment and should not rely on any initialization performed by the compiler. This should be done in initialization runnables.

[rte_sws_7610]⌈ The RTE Generator shall reject configurations where the `handleTerminationAndRestart` attribute of a SW-C is not set to `canBeTerminatedAndRestarted` and this SW-C is mapped on a `Partition` with the `PartitionCanBeRestarted` parameter set to TRUE. ⌋*(RTE00018, RTE00196)*

When a partition is terminated or is being restarted, it is important that the runnable entities of this partition are not activated before the partition returns to the ACTIVE state.

In case of partition restart or termination, event sent to this partition or activation of tasks of this partition are discarded. The RTE can use these mechanism to ensure that ExecutableEntities are not activated.

4.6.1.4 Finalization of the RTE

The finalization routine `Rte_Stop` of the RTE is called by the ECU state manager at the beginning of shutdown phase I when the OS is still available. (For details of the ECU state manager, see [7]. For details of `Rte_Start` and `Rte_Stop` see section 5.8.)

[rte_sws_2538]⌈ The RTE shall not activate, start or release `RunnableEntity`s on a core after `Rte_Stop` has been called on this core. ⌋*(RTE00116, RTE00220)*

Note: RTE does not kill the tasks during the 'running' state of the runnables.

[rte_sws_2535]⌈ RTE shall ignore incoming client server communication requests, before RTE is initialized completely and when it is stopped. ⌋*(RTE00116)*

[rte_sws_2536]⌈ Incoming data and events from sender receiver communication shall be ignored, before RTE is initialized completely and when it is stopped. ⌋*(RTE00116)*

4.6.1.5 Finalization of the *Basic Software Scheduler*

The ECU state manager calls the finalization routine `SchM_Deinit` of the *Basic Software Scheduler* if the scheduling of *Basic Software Module*s has to be stopped.

[rte_sws_7586]⌈ The BSW Scheduler shall neither activate nor start `SchedulableEntity`s on a core after `SchM_Deinit` has been called on this core. ⌋*(RTE00116)*

Note: The BSW Scheduler does not kill the tasks during the 'running' state of the `SchedulableEntity`s.

4.6.2 Initialization and Finalization of AUTOSAR Software-Components

For the initialization and finalization of AUTOSAR software components, RTE provides the mechanism of mode switches. A `SwcModeSwitchEvent` of an appropriate `ModeDeclaration` can be used to trigger a corresponding initialization or finalization runnable (see rte_sws_2562). Runnables that shall not run during initialization or finalization can be disabled in the corresponding modes with a `ModeDisablingDependency` (see rte_sws_2503).

Since category 2 runnables have no predictable execution time and can not be terminated using `ModeDisablingDependencies`, it is the responsibility of the implementer to set meaningful termination criteria for the cat 2 runnables. These criteria could include mode information. At latest, all runnables will be terminated by RTE during the shutdown of RTE, see rte_sws_2538.

It is appropriate to use user defined modes that will be handled in a proprietary `application mode manager`.

All runnables that are triggered by entering an initial mode, are activated immediately after the initialization of RTE. They can be used for initialization. In many cases it might

be preferable to have a multi step initialization supported by a sequence of different initialization modes.

4.7 Variant Handling Support

4.7.1 Overview

The *AUTOSAR Template*s support the creation of *Variant*s in a subset of its model elements. The *Variant Handling* support in the in *AUTOSAR Template*s is driven by the purpose to describe variability in a *AUTOSAR System* on several aspects, e.g.

- Virtual Functional Bus
- Component `SwcInternalBehavior` and `SwcImplementation`
- Deployment of the software components to ECUs
- Communication Matrix
- Basic Software Modules

This approach requires that the RTE Generator is able to process the described Variability in input configurations and partially to implement described variability in the generated RTE and Basic Software Scheduler code.

In the meta-model all locations that may exhibit variability are marked with the stereotype `atpVariation`. This allows the definition of possible variation points. Tagged Values are used to specify additional information.

There are four types of locations in the meta-model which may exhibit variability:

- Aggregations
- Associations
- Attribute Values
- Classes providing property sets

More details about the AUTOSAR Variant Handling Concept can be found in the AUTOSAR Generic Structure Template [10].

[rte_sws_6543]⌈ The RTE generator shall support the `VariationPoint`s defined in the *AUTOSAR Meta Model* ⌋*(RTE00201, RTE00202, RTE00229, RTE00191)*

The list of `VariationPoint`s shall provide an overview about the most prominent ones which impacting the generated RTE code. Further on tables will show which implementation of variability is standardized due to the relevance for contract phase. (see tables 4.13, 4.15, 4.16, 4.17, 4.18, 4.19, 4.21, 4.22, 4.24 and 4.25. But please note that these tables are not listing all possible variation of the input configuration. For that the related Template Specifications are relevant.

4.7.2 Choosing a Variant and Binding Variability

To understand the later definition it is required to clarify the difference between *Choosing a Variant* and *Resolving Variability*.

A particular *PreBuild Variant* in a variant rich input configuration is chosen by assigning particular values to the SwSystemconsts with the means of PredefinedVariants and associated SwSystemconstantValueSets. With this information SwSystemconstDependentFormulas can be evaluated which determines PreBuild conditions of VariationPoints and attribute values. Nevertheless the input configuration contains still the information of all potential variants.

A particular *PostBuild Variant* in a variant rich input configuration is chosen by assigning particular values to the PostBuildVariantCriterion with the means of PredefinedVariants and associated PostBuildVariantCriterionValueSets. With this information PostBuildVariantConditions can be evaluated for instance to check the consistency of chosen *PostBuild Variant*. Nevertheless the input configuration contains still the information of all potential variants.

From an RTE perspective this information is manly used to generate the *RTE Post Build Variant Sets* which are used to bind the PostBuild Variability during initialization of the RTE (call of SchM_Init).

The variability of an input configuration is bound if information related to other variants is removed and only the information of the bound variant is kept. Binding respectively resolving variability in the scope of this specification means that the generated code only implements the particular variant which results out of the chosen variant of the input configuration.

If the variability can not be resolved in a particular phase of the *RTE Generation Process* (see chapter 3) the generated RTE files have to be able to support the potential variants by implementing all potential variants.

If the variability is relevant for the software components contract the RTE Generator uses standardized *Condition Value Macros* to implement the PreBuild Variability. These *Condition Value Macros* are set in the *RTE PreBuild Data Set Contract Phase* and *RTE PreBuild Data Set Generation Phase* to the resulting value of the evaluated ConditionByFormula of the related VariationPoint.

For further definition see sections 4.7.2.3, 4.7.2.4, 4.7.2.5, 4.7.2.6 and 4.7.2.7.

4.7.2.1 General impact of Binding Times on RTE generation

Each VariationPoint has an attribute bindingTime, which defines the latest binding time for this variation point. This controls the capability of the software implementation to bind the variant at latest at a certain point of time. Even if the variability is chosen earlier for instance by assigning SwSystemconstValues to the SwSystemconsts used by the VariationPoints condition the RTE generator has to respect potential

latest binding for VariationPoints supporting the latest binding time. Please note variability with the latest binding time PreCompileTime and PostBuild have a particular semantic for the RTE generation and impacts the generated output. For instance a conditional existence RTE API which is bound at PreCompileTime requires that the RTE generator inserts specific pre processor statements.

RTE Phase	System Designe Time	Code Generation Time	Pre Compile Time	Link Time	Post Build
RTE Contract Phase	R	R	I	n/a	n/a
Basic Software Scheduler Contract Phase	R	R	I	n/a	n/a
RTE PreBuild Data Set Contract Phase	n/a	n/a	RV	n/a	n/a
Basic Software Scheduler Generation Phase	R	R	I	n/a	I
RTE Generation Phase	R	R	I	n/a	I
RTE PreBuild Data Set Generation Phase	n/a	n/a	RV	n/a	n/a
RTE PostBuild Data Set Generation Phase	n/a	n/a	n/a	n/a	RV

Table 4.12: Overview impact of Binding Times on RTE generation

R	resolve variability, a particular variant is the output
I	implement variability, all possible variants in the output
RV	provide values to resolve implemented variability *PreBuild* or *PostBuild*
n/a	not applicable

4.7.2.2 Choosing a particular variant

A particular variant of the variant rich input configuration is chosen via the ECU configuration For that purpose a set of PredefinedVariants is configured to chosen a variant in the input configuration and to later on bind the variability in subsequent phases of the *RTE Generation Process* 3. For further information see document [10].

[rte_sws_6500]⌈ For each PreBuild Variability in the input configuration the RTE Generator shall choose a particular variant according to the PredefinedVariants selected by the parameter EcucVariationResolver. ⌋*(RTE00201, RTE00202, RTE00229, RTE00191)*

[rte_sws_6546]⌈ For each PostBuild Variability in the input configuration the RTE Generator shall choose a particular variant according to the PredefinedVariants selected by the parameter RtePostBuildVariantConfiguration. ⌋*(RTE00201, RTE00202, RTE00229, RTE00191)*

Having variants chosen the RTE generator can apply further consistency checks on the particular variants.

4.7.2.3 SystemDesignTime

Variability with latest binding time `SystemDesignTime` (called `SystemDesignTime Variability`) has to be bound before the *RTE Contract Phase* respectively *Basic Software Scheduler Contract Phase*. Such variability is resolved by RTE generator in all generation phases. Due to that such kind of variability results always in a particular variant and needs no special code generation rules for RTE generator.

[rte_sws_6501]⌈ The RTE generator shall bind `SystemDesignTime Variability` in the *RTE Contract Phase*, *Basic Software Scheduler Contract Phase*, *RTE Generation Phase* and *Basic Software Scheduler Generation Phase* (3). ⌋*(RTE00191)*

[rte_sws_6502]⌈ The RTE Generator shall reject input configurations during the *RTE Contract Phase* where not a particular variant is chosen for each `SystemDesignTime Variability` affecting the software components contract. ⌋*(RTE00201, RTE00018)*

[rte_sws_6503]⌈ The RTE Generator shall reject input configurations during the *Basic Software Scheduler Contract Phase* where not a particular variant is chosen for each `SystemDesignTime Variability` affecting the *Basic Software Scheduler* contract. ⌋*(RTE00229, RTE00018)*

[rte_sws_6504]⌈ The RTE Generator shall reject input configurations during the *Basic Software Scheduler Generation Phase* where not a particular variant is chosen for each `SystemDesignTime Variability` affecting the *Basic Software Scheduler* generation. ⌋*(RTE00229, RTE00018)*

[rte_sws_6505]⌈ The RTE Generator shall reject input configurations during the *RTE Generation Phase* where not a particular variant is chosen for each `SystemDesignTime Variability` affecting the *RTE* generation. ⌋*(RTE00201, RTE00202, RTE00018)*

4.7.2.4 CodeGenerationTime

During *RTE Contract Phase*, *RTE Generation Phase* and *Basic Software Scheduler Generation Phase* the variability with latest binding time *CodeGenerationTime* (called `CodeGenerationTime Variability`) has to be bound and the RTE generator resolves the variability. This denotes that the code is generated for a particular variant. To do this it is required that a particular variant for each `CodeGenerationTime Variability` has to be chosen.

[rte_sws_6507]⌈ The RTE generator shall bind `CodeGenerationTime Variability` in the *RTE Contract Phase*, *Basic Software Scheduler Contract Phase*, *RTE Gen-*

eration Phase and *Basic Software Scheduler Generation Phase* (see sections 3.1.1, 3.1.2, 3.4.1 and 3.4.2). ⌋*(RTE00229, RTE00191)*

[rte_sws_6547]⌈ The RTE Generator shall reject input configurations during the *RTE Contract Phase* where not a particular variant is chosen for each `CodeGenerationTime Variability` affecting the software components contract. ⌋*(RTE00191, RTE00018)*

[rte_sws_6548]⌈ The RTE Generator shall reject input configurations during the *Basic Software Scheduler Contract Phase* where not a particular variant is chosen for each `CodeGenerationTime Variability` affecting the *Basic Software Scheduler* contract. ⌋*(RTE00229, RTE00018)*

[rte_sws_6508]⌈ The RTE Generator shall reject input configurations during the *Basic Software Scheduler Generation Phase* where not a particular variant is chosen for each `CodeGenerationTime Variability` affecting the *Basic Software Scheduler* generation. ⌋*(RTE00229, RTE00018)*

[rte_sws_6509]⌈ The RTE Generator shall reject input configurations during the *RTE Generation Phase* where not a particular variant is chosen for each `CodeGenerationTime Variability` affecting the *RTE* generation. ⌋*(RTE00191, RTE00018)*

4.7.2.5 PreCompileTime

Variability with latest binding time *PreCompileTime* (called `PreCompileTime Variability`) is relevant for the *RTE Contract Phase* and *Basic Software Scheduler Contract Phase* as well as for the *RTE Generation Phase* and *Basic Software Scheduler Generation Phase*. The *Application Header File*, *Application Types Header File*, *Module Interlink Header* and *Module Interlink Types Header* and the generated RTE / *Basic Software Scheduler* has to support the potential variability of the software components and *Basic Software Modules*. The variability is resolved during the execution of the pre processor of the C-Complier.

[rte_sws_6510]⌈ The RTE generator shall implement `PreCompileTime Variability` in the *RTE Contract Phase*, *Basic Software Scheduler Contract Phase*, *RTE Generation Phase*, *Basic Software Scheduler Generation Phase* via pre processor statements in the generated RTE code (see sections 3.1.1, 3.1.2, 3.4.1 and 3.4.2). ⌋*(RTE00191)*

4.7.2.6 LinkTime

The latest Binding Time *LinkTime* will not be supported for *VariationPoint*s relevant for the RTE Generator.

[rte_sws_6511]⌈ The RTE generator shall reject configuration which defines RTE or *Basic Software Scheduler* relevant `LinkTime Variability`. ⌋*(RTE00018)*

4.7.2.7 PostBuild

Variability with latest binding time *PostBuild* (called `PostBuild Variability`) might be bound / rebound after the generated RTE is compiled and has been linked to the executable. The generated RTE binary code has to contain all variants. Which variant is executed during ECU runtime is decided by variant selectors.

[rte_sws_6512]⌈ The RTE generator shall implement `PostBuild Variability` in the *RTE Generation Phase* and *Basic Software Scheduler Generation Phase* via C statements in the generated RTE code (see 3.4.1 and 3.4.2). ⌋*(RTE00191)*

Combining PreBuild and PostBuild Variability

According document [10] it is supported that a `VariationPoint` defines a `PreBuild Variability` in conjunction with `PostBuild Variability`. If the *PreBuild condition* is false, it is not expected that the element which is subject to variability including the code evaluating the *PostBuild condition* gets implemented at all.

[rte_sws_6549]⌈ In cases where a `VariationPoint` defines a `SystemDesignTime Variability` or `CodeGenerationTime Variability` in conjunction with `PostBuild Variability` the `PostBuild Variability` shall only be implemented by the RTE Generator in the generated RTE code if the condition of the `PreBuild Variability` evaluates to true. ⌋*(RTE00191)*

[rte_sws_6550]⌈ In cases where a `VariationPoint` defines a `PreCompileTime Variability` in conjunction with `PostBuild Variability` the `PostBuild Variability` shall only be effective in the RTE executable if the condition of the `PreCompileTime Variability` evaluates to true. ⌋*(RTE00191)*

In this case the `PostBuild Variability` implemented according rte_sws_6512 depends from the `PreCompileTime Variability` implemented according rte_sws_6510.

4.7.3 Variability affecting the RTE generation

4.7.3.1 Software Composition

This section describes the affects of the existence of variation points with regards to compositions. Though the application software compositions have been flattened and effectively eliminated after allocation to an ECU there is still one composition to consider for the RTE (i.e. the `RootSwCompositionPrototype`). The `RootSwCompositionPrototype` contains the atomic software components allocated to the respective ECU, its assembly connections,its delegation connections and the connections of the delegation ports to system signals. Once the variability is resolved for a variation point it must adhere to the constraints and limitations that apply to a model that does not have any variations. For example dangling connectors are not allowed and as such their existence will lead to undefined behavior if such configurations still exist after resolving post-build variation points.

Also within this specification section the wording "'a variant is enabled or disabled"' refers to the variation point's *SwSystemconstDependentFormula* and/or *PostBuildVariantCondition* evaluating to "'true or false"' respectively.

4.7.3.1.1 Variant existence of `SwComponentPrototypes`

[rte_sws_6601]⌈ If a variant is disabled for the aggregation of a `SwComponentPrototype` in a `CompositionSwComponentType` then all RTEEvents destined for Runnables in the respective `SwComponentPrototype` shall be blocked;No RTEEvent is allowed to reach any Runnable that is contained in a "'disabled"' `SwComponentPrototype`. ⌋*(RTE00206, RTE00207, RTE00204)*

Potential misconfigurations of connectors connecting to ports of "'disabled"' SWC's will result in undefined behavior; It is the responsibility of the person considering the variability of the `SwComponentPrototype` to make the connections also variable and valid when a variant selection results in the elimination of a `SwComponentPrototype` from a composition. It is recommended to use predefined variants to ensure proper configurations are established.

4.7.3.1.2 Variant existence of `SwConnectors`

[rte_sws_6602]⌈ If a variant is disabled for a `SwConnector` (i.e. `AssemblySwConnector` or DelegationSwConnector) aggregated in a `CompositionSwComponentType` then the `PortPrototypes` at each end of the connector shall behave as an unconnected port (see section 5.2.7 for the defined RTE behavior) **if** no other variant enables a `SwConnector` between these ports. ⌋*(RTE00206, RTE00207)*

4.7.3.1.3 COM related Variant existence

This section describes the impact on the RTE interaction with the COM layer as a result of variability of DataMappings (i.e. *SenderReceiverToSignalMapping*,*SenderReceiverToSignalGroupMapping* and *ClientServerToSignalGroupMapping* in the *SystemMapping*) as well as the existence of variants for *ISignal*s The Meta Model allows for mapping the same data to different *SystemSignal*s as well as associating a *SystemSignal* with 1 or more *ISignals*.

[rte_sws_6603]⌈ If a variant is enabled for a *SystemMapping* aggregating a *DataMapping* then the RTE shall call the appropriate API's for the applicable mapping type. ⌋*(RTE00206, RTE00207)*

[rte_sws_6604]⌈ The appropriate API shall be determined based on the existence of variants of *ISignal*s to which a *SystemSignal* is associated to. For each enabled ISignal the RTE shall call the proper COM API to send and receive data *SystemSignal*s ⌋*(RTE00206, RTE00207)*

For example for an instance mapping from a VariableDataPrototype to a SystemSignal the RTE shall call the corresponding COM_SendSignal with the proper SignalId and SignalDataPtr based on the selected variant DataMapping.

[rte_sws_6605]⌈ Delegation ports on a `RootSwCompositionPrototype` for which no DataMapping exists (i.e. no variant DataMapping is enabled) shall be considered unconnected because no path exists to a designated SystemSignal. Since this is a delegation port all enabled delegation connectors linking SWC R-ports to the respective delegation port must be considered unconnected (see section 5.2.7). P-Ports shall behave as documented in section 4.7.3.1.2. ⌋*(RTE00206, RTE00207)*

4.7.3.1.4 Variant existence of *PortPrototypes*

[rte_sws_6606]⌈ If no variant is enabled for a delegation port on a `RootSwCompositionPrototype` then all connected R-Ports using a `DelegationSwConnector` to this delegation port shall be considered unconnected (see section 5.2.7). The behavior of the P-ports shall be as defined in section 4.7.3.1.2. ⌋*(RTE00206, RTE00207)*

Note on variant disabling criteria: In a proper variant configuration the following should be followed: when a PortPrototype is eliminated from any `SwComponentType` then any associated `SwConnector` should also have a variation point removing the connection since the connection is illegal.

4.7.3.2 Atomic Software Component and its Internal Behavior

4.7.3.2.1 RTE API which is subject to variability

Following VariationPoints in the Meta Model do control the variant existence of RTE API for a software component. If a RTE API is variant existent, the API mapping and the related entries in the component data structure are 'variant' as well. This means, if a RTE API does not exist the API mapping does not exist as well. A part of the component data structure entries are related to the existences of the port. In these cases the *component data structure entry* depends from the existence of the PortPrototype.

Variation Point Condition Value Macro	RTE API which is subject to variability	form	kind infix
ExclusiveArea rte_sws_6518	Rte_Enter, Rte_Exit	component internal	ExAr
VariableDataPrototype in the role arTyped-PerInstanceMemory rte_sws_6518	Rte_Pim	component internal	PIM
PerInstanceMemory rte_sws_6518	Rte_Pim	component internal	PIM
ParameterDataPrototype in the role perIn-stanceParameter rte_sws_6518	Rte_CData	component internal	Prm
ParameterDataPrototype in the role shared-Parameter rte_sws_6518	Rte_CData	component internal	Prm
ServerCallPoint rte_sws_6515	Rte_Call	component port	
AsynchronousServerCallResultPoint rte_sws_6515	Rte_Result	component port	
InternalTriggeringPoint rte_sws_6519	Rte_IrTrigger	entity internal	IRT
ExternalTriggeringPoint rte_sws_6515	Rte_Trigger	component port	
ModeSwitchPoint rte_sws_6515	Rte_Switch, Rte_SwitchAck	component port	
ModeAccessPoint rte_sws_6515	Rte_Mode	component port	
VariableAccess in the role dataReadAccess	Rte_IRead , Rte_IStatus, Rte_IsUpdated	entity port	

rte_sws_6515			
`VariableAccess` in the role `dataWriteAccess` rte_sws_6515	`Rte_IWrite`, `Rte_IWriteRef`, `Rte_IInvalidate`, `Rte_IFeedback`	entity port	
`VariableAccess` in the role `dataSendPoint` rte_sws_6515	`Rte_Write`, `Rte_Invalidate`, `Rte_Feedback`	component port	
`VariableAccess` in the role `dataReceivePointByArgument` rte_sws_6515	`Rte_Read`	component port	
`VariableAccess` in the role `dataReceivePointByValue` rte_sws_6515	`Rte_DRead`	component port	
`VariableAccess` in the role `readLocalVariable` referring an `explicitInterRunnableVariable` rte_sws_6518	`Rte_IrvRead`	component internal	IRV
`VariableAccess` in the role `writtenLocalVariable` referring an `explicitInterRunnableVariable` rte_sws_6518	`Rte_IrvWrite`	component internal	IRV
`VariableAccess` in the role `readLocalVariable` referring an `implicitInterRunnableVariable` rte_sws_6519	`Rte_IrvIRead`	entity internal	IRV
`VariableAccess` in the role `writtenLocalVariable` referring an `implicitInterRunnableVariable` rte_sws_6519	`Rte_IrvIWrite`	entity internal	IRV
`PortPrototype` referring a `ParameterInterface` rte_sws_6515	`Rte_Prm`	component port	
`PortAPIOption` with attribute `indirectAPI` rte_sws_6515	`Rte_Ports`, `Rte_NPorts`, `Rte_Port`	component port	

Table 4.13: variant existence of RTE API

column	**description**
kind infix	The column kind infix defines infix strings to differentiate condition value macros belonging to variation points of different API sets
form	The column form specifies which names for the macro of the condition value are concatenated to ensure a unique name space of the macro.

form	**description**

component port	The related API is provide for the whole software component and belongs to a software components port
entity port	The related API is provide for a particular `RunnableEntity` and belongs to a software components port
component internal	The related API is provide for the whole software component and belongs to a software component internal functionality
entity internal	The related API is provide per `RunnableEntity` and belongs to a software component internal functionality

Table 4.14: Key to table 4.13

[rte_sws_6517]⌈ The RTE generator shall treat RTE API as variant RTE API only if all elements (e.g. `VariableAccess`) in the input configuration controlling the existence of the same RTE API are subject to variability. ⌋*(RTE00203)*

4.7.3.2.2 Conditional API options

Following variation points in the Meta Model do control the variant properties of RTE API or allocated Memory.

Variation Point Condition Value Macro	Subject to variability
`PortAPIOption` with attribute `portArgValue` not standardized	`PortDefinedArgumentValue` is passed to a `RunnableEntity`

Table 4.15: Conditional API options

4.7.3.2.3 Runnable Entity's and RTEEvents

Following variation points in the Meta Model do control the variant existence and activation of `RunnableEntity`s.

Variation Point Condition Value Macro	Subject to variability
`RunnableEntity` rte_sws_6530	Existence of the `RunnableEntity` prototype
`RTEEvent` not standardized	Activation of the `RunnableEntity`

Table 4.16: variation on Runnable Entity's and RTEEvents

4.7.3.2.4 Conditional Memory Allocation

Following variation points in the Meta Model do control the variant existence of RTE memory allocation for the software component instance.

Variation Point Condition Value Macro	Subject to variability
`implicitInterRunnableVariable` not standardized	variable definition implementing the `implicitInterRunnableVariable`
`explicitInterRunnableVariable` not standardized	variable definition implementing the `explicitInterRunnableVariable`
`arTypedPerInstanceMemory` not standardized	variable definition implementing the `arTypedPerInstanceMemory`
`PerInstanceMemory` not standardized	variable definition implementing the `PerInstanceMemory`
`perInstanceParameter` not standardized	constant definition implementing the `perInstanceParameter`
`sharedParameter` not standardized	variable definition implementing the `sharedParameter`
`InstantiationDataDefProps`, `SwDataDefProps` not standardized	Allocation of the memory objects described via `swAddrMethod`, accessibility for MCD systems described via `swCalibrationAccess`, `displayFormat`, `mcFunction`

Table 4.17: Conditional Memory Allocation

4.7.3.3 NvBlockComponent and its Internal Behavior

Variation Point Condition Value Macro	Subject to variability
`PortPrototype` of a `NvBlockSwComponentType` typed by `NvDataInterface` not standardized	Existence of the ability to access the memory objects of the `ramBlock`
`NvBlockDataMapping` of a `NvBlockDescriptor` not standardized	Existence of the ability to access the memory objects of the `ramBlock`

provide PortPrototype of a NvBlockSwComponentType typed by ClientServerInterface, RunnableEntity and referring OperationInvokedEvent not standardized	Existence of the *Block Management* port and the ability to access the *Block Management* API of the *NvRAM Manager*
require PortPrototype of a NvBlockSwComponentType typed by ClientServerInterface, RoleBasedPortAssignment and referring the PortPrototype not standardized	Existence of the *callback notification* port
NumericalValueSpecification or TextValueSpecification of the ramBlock or romBlocks initValue ValueSpecification (aggregated or referred one) not standardized	initialization values of the memory objects implementing the ramBlock or romBlock
InstantiationDataDefProps not standardized	Allocation of the memory objects implementing the ramBlock or romBlock described via swAddrMethod, accessibility for MCD systems described via swCalibrationAccess, displayFormat, mcFunction

Table 4.18: variation in NvBlockSwComponentTypes

4.7.3.4 Parameter Component

Variation Point Condition Value Macro	Subject to variability
PortPrototype of a ParameterSwComponentType not standardized	Existence of the memory objects / definitions related to the ParameterDataPrototypes in the PortInterface referred by the PortPrototype
NumericalValueSpecification or TextValueSpecification of the ParameterProvideComSpecs initValue ValueSpecification (aggregated or referred one) not standardized	initialization values of the memory objects / definitions related to the ParameterDataPrototypes

Table 4.19: variation in ParameterSwComponentTypes

4.7.3.5 Data Type

Following variation points in the Meta Model do control the variant generation of data types.

Variation Point Condition Value Macro	Subject to variability
ImplementationDataTypeElement	Existence of the structure or union element

rte_sws_6542	
arraySize rte_sws_6541	Number of elements in the array

Table 4.20: variation in ImplementationDataTypes

4.7.3.6 Basic Software Modules and its Internal Behavior

4.7.3.6.1 Basic Software Interfaces

Variation Point Condition Value Macro	Subject to variability
providedEntry not standardized	Existence of the provided BswModuleEntry
outgoingCallback not standardized	Existence of the expected BswModuleEntry
ModeDeclarationGroupPrototype in role providedModeGroup not standardized	Existence of the provided ModeDeclarationGroupPrototype
ModeDeclarationGroupPrototype in role requiredModeGroup not standardized	Existence of the required ModeDeclarationGroupPrototype
Trigger in role releasedTrigger not standardized	Existence of the released Trigger
Trigger in role requiredTrigger not standardized	Existence of the required Trigger

Table 4.21: variability affecting *Basic Software Interfaces*

4.7.4 Variability affecting the Basic Software Scheduler generation

4.7.4.1 Basic Software Scheduler API which is subject to variability

The VariationPoints listed in table 4.22 in the input configuration are controlling the variant existence of *Basic Software Scheduler* API.

Variation Point Condition Value Macro	Subject to variability	form	kind infix
ExclusiveArea rte_sws_6535	SchM_Enter, SchM_Exit	module internal	ExAr
managedModeGroup association to providedModeGroup ModeDeclarationGroupPrototype	SchM_Switch, SchM_SwitchAck	module external	MMod

rte_sws_6536			
accessedModeGroup association to providedModeGroup or requiredModeGroup ModeDeclarationGroupPrototype rte_sws_6536	SchM_Mode	module external	AMod
issuedTrigger association to releasedTrigger Trigger rte_sws_6536	SchM_Trigger	module external	Tr
BswInternalTriggeringPoint rte_sws_6536	SchM_ActMainFunction	entity internal	ITr

Table 4.22: variant existence of Basic Software Scheduler API

column	**description**
kind infix	The column kind infix defines infix strings to differentiate condition value macros belonging to variation points of different API sets
form	The column form specifies which names for the macro of the condition value are concatenated to ensure a unique name space of the macro.

form	**description**
module external	The related API is provide for the whole module and belongs to a module interface
module internal	The related API is provide for the whole module and belongs to a module internal functionality
entity internal	The related API is provide per ExecutableEntity and belongs to a module internal functionality

Table 4.23: Key to table 4.22

[rte_sws_6537]⌈ The RTE generator shall treat the existence of *Basic Software Scheduler* API as subject to variability only if all elements (e.g. managedModeGroup association) in the input configuration controlling the existence of the same *Basic Software Scheduler* API are subject to variability. ⌋*(RTE00229)*

4.7.4.2 Basic Software Entities

The VariationPoints listed in table 4.24 in the input configuration are controlling the variant existence of BswModuleEntitys and the variant activation of BswSchedulableEntitys.

Variation Point Condition Value Macro	Subject to variability
BswSchedulableEntity	Existence of the BswSchedulableEntity prototype

rte_sws_6532	
BswEvent not standardized	Activation of the BswSchedulableEntity

Table 4.24: variability affecting BswSchedulableEntitys

4.7.4.3 API behavior

The VariationPoints listed in table 4.25 in the input configuration are controlling the variant behavior of *Basic Software Scheduler* API.

Variation Point Condition Value Macro	Subject to variability
BswModeSenderPolicy not standardized	Queue length in the mode machine instance dependent from the attribute queueLength
BswModeReceiverPolicy not standardized	attribute supportsAsynchronousModeSwitch has to be considered according the bound variant

Table 4.25: variant existence of BswSchedulableEntity

4.8 Development errors

Errors which can occur at runtime in the RTE are classified as development errors. The RTE uses a BSW module report these types of errors to the DET [24] (Development Error Tracer).

4.8.1 DET Report Identifiers

[rte_sws_6630]⌈ The RTE shall report development errors to the DET and use its assigned module identifier (i.e. 2) to identify itself to the DET. ⌋*(BSW00337, BSW00338)*

[rte_sws_7676]⌈ Development errors shall be reported to the DET if and only if RteDevErrorDetect is enabled. ⌋*(BSW00337, BSW00338)*

[rte_sws_6631]⌈ The RTE shall use the OS Application Identifier as the Instance Id to enable the developer to identify in which runtime section of the RTE the error oc-

curs. This Instance ID is even unique across multi cores and so implicitly allows the development error to be traced to a specific core. ⌋*(BSW00337, BSW00338)*

[rte_sws_6632]⌈ The RTE shall use the Service Id as identified in the table 4.27. Each RTE API template, RTE callback template and RTE API will have an Identifier. This ID Service ID must be used when running code in the context of the respective RTE call. ⌋*(BSW00337, BSW00338)*

4.8.2 DET Error Identifiers

Only a limited set of development identifiers are currently recognized. Each of these need to be detected either at runtime or during initialization of the RTE. To report these errors extra development code must be generated by the RTE generator.

[rte_sws_6633]⌈ An RTE_E_DET_ILLEGAL_SIGNAL_ID (0x01) shall be reported at runtime by the RTE when it receives a COM callback for a signal name (e.g. `Rte_COMCbk_<sn>`, `Rte_COMCbkTAck_<sn>`) which was not expected within the context of the currently-selected postBuild variant. See section 5.9.2 for the list of possible COM callback template API. ⌋*(BSW00337, BSW00338)*

[rte_sws_6634]⌈ An RTE_E_DET_ILLEGAL_VARIANT_CRITERION_VALUE (0x02) shall be reported by the RTE when it determines that a value is assigned to a variant criterion which is not in the list of possible values for that criterion. This error shall be detected during the RTE initialization phase. ⌋*(BSW00337, BSW00338)*

[rte_sws_7684]⌈ An RTE_E_DET_ILLEGAL_VARIANT_CRITERION_VALUE (0x02) shall be reported by the *Basic Software Scheduler* when the `SchM_Init` API is called with a `NULL` parameter. ⌋*(BSW00337, BSW00338)*

[rte_sws_6635]⌈ An RTE_E_DET_ILLEGAL_INVOCATION (0x03) shall be reported by the RTE when it determines that an RTE API is called by a Runnable which should not call that RTE API. The RTE can identify the active Runnable when it dispatches the RTE Event and if it subsequently receives a call from that Runnable to an API that is not part of its contract then this particular error ID must me logged. ⌋*(BSW00337, BSW00338)*

[rte_sws_6637]⌈ An RTE_E_DET_WAIT_IN_EXCLUSIVE_AREA (0x04) shall be reported by the RTE when an application has called an Rte_Enter API and subsequently asks the RTE to enter a wait state. This is illegal because it would lock the ECU. ⌋*(BSW00337, BSW00338)*

[rte_sws_7675]⌈ An RTE_E_DET_ILLEGAL_NESTED_EXCLUSIVE_AREA (0x05) shall be reported by the RTE when an application violates rte_sws_ext_7172. ⌋*(BSW00337, BSW00338)*

[rte_sws_7685]⌈ An RTE_E_DET_SEG_FAULT (0x06) shall be reported by the RTE when the parameters of an RTE API call contain a direct or indirect reference to mem-

ory that is not accessible from the callers partition as defined in rte_sws_2752 and rte_sws_2753. ⌋*(BSW00337, BSW00338)*

[rte_sws_7682]⌈ If `RteDevErrorDetectUninit` is enabled, an RTE_E_DET_UNINIT (0x07) shall be reported by the RTE when one of the APIs :

- Specified in 5.6.
- `Rte_NvMNotifyInitBlock`.
- `Rte_PartitionTerminated`.
- `Rte_PartitionRestarting`.
- `Rte_RestartPartition`.

is called before `Rte_Start`, after `Rte_Stop` or After the partition to witch the API belongs is terminated. ⌋*(BSW00337, BSW00338)*

Note:

- In production mode, No checks are performed.
- In development mode, if an error is detected the API behaviour is undefined and it is left to the Rte implementer.

Rational: The introduction of this developpement check should not introduce big changes to production mode configuration.

[rte_sws_7683]⌈ If `RteDevErrorDetectUninit` is enabled, an RTE_E_DET_UNINIT (0x07) shall be reported by the *Basic Software Scheduler* / RTE when one of the APIs `SchM_Switch`, `SchM_Mode`, `SchM_SwitchAck`, `SchM_Trigger`, `SchM_ActMainFunction`, or `Rte_Start` is called before `SchM_Init`. ⌋*(BSW00337, BSW00338)*

4.8.3 DET Error Classification

The following abbreviations are used to identify the DET error in table 4.27.

Abbreviation	RTE DET Error
ISI	RTE_E_DET_ILLEGAL_SIGNAL_ID
IVCV	RTE_E_DET_ILLEGAL_VARIANT_CRITERION_VALUE
II	RTE_E_DET_ILLEGAL_INVOCATION
INEA	RTE_E_DET_ILLEGAL_NESTED_EXCLUSIVE_AREA
WIEA	RTE_E_DET_WAIT_IN_EXCLUSIVE_AREA
UNINIT	RTE_E_DET_UNINIT

Table 4.26: Abbreviations of RTE DET Errors to APIs

The following table 4.27 indicates which DET errors are relevant for the various RTE APIs, and the service ID associated with the RTE APIs (see rte_sws_6632):

API name	Service ID	ISI	IVCV	II	INEA	WIEA	UNINIT
Rte_Ports APIs	0x10						X
Rte_NPorts APIs	0x11						X
Rte_Port APIs	0x12						X
Rte_Send APIs	0x13						X
Rte_Write APIs	0x14						X
Rte_Switch APIs	0x15						X
Rte_Invalidate APIs	0x16						X
Rte_Feedback APIs	0x17					X	X
Rte_SwitchAck APIs	0x18					X	X
Rte_Read APIs	0x19						X
Rte_DRead APIs	0x1A						X
Rte_Receive APIs	0x1B					X	X
Rte_Call APIs	0x1C					X	X
Rte_Result APIs	0x1D					X	X
Rte_Pim APIs	0x1E						X
Rte_CData APIs	0x1F						X
Rte_Prm APIs	0x20						X
Rte_IRead APIs	0x21						X
Rte_IWrite APIs	0x22						X
Rte_IWriteRef APIs	0x23						X
Rte_IInvalidate APIs	0x24						X
Rte_IStatus APIs	0x25						X
Rte_IrvIRead APIs	0x26						X
Rte_IrvIWrite APIs	0x27						X
Rte_IrvRead APIs	0x28						X
Rte_IrvWrite APIs	0x29						X
Rte_Enter APIs	0x2A						X
Rte_Exit APIs	0x2B				X		X
Rte_Mode APIs	0x2C						X
Rte_Trigger APIs	0x2D						X
Rte_IrTrigger APIs	0x2E						X
Rte_IFeedback APIs	0x2F						X
Rte_IsUpdated APIs	0x30						X
trigger by TimingEvent	0x50			X			
trigger by BackgroundEvent	0x51			X			
trigger by SwcModeSwitchEvent	0x52			X			

trigger by AsynchronousServerCall-ReturnsEvent	0x53			X			
trigger by DataReceiveErrorEvent	0x54			X			
trigger by OperationInvokedEvent	0x55			X			
trigger by DataReceivedEvent	0x56			X			
trigger by DataSendCompletedEvent	0x57			X			
trigger by ExternalTriggerOccurredEvent	0x58			X			
trigger by InternalTriggerOccurredEvent	0x59			X			
trigger by DataWriteCompletedEvent	0x5A			X			
Rte_Start API	0x70						X
Rte_Stop API	0x71						
Rte_PartitionTerminated APIs	0x72						
Rte_PartitionRestarting APIs	0x73						
Rte_RestartPartition APIs	0x74						
Rte_COMCbkTAck_<sn> callbacks	0x90	X					
Rte_COMCbkTErr_<sn> callbacks	0x91	X					
Rte_COMCbkInv_<sn> callbacks	0x92	X					
Rte_COMCbkRxTOut_<sn> callbacks	0x93	X					
Rte_COMCbkTxTOut_<sn> callbacks	0x94	X					
Rte_COMCbk_<sg> callbacks	0x95	X					
Rte_COMCbkTAck_<sg> callbacks	0x96	X					
Rte_COMCbkTErr_<sg> callbacks	0x97	X					
Rte_COMCbkInv_<sg> callbacks	0x98	X					
Rte_COMCbkRxTOut_<sg> callbacks	0x99	X					
Rte_COMCbkTxTOut_<sg> callbacks	0x9A	X					
Rte_SetMirror callbacks	0x9B						
Rte_GetMirror callbacks	0x9C						
Rte_NvMNotifyJobFinished callbacks	0x9D						
Rte_NvMNotifyInitBlock callbacks	0x9E						X
SchM_Init API	0x00		X				
SchM_Deinit API	0x01						
SchM_GetVersionInfo API	0x02						
SchM_Enter APIs	0x03						X
SchM_Exit APIs	0x04				X		X
SchM_ActMainFunction APIs	0x05						X
SchM_Switch APIs	0x06						X
SchM_Mode APIs	0x07						X
SchM_SwitchAck APIs	0x08						X
SchM_Trigger APIs	0x09						X

Table 4.27: Applicability of RTE DET Errors to APIs

5 RTE Reference

"Everything should be as simple as possible, but no simpler."
– *Albert Einstein*

5.1 Scope

This chapter presents the RTE API from the perspective of AUTOSAR applications and basic software – the same API applies to all software whether they are AUTOSAR software-components or basic software.

Section 5.2 presents basic principles of the API including naming conventions and supported programming languages. Section 5.3 describes the header files used by the RTE and the files created by an RTE generator. The data types used by the API are described in Section 5.5 and Sections 5.6 and 5.7 provide a reference to the RTE API itself including the definition of runnable entities. Section 5.11 defines the events that can be monitored during VFB tracing.

5.1.1 Programming Languages

The RTE is required to support components written using the C and C++ programming languages [RTE00126] as well as legacy software modules [RTE_IN016]. The ability for multiple languages to use the same generated RTE is an important step in reducing the complexity of RTE generation and therefore the scope for errors.

[rte_sws_1167]⌈ The RTE shall be generated in C. ⌋*(RTE00126)*

[rte_sws_1168]⌈ All RTE code, whether generated or not, shall conform to the HIS subset of the MISRA C standard [25]. In technically reasonable, exceptional cases MISRA violations are permissible. Except for MISRA rule #11, such violations shall be clearly identified and documented. ⌋*(BSW007)*

Specified MISRA violations are defined in Appendix C.

In realistic use cases, the RTE will generate C identifiers (functions, types, variables, etc) whose name will be longer than the maximum size supported by the MISRA C standard (rule #11). Users should configure the RTE to indicate the maximum C identifiers' size supported by their tool chain to make sure that no issues will be caused by these MISRA violation.

[rte_sws_7300]⌈ If a RteToolChainSignificantCharacters limit has been configured, the RTE generator shall provide the list of C RTE identifiers whose name is not unique when only the first RteToolChainSignificantCharacters characters are considered. ⌋*(BSW007)*

The RTE API presented in Section 5.6 is described using C. The API is also directly accessible from an AUTOSAR software-component written using C++ provided all API functions and instances of data structures are imported with C linkage.

[rte_sws_1011]⌈ The RTE generator shall ensure that, for a component written in C++, all imported RTE symbols are declared using C linkage. ⌋*(RTE00138)*

For the RTE API for C and C++ components the import of symbols occurs within the application header file (Section 5.3.3).

5.1.2 Generator Principles

5.1.2.1 Operating Modes

An object-code component is compiled against an application header file that is created during the first “RTE Contract” phase of RTE generation. The object code is then linked against an RTE created during the second “RTE Generation” phase. To ensure that the object-code component and the RTE code are compatible the RTE generator supports *compatibility mode* that uses well-defined data structures and types for the component data structure. In addition, an RTE generator may support a *vendor* operating mode that removes compatibility between RTE generators from different vendors but permits implementation specific, and hence potentially more efficient, data structures and types.

[rte_sws_1195]⌈ All RTE operating modes shall be source-code compatible at the SW-C level. ⌋*(RTE00024, RTE00140)*

Requirement rte_sws_1195 ensures that a SW-C can be used in any operating mode as long as the source is available. The converse is not true – for example, an object-code SW-C compiled after the “RTE Contract” phase must be linked against an RTE created by an RTE generator operating in the same operating mode. If the vendor mode is used in the “RTE Contract” phase, an RTE generator from the same vendor (or one compatible to the vendor-mode features of the RTE generator used in the “RTE Contract” phase) has to be used for the “RTE Generation” phase.

5.1.2.1.1 Compatibility Mode

Compatibility mode is the default operating mode for an RTE generator and guarantees compatibility even between RTE generators from different vendors through the use of well-defined, “standardized”, data structures. The data structures that are used by the generated RTE in the compatibility mode are defined in Section 5.4.

Support for compatibility mode is required and therefore is guaranteed to be implemented by all RTE generators.

[rte_sws_1151]⌈ The *compatibility mode* shall be the default operating mode and shall be supported by all RTE generators, whether they are for the “RTE Contract” or “RTE Generation” phases. ⌋*(RTE00145)*

The compatibility mode uses custom (generated) functions with standardized names and data structures that are defined during the “RTE Contract” phase and used when compiling object-code components.

[rte_sws_1216]⌈ SW-Cs that are compiled against an “RTE Contract” phase application header file (i.e. object-code SW-Cs) generated in compatibility mode shall be compatible with an RTE that was generated in compatibility mode. ⌋*(RTE00145)*

The use of well-defined data structures imposes tight constraints on the RTE implementation and therefore restricts the freedom of RTE vendors to optimize the solution of object-code components but have the advantage that RTE generators from different vendors can be used to compile a binary-component and to generate the RTE.

Note that even when an RTE generator is operating in compatibility mode the data structures used for *source-code* components are not defined thus permitting vendor-specific optimizations to be applied.

5.1.2.1.2 Vendor Mode

Vendor mode is an optional operating mode where the data structures defined in the “RTE Contract” phase and used in the “RTE Generation” phase are implementation specific rather than “standardized”.

[rte_sws_1152]⌈ An RTE generator may optionally support *vendor mode*. ⌋*(RTE00083)*

The data structures defined and declared when an RTE generator operates in vendor mode are implementation specific and therefore *not* described in this document. This omission is deliberate and permits vendor-specific optimizations to be implemented for object-code components. It also means that RTE generators from different vendors are unlikely to be compatible when run in the vendor mode.

[rte_sws_1234]⌈ An AUTOSAR software-component shall be assumed to be operating in “compatibility” mode unless “vendor mode” is explicitly requested. ⌋*(RTE00145, RTE00146)*

The potential for more efficient implementations of object-code components offered by the vendor mode comes at the expense of requiring high cohesion between object-code components (compiled after the “RTE Contract” phase) and the generated RTE. However, this is not as restrictive as it may seem at first sight since the tight coupling is also reflected in many other aspects or the AUTOSAR methodology, not least of

which is the requirement that the same compiler (and compatible options) is used when compiling both the object-code component and the RTE.

5.1.2.2 Optimization Modes

The actual RTE code is generated – based on the input information – for each ECU individually. To allow optimization during the RTE generation one of the two general optimization directions can be specified: **MEMORY** consumption or execution **RUNTIME**.

[rte_sws_5053]⌈ The RTE Generator shall optimize the generated RTE code either for memory consumption or execution runtime depending on the provided input information `RteOptimizationMode`. ⌋*(RTE00023)*

5.1.2.3 Build support

The generated RTE code has to respect several rules in order to be integrated with other AUTOSAR software in the build process.

[rte_sws_5088]⌈ All memory[1] allocated by the RTE shall be wrapped in the segment declarations defined in the *Specification of Memory Mapping* [26] using `RTE` as the `<MSN>` (Module Short Name). ⌋*(RTE00148, RTE00169)*

Due to the structure of the AUTOSAR Meta Model the input configuration might contain several `DataPrototype`s which are resulting only in one memory object. In this case it is required to define rules which `SwAddrMethod` is used to allocate the memory and to decide about its initialization. Therefore precedence rules for `SwAddrMethod`s are defined by rte_sws_7590 and rte_sws_7591.

[rte_sws_7589]⌈ For `AutosarDataPrototype implementations` the `<SEGMENT>` infix for the `Memory Allocation Keyword` shall be set to the `shortName` of the preceding `SwAddrMethod` if there is one defined and if rte_sws_7592 is not applicable. ⌋*(RTE00148, RTE00169)*

[rte_sws_7047]⌈ If the `memoryAllocationKeywordPolicy` of the preceding `SwAddrMethod` is set to `AddrMethodShortName` the `<ALIGNMENT>` suffix with leading underscore of the `Memory Allocation Keyword` used by the `AutosarDataPrototype implementations` and `PerInstanceMemory implementations` shall be omitted. ⌋*(RTE00148, RTE00169)*

[rte_sws_7048]⌈ If the `memoryAllocationKeywordPolicy` of the preceding `SwAddrMethod` is set to `AddrMethodShortNameAndAlignment` the `<ALIGNMENT>` suffix with leading underscore of the `Memory Allocation Keyword` used by the `AutosarDataPrototype implementations` and `PerInstanceMemory implementations` shall be set to the resulting alignment as defined in

[1] *memory* refers to all elements in the generated RTE which will later occupy space in the ECU's memory and is directly associated with the RTE. This includes code, static data, parameters, etc.

rte_sws_7049, rte_sws_7050, rte_sws_7051, rte_sws_7052 and rte_sws_7053. ⌋*(RTE00148, RTE00169)*

[rte_sws_8303]⌈ The alignment of a `PerInstanceMemory` shall be set to `UNSPECIFIED`. ⌋*(RTE00013, RTE00077)*

[rte_sws_7049]⌈ The alignment defined by the preceding (see rte_sws_7196) `swAlignment` attribute of a `AutosarDataPrototype` precedes the alignment defined by the `ImplementationDataType` related to the `AutosarDataPrototype` as defined in rte_sws_7050, rte_sws_7051, rte_sws_7052 and rte_sws_7053. ⌋*(RTE00148, RTE00169)*

[rte_sws_7050]⌈ The alignment of a `AutosarDataPrototype` related to a `Primitive Implementation Data Type` or `Array Implementation Data Type` shall be set to the `baseTypeSize` of the referred `SwBaseType`. ⌋*(RTE00148, RTE00169)*

[rte_sws_7051]⌈ The alignment of a `AutosarDataPrototype` related to a `Structure Implementation Data Type` or `Union Implementation Data Type` shall be set to to biggest `baseTypeSize` of the `SwBaseTypes` used by the elements. ⌋*(RTE00148, RTE00169)*

Note: According rte_sws_7051 structures and unions are aligned according the size of the biggest primitive element in the structure.

[rte_sws_7052]⌈ The alignment of a `AutosarDataPrototype` related to a `Redefinition Implementation Data Type` shall be determined from the redefined `ImplementationDataType`. ⌋*(RTE00148, RTE00169)*

[rte_sws_7053]⌈ The alignment of a `AutosarDataPrototype` related to a `Pointer Implementation Data Type` shall be set to `UNSPECIFIED`. ⌋*(RTE00148, RTE00169)*

Note: If the RTE generator does not implement the memory objects related to `VariableDataPrototype`s and `ParameterDataPrototype`s for instance due to communication via IOC the assigned `SwAddrMethod`s might have no effect on the generated RTE code.

[rte_sws_7592]⌈ If the RTE Generator requires several non automatic memory objects per `AutosarDataPrototype`s (e.g. due to partitioning) the RTE Generator is permitted to select the `<SEGMENT>` infix for the auxiliary memory objects. ⌋*(RTE00148, RTE00169)*

Note: For definitions and declarations for memory objects allocated by the RTE and implementing `AutosarDataPrototype`s without an assigned `SwAddrMethod` the

RTE Generator is permitted to select the infix but still has to follow rte_sws_5088.

[rte_sws_7590]⌈ The SwAddrMethod of a AutosarDataPrototype in the pPort precedes the assigned SwAddrMethod(s) of the AutosarDataPrototype in the rPort.⌋*(RTE00148, RTE00169)*

[rte_sws_7591]⌈ The SwAddrMethod of the ramBlocks has always higher precedence as the assigned SwAddrMethods of the VariableDataPrototypes in the PortPrototypes.⌋*(RTE00148, RTE00169)*

[rte_sws_5089]⌈ The RTE Generator shall provide information on the used memory segments and their attributes from rte_sws_5088 in the generated *Basic Software Module Description*(see rte_sws_5086). The information shall be provided in the MemorySection elements of the *Basic Software Module Description* [9].⌋*(RTE00148, RTE00169, RTE00170)*

[rte_sws_5090]⌈ The RTE Generator shall provide information about the generated artifacts which are produced during the RTE generation, using the generated *Basic Software Module Description*(see rte_sws_5086). The information shall be provided in the Implementation::generatedArtifact elements of the *Basic Software Module Description* [9].⌋*()*

5.1.2.4 Debugging support

For the support of the AUTOSAR Debugging (see [27]) several requirements have to be respected.

[rte_sws_5094]⌈ Each variable that shall be accessible by AUTOSAR Debugging, shall be defined as global variable.⌋*()*

[rte_sws_5095]⌈ All type definitions of variables which shall be debugged, shall be accessible by the Rte types header file *Rte_Type.h*.⌋*()*

[rte_sws_5096]⌈ The declaration of variables in the header file shall be such, that it is possible to calculate the size of the variables by C-'sizeof()'.⌋*()*

[rte_sws_5097]⌈ Variables available for debugging shall be described in the respective *Basic Software Module Description* (see rte_sws_5086, [9]) using the elements BswDebugInfo.⌋*()*

[rte_sws_5098]⌈ If the state of a Runnable Entity is kept in a variable in the generated RTE, it shall be possible to debug the state of this Runnable Entity (rte_sws_2697).⌋*()*

[rte_sws_5105]⌈ If the Mode Machine Instance is kept in a variable in the generated RTE, it shall be possible to debug the state of this Mode Machine Instance,⌋*()*

5.1.2.5 Software Component Namespace

The concept of RTE requires that objects and definitions which are related to one software component are generated in a global name space. Nevertheless in this global name space labels have to be unique for instance to support a correct linkage by C Linker Locater. To ensure unique labels such objects and definitions related to a specific software component are typically prefixed or infixed with the component type symbol.
When `AtomicSoftwareComponentType`s of several vendors are integrated in the same ECU name clashes might occur if the identical component type name is accidentally used twice. To ease the dissolving of name clashes the RTE supports the superseding of the `AtomicSoftwareComponentType.shortName` with the `SymbolProps.symbol` attribute.

The resulting name related to an `AtomicSoftwareComponentType` is called `component type symbol` in this document.

[rte_sws_6714]⌈ The `component type symbol` shall be the value of the `SymbolProps.symbol` attribute of the `AtomicSoftwareComponentType` if the `symbol` attribute is defined. ⌋*()*

[rte_sws_6715]⌈ The `component type symbol` shall be the `shortName` of the `AtomicSoftwareComponentType` if no `symbol` attribute for this `AtomicSoftwareComponentType`is defined. ⌋*()*

Please note that the `component type symbol` is not applied for file names, e.g *Application Header File* or includes of Memory Mapping Header files. Its expected that a build environment can handle two equally named files.

5.1.3 Generator external configuration switches

There are use-cases where there is need to influence the behavior of the RTE Generator without changing the RTE Configuration description. In order to support such use-cases this section collects the *external configuration switches*.

Note: it is not specified how these switches shall be implemented in the actual RTE Generator implementation.

Unconnected R-Port check

[rte_sws_5099]⌈ The RTE Generator shall support the *external configuration switch* `strictUnconnectedRPortCheck` which, when enabled, forces the RTE Generator to consider unconnected R-Ports as an error. ⌋*(RTE00139)*

Missing input configuration check

[rte_sws_5148]⌈ The RTE Generator shall support the *external configuration switch* `strictConfigurationCheck` which, when enabled, forces the RTE Generator to consider missing input configuration information as an error. If the *external configura-*

tion switch `strictConfigurationCheck` is not provided the value shall be considered as *true*. ⌋*()*

For Details on the use-cases please refer to section 3.7.

Missing initialization values

[rte_sws_7680]⌈ The RTE Generator shall support the *external configuration switch* `strictInitialValuesCheck`. This swich, when enabled, forces the RTE Generator to check initial values against constraints defined in rte_sws_4525, rte_sws_7642 and rte_sws_7681. Not fullfilled constraints shall be considered as errors by the RTE Generator. ⌋*(RTE00108)*

5.2 API Principles

[rte_sws_1316]⌈ The RTE shall be configured and/or generated for each ECU. ⌋*(RTE00021)*

Part of the process is the customization (i.e. configuration or generation) of the RTE API for each AUTOSAR software-component on the ECU. The customization of the API implementation for each AUTOSAR software-component, whether by generation anew or configuration of library code, permits improved run-time efficiency and reduces memory overheads.

The design of the RTE API has been guided by the following core principles:

- The API should be orthogonal – there should be only one way of performing a task.
- **[rte_sws_1314]**⌈ The API shall be compiler independent. ⌋*(RTE00100)*
- **[rte_sws_3787]**⌈ The RTE implementation shall use the compiler abstraction. ⌋*(RTE00149)*
- **[rte_sws_1315]**⌈ The API shall support components where the source-code is available [RTE00024] and where only object-code is available [RTE00140]. ⌋*(RTE00024, RTE00140)*
- The API shall support the multiple instantiation of AUTOSAR software-components [RTE00011] that share code [RTE00012].

Two forms of the RTE API are available to software-components; direct and indirect. The direct API has been designed with regard to efficient invocation and includes an API mapping that can be used by an RTE generator to optimize a component's API, for example, to permit the direct invocation of the generated API functions or even eliding the generated RTE completely. The indirect API cannot be optimized using the API mapping but has the advantage that the handle used to access the API can be stored in memory and accessed, via an iterator, to apply the same API to multiple ports.

5.2.1 RTE Namespace

All RTE symbols (e.g. function names, global variables, etc.) visible within the global namespace are required to use the “Rte” prefix.

[rte_sws_1171]⌈ All externally visible symbols created by the RTE generator shall use the prefix `Rte_`.

This rule shall not be applied for the following symbols:

- type names representing AUTOSAR Data Types (specified in rte_sws_7104, rte_sws_7109, rte_sws_7110, rte_sws_7111, rte_sws_7114, rte_sws_7144, rte_sws_7148)
- enumeration literals of implementation data types (specified in rte_sws_3810)
- range limits of `ApplicationDataType`s (specified in rte_sws_5052)

This rule shall be applied for RTE internal types to avoid name clashes with other modules and SWCs. ⌋*(BSW00307, BSW00300, RTE00055)*

In order to maintain control over the RTE namespace the creation of symbols in the global namespace using the prefix `Rte_` is reserved for the RTE generator.

The generated RTE is required to work with components written in several source languages and therefore should not use language specific features, such as C++ namespaces, to ensure symbol name uniqueness.

5.2.2 Direct API

The direct invocation form is the form used to present the RTE API in Section 5.6. The RTE direct API mapping is designed to be optimizable so that the instance handle is elided (and therefore imposes zero run-time overhead) when the RTE generator can determine that exactly one instance of a component is mapped to an ECU.

All runnable entities for a AUTOSAR software-component type are passed the same instance handle type (as the first formal parameter) and can therefore use the same type definition from the component’s application header file.

The direct API can also be further optimized for source code components via the API mapping.

The direct API is typically implemented as macros that are modified by the RTE generator depending on configuration. This technique places certain restrictions on how the API can be used within a program, for example, it is not possible in C to take the address of a macro and therefore direct API functions cannot be placed within a function table or array. If it is required by the implementation of a software-component to derive a pointer to an object for the port API the `PortAPIOption enableTakeAddress` can be used. For instance in an implementation of an `AUTOSAR Service` this feature might be used to setup a constant function pointer table storing the configura-

tion of callback functions per ID. Additionally the indirect API provides support for API addresses and iteration over ports.

[rte_sws_7100]⌈ If a `PortPrototype` is referenced by `PortAPIOption` with `enableTakeAddress = TRUE` the RTE generator has to provide "C" functions and non function like macro for the API related to this port. ⌋*()*

The `PortAPIOption enableTakeAddress = TRUE` is not supported for software-components supporting multiple instantiation.

5.2.3 Indirect API

The indirect API is an optional form of API invocation that uses indirection through a port handle to invoke RTE API functions rather than direct invocation. This form is less efficient (the indirection cannot be optimized away) but supports a different programming style that may be more convenient. For example, when using the indirect API, an array of port handles of the same interface and provide/require direction is provided by RTE and the same RTE API can be invoked for multiple ports by iterating over the array.

Both direct and indirect forms of API call are equivalent and result in the same generated RTE function being invoked.

Whether the indirect API is generated or not can be specified for each software component and for each port prototype of the software component separately with the `indirectAPI` attribute.

The semantics of the port handle must be the same in both the “RTE Contract” and “RTE Generation” phases since the port handle accesses the standardized data structures of the RTE.

It is possible to mix the indirect and direct APIs within the same SW-C, if the indirect API is present for the SW-C.

The indirect API uses port handles during the invocation of RTE API calls. The type of the port handle is determined by the port interface that types the port which means that if a component declares multiple ports typed by the same port interface the port handle points to an array of port data structures and the same API invoked for each element.

The port handle type is defined in Section 5.4.2.5.

5.2.3.1 Accessing Port Handles

An AUTOSAR SW-C needs to obtain port handles using the instance handle before the indirect API can be used. The definition of the instance handle in Section 5.4.2 defines

the “Port API” section of the component data structure and these entries can be used to access the port handles in either object-code or source-code components.

The API `Rte_Ports` and `Rte_NPorts` provides port data handles of a given interface. Example 5.1 shows how the indirect API can be used to apply the same operation to multiple ports in a component within a loop.

Example 5.1

The port handle points to an array that can be used within a loop to apply the same operation to each port. The following example sends the same data to each receiver:

```
1 void TT1(Rte_Instance self)
2 {
3   Rte_PortHandle_interface1_P my_array;
4   my_array=Rte_Ports_interface1_P(self);
5   int s;
6   for(s = 0; s < Rte_NPorts_interface1_P(self); s++) {
7     my_array[s].Send_a(23);
8   }
9 }
```

Note that if `csInterface1` is a client/server interface with an operation `op`, the mechanism sketched in Example5.1 only works if `op` is invoked either by all clients synchronously or by all clients asynchronously, since the signature of `Rte_Call` and the existence of `Rte_Result` depend on the kind of invocation (see restriction rte_sws_3605.

5.2.4 `VariableAccess` in the `dataReadAccess` and `dataWriteAccess` roles

The RTE is required to support access to data with implicit semantics. The required semantics are subject to two constraints:

- For `VariableAccess` in the `dataReadAccess` role, the data accessed by a runnable entity must not change during the lifetime of the runnable entity.
- For `VariableAccess` in the `dataWriteAccess` role, the data written by a runnable entity is only visible to other runnable entities after the accessing runnable entity has terminated.

The generated RTE satisfies both requirements through data copies that are created when the RTE is generated based on the known task and runnable mapping.

Example 5.2

Consider a data element, `a`, of port `p` which is accessed using a `VariableAccess` in the `dataReadAccess` role by runnable `re1` and a `VariableAccess` in the

dataWriteAccess role by runnable re2. Furthermore, consider that re1 and re2 are mapped to different tasks and that execution of re1 can pre-empt re2.

In this example, the RTE will create two different copies to contain a to prevent updates from re2 'corrupting' the value access by re1 since the latter must remain unchanged during the lifetime of re1.

The RTE API includes three API calls to support VariableAccesses in the dataReadAccess and dataWriteAccess roles for a software-component; Rte_IRead (see Section 5.6.18), Rte_IWrite, and Rte_IWriteRef (see Section 5.6.19 and 5.6.20). The API calls Rte_IRead and Rte_IWrite access the data copies (for read and write access respectively). The API call Rte_IWriteRef returns a reference to the data copy, thus enabling the runnable to write the data directly. This is especially useful for Structure Implementation Data Type and Array Implementation Data Type. The use of an API call for reading and writing enables the definition to be changed based on the task and runnable mapping without affecting the software-component code.

Example 5.3

Consider a data element, a, of port p which is declared as being accessed using VariableAccesses in the dataWriteAccess role by runnables re1 and re2 within component c. The RTE API for component c will then contain four API functions to write the data element;

```
1 void Rte_IWrite_re1_p_a(Rte_Instance self, <type> val);
2 void Rte_IWrite_re2_p_a(Rte_Instance self, <type> val);
3 <type> Rte_IWriteRef_re1_p_a(Rte_Instance self);
4 <type> Rte_IWriteRef_re2_p_a(Rte_Instance self);
```

The API calls are used by re1 and re2 as required. The definitions of the API depend on where the data copies are defined. If both re1 and re2 are mapped to the same task then each can access the same copy. However, if re1 and re2 are mapped to different (pre-emptable) tasks then the RTE will ensure that each API access a different copy.

The Rte_IRead and Rte_IWrite use the "data handles" defined in the component data structure (see Section 5.4.2).

5.2.5 Per Instance Memory

The RTE is required to support Per Instance Memory [RTE00013].

The component's instance handle defines a particular instance of a component and is therefore used when accessing the *Per Instance Memory* using the Rte_Pim API.

The Rte_Pim API does not impose the RTE to apply a data consistency mechanism for the access to *Per Instance Memory*. An application is responsible for consistency of accessed data by itself. This design decision permits efficient (zero overhead) access when required. If a component possesses multiple runnable entities that require concurrent access to the same *Per Instance Memory*, an exclusive area can be used to ensure data consistency, either through explicit Rte_Enter and Rte_Exit API calls or by declaring that, implicitly, the runnable entities run inside an exclusive area.

Thus, the *Per Instance Memory* is exclusively used by a particular software-component instance and needs to be declared and allocated (statically).

In general there are two different kinds of *Per Instance Memory* available which are varying in the typing mechanisms. 'C' typed PerInstanceMemory is typed by the description of a 'C' typedef whereas arTypedPerInstanceMemory (*AUTOSAR Typed Per Instance Memory*) is typed by the means of an AutosarDataType. Nevertheless both kinds of *Per Instance Memory* are accessed via the Rte_Pim API.

[rte_sws_7161]⌈ The generated RTE shall declare arTypedPerInstanceMemory in accordance to the associated ImplementationDataType of a particular arTypedPerInstanceMemory. ⌋*(RTE00013, RTE00077)*

Note: The related *AUTOSAR data type* will generated in the RTE Types Header File (see chapter 5.3.6).

[rte_sws_2303]⌈ The generated RTE shall declare 'C' typed PerInstanceMemory in accordance to the attribute type of a particular PerInstanceMemory. ⌋*(RTE00013, RTE00077)*

In addition, the attribute type needs to be defined in the corresponding software-component header. Therefore, the attribute typeDefinition of the PerInstanceMemory contains its definition as plain text string. It is assumed that this text is valid 'C' syntax, because it will be included verbatim in the application header file.

[rte_sws_2304]⌈ The generated RTE shall define the type of a 'C' typed PerInstanceMemory by interpreting the text string of the attribute typeDefinition of a particular PerInstanceMemory as the 'C' definition. This type shall be named according to the attribute type of the PerInstanceMemory. ⌋*(RTE00013, RTE00077)*

[rte_sws_7133]⌈ The type of a 'C' typed PerInstanceMemory shall be defined in the *RTE Types Header File* as

```
typedef <typedefinition> Rte_PimType_<cts>_<type>;
```

where <typedefinition> is the content of the typeDefinition attribute of the PerInstanceMemory,
<type> is the type name defined in the type attribute of the the PerInstanceMemory and

<cts> the component type symbol of the AtomicSwComponentType to which the PerInstanceMemory belongs.. ⌋*(RTE00013, RTE00077)*

[rte_sws_3782]⌈ The type of a 'C' typed PerInstanceMemory shall be defined in the *Application Header File* as

```
typedef Rte_PimType_<cts>_<type> <type>;
```

where <cts> is the component type symbol of the AtomicSwComponentType to which the PerInstanceMemory belongs and
<type> is the type name defined in the type attribute of the PerInstanceMemory. ⌋*(RTE00013, RTE00077)*

[rte_sws_7134]⌈ The RTE generator shall generate type definitions for 'C' typed PerInstanceMemory (see rte_sws_7133 and rte_sws_3782) only once for all 'C' typed PerInstanceMemorys of same Software Component Type defining identical couples of type and typeDefinition attributes. ⌋*(RTE00013, RTE00165)*

Note: This shall support, that a Software Component Type can define several PerInstanceMemory's using the identical 'C' type.

[rte_sws_7135]⌈ The RTE generator shall reject configurations where 'C' typed PerInstanceMemorys with identical type attributes but different typeDefinition attributes in the same Software Component Type are defined. ⌋*(RTE00013, RTE00018)*

Note: This would lead to an compiler error due to incompatible redefinition of a 'C' type.

[rte_sws_2305]⌈ The generated RTE shall instantiate (or allocate) declared PerInstanceMemory. ⌋*(RTE00013, RTE00077)*

[rte_sws_7182]⌈ The generated RTE shall initialize declared PerInstanceMemory according the initValue attribute if

- an initValue is defined

  AND

- no SwAddrMethod is defined for PerInstanceMemory.

⌋*(RTE00013, RTE00077)*

[rte_sws_8304]⌈ Variables implementing PerInstanceMemory shall be initialized by RTE if

- an initValue is defined

  AND

- a SwAddrMethod is defined for PerInstanceMemory

  AND

- the RteInitializationStrategy for the sectionInitializationPolicy of the related SwAddrMethod is NOT configured to RTE_INITIALIZATION_STRATEGY_NONE.

⌋*(RTE00013, RTE00077)*

[rte_sws_7183]⌈ The generated RTE shall instantiate (or allocate) declared arTypedPerInstanceMemory. ⌋*(RTE00013, RTE00077)*

[rte_sws_7184]⌈ The generated RTE shall initialize declared arTypedPerInstanceMemory according the ValueSpecification of the VariableDataPrototype defining the arTypedPerInstanceMemory if the general initialization conditions in rte_sws_7046 are fulfilled. ⌋*(RTE00013, RTE00077)*

[rte_sws_5062]⌈ In case the PerInstanceMemory or arTypedPerInstanceMemory is used as a permanent ram mirror for the *NvRam manager* the name for the instantiated PerInstanceMemory or arTypedPerInstanceMemory shall be taken from the input information RteNvmRamBlockLocationSymbol. Otherwise the RTE generator is free to choose an arbitrary name. ⌋*(RTE00013, RTE00077)*

Note that, in cases where a PerInstanceMemory is not initialized due to rte_sws_7182 or rte_sws_7184, the memory allocated for a PerInstanceMemory is not initialized by the generated RTE, but by the corresponding software-component instances.

[rte_sws_7693]⌈ In case a ParameterDataPrototype in the role perInstanceParameter is used as a romBlock for the NVRam Manager, then the name for the instantiated ParameterDataPrototype shall be taken from the input information RteNvmRomBlockLocationSymbol. Otherwise the RTE generator is free to choose an arbitrary name. ⌋*(RTE00154)*

Example 5.4

This description of a software component

```
<AR-PACKAGE>
  <SHORT-NAME>SWC</SHORT-NAME>
  <ELEMENTS>
    <APPLICATION-SW-COMPONENT-TYPE>
      <SHORT-NAME>TheSwc</SHORT-NAME>
      <INTERNAL-BEHAVIORS>
        <SWC-INTERNAL-BEHAVIOR>
          <SHORT-NAME>TheSwcInternalBehavior</SHORT-NAME>
          <PER-INSTANCE-MEMORYS>
            <PER-INSTANCE-MEMORY>
              <SHORT-NAME>MyPIM</SHORT-NAME>
              <TYPE>MyMemType</TYPE>
              <TYPE-DEFINITION>struct {uint16 val1; uint8 * val2;}</
                  TYPE-DEFINITION>
            </PER-INSTANCE-MEMORY>
          </PER-INSTANCE-MEMORYS>
        </SWC-INTERNAL-BEHAVIOR>
      </INTERNAL-BEHAVIORS>
    </APPLICATION-SW-COMPONENT-TYPE>
  </ELEMENTS>
```

```
</AR-PACKAGE>
```

will e. g. result in the following code:

In the *RTE Types Header File*:

```
/* typedef to ensure unique typename */
/* according to the attributes */
/* 'type' and 'typeDefinition' */
typedef struct{
                 uint16 val1;
                  uint8 * val2;
} Rte_PimType_TheSwc_MyMemType;
```

In the respective *Application Header File*:

```
/* typedef visible within the scope */
/* of the component according to the attributes */
/* 'type' and 'typeDefinition' */
typedef Rte_PimType_TheSwc_MyMemType MyMemType;
```

In Rte.c:

```
/* declare and instantiate mem1 */
/* "mem1" name may be taken from RteNvmRamBlockLocationSymbol */
Rte_PimType_TheSwc_MyMemType mem1;
```

Note that the name used for the definition of the `PerInstanceMemory` may be used outside of the RTE. One use-case is to support the definition of the link between the NvRam Manager's permanent blocks and the software-components. The name in `RteNvmRamBlockLocationSymbol` is used to configure the location at which the NvRam Manager shall store and retrieve the permanent block content. For a detailed description please refer to the AUTOSAR Software Component Template [2].

5.2.6 API Mapping

The RTE API is implemented by macros and generated API functions that are created (or configured, depending on the implementation) by the RTE generator during the "RTE Generation" phase. Typically one customized macro or function is created for each "end" of a communication though the RTE generator may elide or combine custom functions to improve run-time efficiency or memory overheads.

[rte_sws_1274]⌈ The API mapping shall be implemented in the application header file. ⌋*(BSW00330, RTE00027, RTE00051, RTE00083, RTE00087)*

The RTE generator is required to provide a mapping from the RTE API name to the generated function [RTE00051]. The API mapping provides a level of indirection necessary to support binary components and multiple component instances. The indirection is necessary for two reasons. Firstly, some information may not be known when the

component is created, for example, the component's instance name, but are necessary to ensure that the names of the generated functions are unique. Secondly, the names of the generated API functions should be unique (so that the ECU image can link correctly) and the steps taken to ensure this may make the names not "user-friendly". Therefore, the primary rationale for the API mapping is to provide the required abstraction that means that a component does not need to concern itself with the preceding problems.

The requirements on the API mapping depend on the phase in which an RTE generator is operating. The requirements on the API mapping are only binding for RTE generators operating in compatibility mode.

5.2.6.1 "RTE Contract" Phase

Within the "RTE Contract" phase the API mapping is required to convert from the source API call (as defined in Section 5.6) to the runnable entity provided by a software-component or the implementation of the API function created by the RTE generator.

When compiled against a "RTE Contract" phase header file a software-component that can be multiple instantiated is required to use a general API mapping that uses the instance handle to access the function table defined in the component data structure.

[rte_sws_3706]⌈ If a software-component `supportsMultipleInstantiation`, the "RTE Contract" phase API mapping shall access the generated RTE functions using the instance handle to indirect through the generated function table in the component data structure. ⌋*(RTE00051)*

Example 5.5

For a require client-server port 'p1' with operation 'a' with a single argument, the general form of the API mapping would be:

```
1 #define Rte_Call_p1_a(s,v) ((s)->p1.Call_a(v))
```

Where `s` is the instance handle.

[rte_sws_6516]⌈ The RTE Generator shall wrap each API mapping and API function definition of a variant existent API according table 4.13 if the variability shall be implemented.

```
1 #if (<condition> [||<condition>])
2
3 <API Mapping>
4
5 #endif
```

where `condition` are the condition value macro(s) of the `VariationPoint`s relevant for the conditional existence of the RTE API (see table 4.13), `API Mapping` is

the code according an invariant API Mapping (see also rte_sws_1274, rte_sws_3707, rte_sws_3837, rte_sws_1156) ⌋*(RTE00201)*

Note: In case of explicit communication any existent access points in the meta model might result in the related API which results in a or condition for the pre processor.

Example 5.6

For a require client-server port 'p1' with operation 'a' with a single argument of the component 'c1' defining a `ServerCallPoint` which is subject of variability in runnable 'run1', the general form of the conditional API mapping would be:

```
1
2 #if (Rte_VPCon_c1_run1_p1_a)
3
4 #define Rte_Call_p1_a(s,v) ((s)->p1.Call_a(v))
5
6 #endif
7
```

[rte_sws_3707]⌈ If a software-component does not `supportsMultipleInstantiation`, the "RTE Contract" phase API mapping shall access the generated RTE functions directly. ⌋*(RTE00051)*

When accessed directly, the names of the generated functions are formed according to the following rule:

[rte_sws_3837]⌈ The function generated for API calls `Rte_<name>_<api_extension>` that are intended to be called by the software component shall be

`Rte_<name>_<cts>_<api_extension>`,

where `<name>` is the API root (e.g. `Receive`),
`<cts>` the `component type symbol` of the `AtomicSwComponentType`,
and `<api_extension>` is the extension of the API dependent on `<name>` (e.g. `<re>_<p>_<o>`). ⌋*(RTE00051)*

[rte_sws_1156]⌈ In compatibility mode, the following API calls shall be implemented as macros:

- `Rte_Pim`
- `Rte_IRead`
- `Rte_IWrite`
- `Rte_IWriteRef`
- `Rte_IStatus`
- `Rte_IrvIRead`

- `Rte_IrvIWrite`

The generated macros for these API calls shall map to the relevant fields of the component data structure. ⌋*(RTE00051)*

Note that the rule described in rte_sws_3837 does not apply for the life cycle APIs, nor for the callback APIs, nor for the APIs that are implemented as macros (see rte_sws_1156).

The functions generated that are the destination of the API mapping, which is created during the “RTE Contract” phase, are created by the RTE generator during the second “RTE Generation” phase.

[rte_sws_1153]⌈ The generated function (or runnable) shall take the same parameters, in the same order, as the API mapping. ⌋*(RTE00051)*

Example 5.7

For a require client-server port ‘p1’ with operation ‘a’ with a single argument for component type ‘c1’ for which multiple instantiation is forbidden, the following mapping would be generated:

```
#define Rte_Call_p1_a Rte_Call_c1_p1_a
```

5.2.6.2 “RTE Generation” Phase

There are no requirements on the *form* that the API mapping created during the “RTE Generation” phase should take. This is because the application header files defined during this phase are used by source-code components and therefore compatibility between the generated RTE and source-code components is automatic.

The RTE generator is required to produce the component data structure instances required by object-code components and multiple instantiated source-code components.

If multiple instantiation of a software-component is forbidden, then the API mapping specified for the “RTE Contract” phase (Section 5.2.6.1) defines the names of the generated functions. If multiple instantiation is possible, there are no corresponding requirements that define the name of the generated function since all accesses to the generated functions are performed via the component data structure which contains well-defined entries (Sections 5.4.2.5 and 5.4.2.5).

5.2.6.3 Function Elision

Using the “RTE Generation” phase API mapping, it is possible for the RTE generator to elide the use of generated RTE functions.

[rte_sws_1146]⌈ If the API mapping elides an RTE function the “RTE Generation” phase API mapping mechanism shall ensure that the invoking component still receives a “return value” so that no changes to the AUTOSAR software-component are necessary. ⌋*(RTE00051)*

In C, the elision of API calls can be achieved using a comma expression[2]

Example 5.8

As an example, consider the following component code:

```
1 Std_ReturnType s;
2 s = Rte_Send_p1_a(self,23);
```

Furthermore, assume that the communication attributes are specified such that the sender-receiver communication can be performed as a direct assignment and therefore no RTE API call needs to be generated. However, the component source cannot be modified and expects to receive an `Std_ReturnType` as the return. The “RTE Generation” phase API mapping could then be rewritten as:

```
1 #define Rte_Send_p1_a(s,a) (<var> = (a), RTE_E_OK)
```

Where `<var>` is the implementation dependent name for an RTE created cache between sender and receiver.

5.2.6.4 API Naming Conventions

An AUTOSAR software-component communicates with other components (including basic software) through ports and therefore the names that constitute the RTE API are formed from the combination of the API call’s functionality (e.g. Call, Send) that defines the API root name and the access point through which the API operates.

For any API that operates through a port, the API’s access point includes the port name.

A `SenderReceiverInterface` can support multiple data items and a `ClientServerInterface` can support multiple operations, any of which can be invoked through the requiring port by a client. The RTE API therefore needs a mechanism to indicate which data item/operation on the port to access and this is implemented by including the data item/operation name in the API’s access point.

[2]This is contrary to MISRA Rule 42 “*comma expression shall not be used except in the control expression of a for loop*”. However, a comma expression is valid, legal, C and the elision cannot be achieved without a comma expression and therefore the rule must be relaxed.

As described above, the RTE API mapping is responsible for mapping the RTE API name to the correct generated RTE function. The API mapping permits an RTE generator to include targeted optimization as well as removing the need to implement functions that act as routing functions from generic API calls to particular functions within the generated RTE.

For C and C++ the RTE API names introduce symbols into global scope and therefore the names are required to be prefixed with `Rte_` rte_sws_1171.

5.2.6.5 API Parameters

All API parameters fall into one of two classes; parameters that are strictly read-only ("In" parameters) and parameters whose value may be modified by the API function ("In/Out" and "Out" parameters).

The type of these parameters is taken from the data element prototype or operation prototype in the interface that characterizes the port for which the API is being generated.

- **"In" Parameters**

  [rte_sws_1017]⌈ All input parameters that are a `Primitive Implementation Data Type` shall be passed by value. ⌋*(RTE00059, RTE00061)*

  [rte_sws_1018]⌈ All input parameters that are of type `Structure Implementation Data Type` or `Union Implementation Data Type` shall be passed by reference. ⌋*(RTE00060, RTE00061)*

  [rte_sws_5107]⌈ All input parameters that are an `Array Implementation Data Type` shall be passed as an array expression (that is a pointer to the array base type). ⌋*(RTE00060, RTE00061)*

  [rte_sws_7661]⌈ All input parameters that are a data type of category `DATA_REFERENCE` shall be passed as a pointer to the data type specified by the `SwPointerTargetProps`. ⌋*(RTE00059, RTE00061)*

  [rte_sws_7086]⌈ All input parameters that are passed by reference (rte_sws_1018) or passed as an array expression (rte_sws_5107) shall be declared as pointer to const with the means of the `P2CONST` macro. ⌋*(RTE00060, BSW007)*

  Please note that the description of the `P2CONST` macro can be found in [28].

- **“Out” Parameters**

  [rte_sws_1019]⌈ All output parameters that of type `Primitive Implementation Data Type` shall be passed by reference. ⌋*(RTE00061)*

  [rte_sws_7082]⌈ All output parameters that are of type `Structure Implementation Data Type` or `Union Implementation Data Type` shall be passed by reference. ⌋*(RTE00060, RTE00061)*

  [rte_sws_5108]⌈ All output parameters that are an `Array Implementation Data Type` shall be passed as an array expression (that is a pointer to the array base type). ⌋*(RTE00060, RTE00061)*

  [rte_sws_7083]⌈ All output parameters that are of type `Pointer Implementation Data Type` shall be passed as a pointer to the `Pointer Implementation Data Type`. ⌋*(RTE00059, RTE00061)*

- **“In/Out” Parameters**

  [rte_sws_1020]⌈ All bi-directional parameters (i.e. both input and output) that are of type `Primitive Implementation Data Type` or `Structure Implementation Data Type` or `Union Implementation Data Type` shall be passed by reference. ⌋*(RTE00059, RTE00061)*

  [rte_sws_5109]⌈ All bi-directional parameters (i.e. both input and output) that are an `Array Implementation Data Type` shall be passed as an array expression (that is a pointer to the array base type). ⌋*(RTE00061)*

  [rte_sws_7084]⌈ All input, output and bi-directional parameters which related `DataPrototype` is typed or mapped to an `Redefinition Implementation Data Type` shall be treated according the kind of data type redefined by the `Redefinition Implementation Data Type`. The possible kinds of data types supported by RTE are listed in 5.3.4.2. ⌋*(RTE00059, RTE00060, RTE00061)*

In order to indicate the direction of the individual API parameters, the descriptions of the API signatures in this API reference chapter use the direction qualifiers ”IN”, ”OUT”, and ”INOUT”. These direction qualifiers are not part of the actual API prototypes. Especially, the user cannot expect that these direction qualifiers are available for the application.

Example 5.9

Consider an RTE API call taking an array as an “out” parameter for a singly instantiated SW-C. The signature of the API will be:

```
1 FUNC(Std_ReturnType, RTE_CODE) Rte_Write_<p>_<o>_(VAR(longArray_8,
2 AUTOMATIC) value)
```

And the function could be invoked as follows:

```
1 longArray_8 myArray;
2 Rte_Write_p1_d1(myArray);
```

5.2.6.6 Return Values

A subset of the RTE API's returning the values instead of using OUT Parameters. In the API section these API signatures defining a <return> value. In addition to the following rules some of the APIs might specify additionally const qualifiers.

[rte_sws_7069]⌈ The RTE Generator shall determine the <return> type according the applicable ImplementationDataType of the DataPrototype for which the API provides access. ⌋*(RTE00059)*

[rte_sws_8300]⌈ A pointer return value of an RTE API shall be declared as pointer to const with the means of the FUNC_P2CONST macro or P2CONST if the pointer is not used to modify the addressed object. ⌋*(RTE00059)*

Please note that the FUNC_P2CONST macro is applicable if the RTE API is implemented as an real function and the P2CONST might be used if the RTE API is implemented as a macro.

Requirement rte_sws_8300 applies for instance for the RTE APIs Rte_Prm, Rte_CData, Rte_IrvRead, Rte_IrvIRead in the cases where the API grants access to composite data (arrays, structures, unions).

Please note, that the the implementation of the C data types are specified in section 5.3.4 "RTE Types Header File".

[rte_sws_7070]⌈ If the DataPrototype is associated to a Primitive Implementation Data Type the RTE API shall return the value of the DataPrototype for which the API provides access. The type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Example 5.10

Consider an RTE API call return a primitive as defined in the example 5.3 for a singly instantiated SW-C. The signature of the API will be:

```
1 MyUint8 Rte_IRead_<re>_<p>_<o>(void);
```

Please note that the usage of Compiler Abstraction is not shown in the example.

[rte_sws_7071]⌈ If the DataPrototype is associated to a Structure Implementation Data Type or Union Implementation Data Type, the RTE API shall return a pointer to a variable holding the DataPrototype value provided by the

API. The type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Example 5.11

Consider an RTE API call return a structure as defined in the example 5.7 for a singly instantiated SW-C. The signature of the API will be:

```
1
2 FUNC_P2CONST(RecA, RTE_VAR_FAST_INIT, RTE_CODE)
3               Rte_IRead_<re>_<p>_<o>(void);
4
```

Please note that the usage of Compiler Abstraction assumes that the SwAddrMethod of the accessed VariableDataPrototype is named "'VAR_FAST_INIT"'. Further on the example does not respect the principles of API mapping.

[rte_sws_7072]⌈ If the DataPrototype is associated to an Array Implementation Data Type the RTE API shall return an array expression (that is a pointer to the array base type) pointing to variable holding the value of the DataPrototype for which the API provides access. If the leaf ImplementationDataTypeElement is typed by a SwBaseType the array type name shall be equal to the nativeDeclaration attribute of the SwBaseType. If the leaf ImplementationDataTypeElement is typed by an ImplementationDataType the type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Example 5.12

Consider an RTE API call return an array as defined in the example 5.5 for a singly instantiated SW-C. The signature of the API will be:

```
1 FUNC_P2CONST(unsigned char, RTE_VAR_POWER_ON_INIT, RTE_CODE)
2               Rte_IRead_<re>_<p>_<o>(void);
```

Please note that the usage of Compiler Abstraction assumes that the SwAddrMethod of the accessed VariableDataPrototype is named "'VAR_POWER_ON_INIT"'. Further on the example does not respect the principles of API mapping.

Example 5.13

Consider an RTE API call return an array as defined in the example 5.6 for a singly instantiated SW-C. The signature of the API will be:

```
1 FUNC_P2CONST(uint8, RTE_VAR_NO_INIT, RTE_CODE)
2               Rte_IRead_<re>_<p>_<o>(void);
```

Please note that the usage of Compiler Abstraction assumes that the SwAddrMethod of the accessed VariableDataPrototype is named "'VAR_NO_INIT"'. Further on the example does not respect the principles of API mapping.

[rte_sws_7073]⌈ If the DataPrototype is associated to a Pointer Implementation Data Type the RTE API shall return the value of the DataPrototype for which the API provides access. The type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)* Please not that in this case the value is a pointer.

[rte_sws_7074]⌈ If the DataPrototype is associated to a Redefinition Implementation Data Type the RTE Generator shall determine the API return value behaviour as described in rte_sws_7070, rte_sws_7071, rte_sws_7072, rte_sws_7073, rte_sws_7074 according the referenced ImplementationDataType. Nevertheless except for Array Implementation Data Type the type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Please note that Redefinition Implementation Data Type might redefine an other Redefinition Implementation Data Type again.

5.2.6.7 Return References

A subset of the RTE API's returning a reference to the memory location where the data can be accessed instead of using IN/OUT Parameters. In the API section these API signatures defining a <return reference> value.

[rte_sws_7076]⌈ The RTE Generator shall determine the <return reference> type according the applicable ImplementationDataType of the DataPrototype for which the API provides access. ⌋*(RTE00059)*

Please note, that the the implementation of the C data types are specified in section 5.3.4 "RTE Types Header File".

[rte_sws_7077]⌈ If the DataPrototype is associated to a Primitive Implementation Data Type the RTE API shall return a pointer to variable holding the data of the value of the DataPrototype for which the API provides access. The type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Example 5.14

Consider an RTE API call return a reference to a primitive as defined in the example 5.3 for a singly instantiated SW-C. The signature of the API will be:

```
1 MyUint8 * Rte_IWriteRef_<re>_<p>_<o>(void);
```

Please note that the usage of Compiler Abstraction is not shown in the example.

[rte_sws_7078]⌈ If the DataPrototype is associated to a Structure Implementation Data Type or Union Implementation Data Type the RTE API shall return a pointer to variable holding the value of the DataPrototype for which the API

provides access. The type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Example 5.15

Consider an RTE API call return a reference to a structure as defined in the example 5.7 for a singly instantiated SW-C. The signature of the API will be:

```
RecA * Rte_IWriteRef_<re>_<p>_<o>(void);
```

Please note that the usage of Compiler Abstraction is not shown in the example.

[rte_sws_7079]⌈ If the DataPrototype is associated to an Array Implementation Data Type the RTE API shall return an array expression (that is a pointer to the array base type) pointing to variable holding the value of the DataPrototype for which the API provides access. If the leaf ImplementationDataTypeElement is typed by a SwBaseType the array type name shall be equal to the nativeDeclaration attribute of the SwBaseType. If the leaf ImplementationDataTypeElement is typed by an ImplementationDataType the type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Example 5.16

Consider an RTE API call return a reference to an array as defined in the example 5.5 for a singly instantiated SW-C. The signature of the API will be:

```
unsigned char * Rte_IWriteRef_<re>_<p>_<o>(void);
```

Example 5.17

Consider an RTE API call return a reference to an array as defined in the example 5.6 for a singly instantiated SW-C. The signature of the API will be:

```
uint8 * Rte_IWriteRef_<re>_<p>_<o>(void);
```

Please note that the usage of Compiler Abstraction is not shown in the examples.

[rte_sws_7080]⌈ If the DataPrototype is associated to a Pointer Implementation Data Type the RTE API shall return a pointer pointing to variable holding the value of the DataPrototype for which the API provides access. The type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)* Please not that in this case the value is a pointer again.

[rte_sws_7081]⌈ If the DataPrototype is associated to a Redefinition Implementation Data Type the RTE Generator shall determine the API return value behaviour as described in rte_sws_7077, rte_sws_7078, rte_sws_7079, rte_sws_7080, rte_sws_7081 according the referenced ImplementationDataType. Nevertheless except for Array Implementation Data Type the type name shall be equal to the shortName of these ImplementationDataType. ⌋*(RTE00059)*

Please note that `Redefinition Implementation Data Type` might redefine an other `Redefinition Implementation Data Type` again.

5.2.6.8 Error Handling

In RTE, error and status information is defined with the data type `Std_ReturnType`, see Section 5.5.1.

It is possible to distinguish between infrastructure errors and application errors. Infrastructure errors are caused by a resource failure or an invalid input parameter. Infrastructure errors usually occur in the basic software or hardware along the communication path of a data element. Application errors are reported by a SW-C or by AUTOSAR services. RTE has the capability to treat application errors that are forwarded

- by return value in client server communication or
- by signal invalidation in sender receiver communication with data semantics.

Errors that are detected during an RTE API call are notified to the caller using the API's return value.

[rte_sws_1034]⌈ Error states (including 'no error') shall only be passed as return value of the RTE API to the AUTOSAR SW-C. ⌋*(RTE00094)*

Requirement rte_sws_1034 ensures that, irrespective of whether the API is blocking or non-blocking, the error is collected at the same time the data is made available to the caller thus ensuring that both items are accessed consistently.

Certain RTE API calls operate asynchronously from the underlying communication mechanism. In this case, the return value from the API indicates only errors detected during that API call. Errors detected after the API has terminated are returned using a different mechanism rte_sws_1111. RTE also provides an 'implicit' API for direct access to virtually shared memory. This API does not return any errors. The underlying communication is decoupled. Instead, an API is provided to pick up the current status of the corresponding data element.

5.2.6.9 Success Feedback

The RTE supports the notification of results of transmission attempts to an AUTOSAR software-component.

The `Rte_Feedback` API rte_sws_1083 or the `Rte_IFeedback` API rte_sws_7367 can be configured to return the transmission result as either a blocking or non-blocking API or via activation of a runnable entity.

5.2.7 Unconnected Ports

[rte_sws_1329]⌈ The RTE shall handle both require and provide ports that are not connected. ⌋*(RTE00139)*

The handling of require ports as an error is described in requirement rte_sws_5099.

The API calls for unconnected ports are specified to behave as if the port was connected but the remote communication point took no action.

Unconnected require ports are regarded by the RTE generator as an invalid configuration (see rte_sws_3019) if the strict handling has been enabled (see rte_sws_5099).

5.2.7.1 Data Elements

5.2.7.1.1 Explicit Communication

[rte_sws_1330]⌈ A `Rte_Read` API for an unconnected require port typed by a `SenderReceiverInterface` or `NvDataInterface` shall return the `RTE_E_UNCONNECTED` code and provide the `initValue` as if a sender was connected but did not transmit anything. ⌋*(RTE00139, RTE00200)*

[rte_sws_7663]⌈ A `Rte_DRead` API for an unconnected require port typed by a `SenderReceiverInterface` or `NvDataInterface` shall return the `initValue` as if a sender was connected but did not transmit anything. ⌋*(RTE00139, RTE00200)*

Requirements rte_sws_1330 and rte_sws_7663 apply to elements with "'data"' semantics and therefore "last is best"' semantics. This means that the initial value will be returned.

[rte_sws_1331]⌈ A blocking or non-blocking `Rte_Receive` API for an unconnected require port typed by a `SenderReceiverInterface` shall return `RTE_E_UNCONNECTED` immediately. ⌋*(RTE00139, RTE00200)*

The existence of blocking and non-blocking `Rte_Read`, `Rte_DRead` and `Rte_Receive` API calls is controlled by the presence of `VariableAccess`es in the `dataReceivePointByValue` or `dataReceivePointByArgument` role, `DataReceivedEvent`s and `WaitPoint`s within the SW-C description rte_sws_1288, rte_sws_1289 and rte_sws_1290.

[rte_sws_1344]⌈ A blocking or non-blocking `Rte_Feedback` API for a `VariableDataPrototype` of an unconnected provide port shall return `RTE_E_UNCONNECTED` immediately. ⌋*(RTE00139)*

The existence of blocking and non-blocking `Rte_Feedback` API is controlled by the presence of `VariableAccess`es in the `dataSendPoint` role, `DataSendCompletedEvent`s and `WaitPoint`s within the SW-C description for a `Variable-`

DataPrototype with acknowledgement enabled, see rte_sws_1283, rte_sws_1284, rte_sws_1285 and rte_sws_1286.

[rte_sws_1332]⌈ The Rte_Send or Rte_Write API for an unconnected provide port typed by a SenderReceiverInterface or NvDataInterface shall discard the input parameters and return RTE_E_OK. ⌋*(RTE00139)*

The existence of Rte_Send or Rte_Write is controlled by the presence of VariableAccesses in the dataSendPoint role within the SW/C description rte_sws_1280 and rte_sws_1281.

[rte_sws_3783]⌈ The Rte_Invalidate API for an unconnected provide port typed by a SenderReceiverInterface shall return RTE_E_OK. ⌋*(RTE00139)*

The existence of Rte_Invalidate is controlled by the presence of VariableAccesses in the dataSendPoint role within the SW/C description for a VariableDataPrototype which is marked as invalidatable by an associated InvalidationPolicy. The handleInvalid attribute of the InvalidationPolicy has to be set to keep or replace to enable the invalidation support for this dataElement (rte_sws_1282).

5.2.7.1.2 Implicit Communication

[rte_sws_7378]⌈ An Rte_IFeedback API for a VariableDataPrototype of an unconnected provide port shall return RTE_E_UNCONNECTED immediately. ⌋*(RTE00139, RTE00185)*

The existence of an Rte_IFeedback API is controlled by the presence of VariableAccesses in the dataWriteAccess role, and DataWriteCompletedEvents within the SWC description for a VariableDataPrototype with acknowledgement enabled, see rte_sws_7646, rte_sws_7647.

[rte_sws_1346]⌈ An Rte_IRead API for an unconnected require port typed by a SenderReceiverInterface or NvDataInterface shall return the initial value. ⌋*(RTE00139)*

The existence of Rte_IRead is controlled by the presence of a VariableAccess in the dataReadAccess role in the SW-C description rte_sws_1301.

[rte_sws_1347]⌈ An Rte_IWrite API for an unconnected provide port typed by a SenderReceiverInterface or NvDataInterface shall discard the written data. ⌋*(RTE00139)*

The existence of Rte_IWrite is controlled by the presence of a VariableAccess in the dataWriteAccess role in the SW-C description rte_sws_1302.

[rte_sws_3784]⌈ An Rte_IInvalidate API for an unconnected provide port typed by a SenderReceiverInterface shall perform no action. ⌋*(RTE00139)*

The existence of Rte_IInvalidate is controlled by the presence of a VariableAccess in the dataWriteAccess role in the SW-C description for a VariableDataPrototype which is marked as invalidatable by an associated InvalidationPolicy. The handleInvalid attribute of the InvalidationPolicy has to be set to keep or replace to enable the invalidation support for this dataElement (rte_sws_3801).

[rte_sws_3785]⌈ An Rte_IStatus API for an unconnected require port typed by a SenderReceiverInterface shall return RTE_E_UNCONNECTED. ⌋*(RTE00139, RTE00200)*

The existence of Rte_IStatus is controlled by the presence of a VariableAccess in the dataReadAccess role in the SW-C description for a VariableDataPrototype with data element outdated notification or data element invalidation rte_sws_2600.

5.2.7.2 Mode Switch Ports

For the mode user an unconnected mode switch port behaves as if it was connected to a mode manager that never sends a mode switch notification.

[rte_sws_2638]⌈ A Rte_Mode API for an unconnected mode switch port of a mode user shall return the initial state. ⌋*(RTE00139)*

[rte_sws_2639]⌈ Regarding the modes of an unconnected mode switch port of a mode user, the mode disabling dependencies on the initial mode shall be permanently active and the mode disabling dependencies on all other modes shall be inactive. ⌋*(RTE00139)*

[rte_sws_2640]⌈ Regarding the modes of an unconnected mode switch port of a mode user, RTE will only generate a SwcModeSwitchEvent for entering the initial mode which occurs directly after startup. ⌋*(RTE00139)*

[rte_sws_2641]⌈ The Rte_Switch API for an unconnected mode switch port of the mode manager shall discard the input parameters and return RTE_E_OK. ⌋*(RTE00139)*

[rte_sws_2642]⌈ A blocking or non blocking Rte_SwitchAck API for an unconnected mode switch port of the mode manager shall return RTE_E_UNCONNECTED immediately. ⌋*(RTE00139)*

5.2.7.3 Client-Server

[rte_sws_1333]⌈ The `Rte_Result` API for an unconnected asynchronous require port typed by a `ClientServerInterface` shall return `RTE_E_UNCONNECTED` immediately. ⌋*(RTE00139, RTE00200)*

[rte_sws_1334]⌈ The `Rte_Call` API for an unconnected require port typed by a `ClientServerInterface` shall return `RTE_E_UNCONNECTED` immediately. ⌋*(RTE00139, RTE00200)*

5.2.8 Non-identical port interfaces

Two ports are permitted to be connected provided that they are characterized by compatible, but not necessarily identical, interfaces. For the full definition of whether two interfaces are compatible, see the Software Component Template [2].

[rte_sws_1368]⌈ The RTE generator must report an error if two connected ports are connected by incompatible interfaces. ⌋*(RTE00137)*

A significant issue in determining whether two interfaces are compatible is that the interface characterizing the require port may be a strict subset of the interface characterizing the provide port. This means that there may be provided data elements or operations for which there is no corresponding element in the require port. This can be imagined as a multi-strand wire between the two ports (the assembly connector) where each strand represents the connection between two data elements or operations, and where some of the strands from the 'provide' end are not connected to anything at the 'require' end.

Define, for the purposes of this section, an "unconnected element" as a data element or operation that occurs in the provide interface, but for which no corresponding data element or operation occurs in a particular R-Port's interface.

[rte_sws_1369]⌈ For each data element or operation within the provide interface, every connected requirer with an "unconnected element" must be treated as if it were not connected. ⌋*(RTE00137)*

Note that requirement rte_sws_1369 means that in the case of a 1:n Sender-Receiver the `Rte_Write` call may transmit to some but not all receivers.

The extreme is if all connected requirers have an "unconnected element":

[rte_sws_1370]⌈ For a data element or operation in a provide interface which is an unconnected element in every connected R-Port, the generated `Rte_Send`, `Rte_Write`, `Rte_IWrite`, or `Rte_IWriteRef` APIs must act as if the port were unconnected. ⌋*(RTE00137)*

See Section 5.2.7 for the required behavior in this case.

5.3 RTE Modules

Figure 5.1 defines the relationship between header files and how those files are included by modules implementing AUTOSAR software-components and by general, non-component, code.

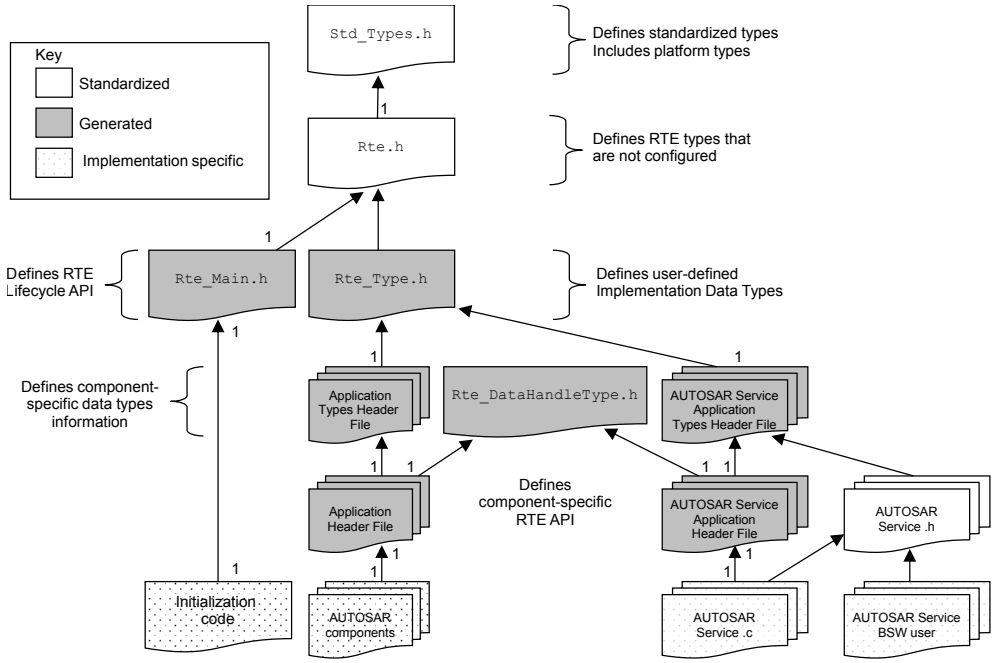


Figure 5.1: Relationships between RTE Header Files

The output of an RTE generator can consist of both generated code and configuration for “library” code that may be supplied as either object code or source code. Both configured and generated code reference standard definitions that are defined in the *RTE Header File*.

The relationship between the *RTE header file*, *Application Header File*s, the *Lifecycle Header File* and AUTOSAR software-components is illustrated in Figure 5.1.

In general a RTE can be partitioned in several files. The partitioning depends from the RTE vendors software design and generation strategy. Nevertheless it shall be possible to clearly identify code and header files which are part of the RTE module.

[rte_sws_7139]⌈ Every file of the RTE beside Rte.h and Rte.c shall be named with the prefix `Rte_`. ⌋*(BSW00300)*

5.3.1 RTE Header File

The RTE header file defines fixed elements of the RTE that do not need to be generated or configured for each ECU.

[rte_sws_1157]⌈ For C/C++ AUTOSAR software-components, the name of the RTE header file shall be `Rte.h`. ⌋*(BSW00300)*

Typically the contents of the RTE header file are fixed for any particular implementation and therefore it is not created by the RTE generator. However, customization for each generated RTE is not forbidden.

[rte_sws_1164]⌈ The RTE header file shall include the file `Std_Types.h`. ⌋*(RTE00149, RTE00150, BSW00353)*

The file `Std_Types.h` is the standard AUTOSAR file [29] that defines basic data types including platform specific definitions of unsigned and signed integers and provides access to the compiler abstraction.

The contents of the RTE header file are not restricted to standardized elements that are defined within this document – it can also contain definitions specific to a particular implementation.

5.3.2 Lifecycle Header File

The Lifecycle header file defines the two RTE Lifecycle API calls `Rte_Start` and `Rte_Stop` (see Section 5.8).

[rte_sws_1158]⌈ For C/C++ AUTOSAR software-components, the name of the lifecycle header file shall be `Rte_Main.h`. ⌋*(BSW00300)*

[rte_sws_1159]⌈ The lifecycle header file shall include the *RTE header file*. ⌋*(RTE00051)*

5.3.3 Application Header File

The application header file [RTE00087] is central to the definition of the RTE API. An application header file defines the RTE API and any associated data structures that are required by the SW-C to use the RTE implementation. But the application header file is not allowed to create objects in memory.

[rte_sws_1000]⌈ The RTE generator shall create an application header file for each software-component type (excluding `ParameterSwComponentType`s and `NvBlockSwComponentType`s) defined in the input. ⌋*(RTE00087, RTE00024, RTE00140)*

[rte_sws_3786]⌈ The application header file shall not contain code that creates objects in memory. ⌋*(RTE00087, BSW00308)*

RTE generation consists of two phases; an initial “RTE Contract” phase and a second “RTE Generation” phase (see Section 2.3). Object-code components are compiled after the first phase of RTE generation and therefore the application header file should conform to the form of definitions defined in Sections 5.4.1 and 5.5.2. In contrast, source-code components are compiled after the second phase of RTE generation and therefore the RTE generator produces an optimized application header file based on knowledge of component instantiation and deployment.

5.3.3.1 File Name

[rte_sws_1003]⌈ The name of the application header file shall be formed by prefixing the AUTOSAR software-component type name with `Rte_` and appending the result with `.h`. ⌋*(BSW00300)*

Example 5.18

The following declaration in the input XML:

```
<APPLICATION-SOFTWARE-COMPONENT-TYPE>
    <SHORT-NAME>Source</SHORT-NAME>
</APPLICATION-SOFTWARE-COMPONENT-TYPE>
```

should result in the application header file `Rte_Source.h` being generated.

The component type name is used rather than the component instance name for two reasons; firstly the same component code is used for all component instances and, secondly, the component instance name is an internal identifier, and should not appear outside of generated code.

5.3.3.2 Scope

RTE supports two approaches for the scope of the application header file, a SW-C based, and a runnable based approach.

1. Always, the application header file provides only the API that is specific for one atomic SW-C, see rte_sws_1004.
2. The scope of the application header file can be further reduced to one runnable by using the mechanism described in rte_sws_2751.

Many of the RTE APIs are specific to runnables. The restrictions for the usage of the generated APIs are defined in the 'Existence' parts of each API subsection in 5.6. To prevent run time errors by the misuse of APIs that are not supported for a runnable, it is recommended to use the runnable based approach of the application header file.

[rte_sws_1004]⌈ The application header file for a component shall contain only information relevant to that component. ⌋*(RTE00087, RTE00017, RTE00167)*

[rte_sws_2751]⌈ If the pre-compiler Symbol `RTE_RUNNABLEAPI_<rn>` is defined for a runnable with short name `<rn>` when the application header file is included, the application header file shall not declare APIs that are not valid to be used by the runnable *rn*. ⌋*(RTE00017)*

For example, to restrict the application header file of the SW-C `mySwc` to the API of the runnable `myRunnable`, the following sequence can be used:

```
#include <Rte_c1.h>

void
runnable_entry(Rte_Instance self)
{
  /* ... server code ... */
}
```

Figure 5.2: Skeleton server runnable entity

```
#define RTE_RUNNABLEAPI_myRunnable
#include <Rte_mySwc.h>

// runnable source code

```

Note that this mechanism does not support to restrict the application header file to the super set of two or more runnable APIs. In other words, runnables should be kept in separate source files, if the runnable based approach is used.

Requirements rte_sws_1004 and rte_sws_2751 mean that compile time checks ensure that a component (or runnable) that uses the application header file only accesses the generated data structures and functions to which it has been configured. Any other access, e.g. to fields not defined in the customized data structures or RTE API, will fail with a compiler error [RTE00017].

The definitions of the RTE API contained in the application header file can be optimized during the “RTE Generation” phase when the mapping of software-components to ECUs and the communication matrix is known. Consequently multiple application header files must not be included in the same source module to avoid conflicting definitions of the RTE API definitions that the files contains.

Figure 5.2 illustrates the code structure for the declaration of the entry point of a runnable entity that provides the implementation for a ServerPort in component `c1`. The RTE generator is responsible for creating the API and tasks used to execute the server and the symbol name of the entry point is extracted from the attribute symbol of the runnable entity. The example shows that the first parameter of the entry point function is the software-component’s instance handle rte_sws_1016.

Figure 5.2 includes the component-specific application header file `Rte_c1.h` created by the RTE generator. The RTE generator will also create the supporting data structures and the task body to which the runnable is mapped.

The RTE is also responsible for preventing conflicting concurrent accesses when the runnable entity implementing the server operation is triggered as a result of a request from a client received via the communication service or directly via inter-task communication.

5.3.3.3 File Contents

Multiple application header file must not be included in the same module (rte_sws_1004) and therefore the file contents should contain a mechanism to enforce this requirement.

[rte_sws_1006]⌈ An application header file shall include the following mechanism before any other definitions.

```
#ifdef RTE_APPLICATION_HEADER_FILE
#error Multiple application header files included.
#endif /* RTE_APPLICATION_HEADER_FILE */
#define RTE_APPLICATION_HEADER_FILE
```

⌋*(RTE00087)*

[rte_sws_7131]⌈ The application header file shall include the *Application Types Header File*. ⌋*(RTE00087)*

The name of the *Application Types Header File* is defined in Section 5.3.6.

[rte_sws_7924]⌈ The application header file shall include the *RTE Data Handle Types Header File* (see Section 5.3.5). ⌋*(RTE00087)*

[rte_sws_1005]⌈ The application header file shall be valid for both C and C++ source. ⌋*(RTE00126, RTE00138)*

Requirement rte_sws_1005 is met by ensuring that all definitions within the application header file are defined using C linkage if a C++ compiler is used.

[rte_sws_3709]⌈ All definitions within in the application header file shall be preceded by the following fragment;

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

[rte_sws_3710]⌈ All definitions within the application header file shall be suffixed by the following fragment;

```
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

5.3.3.3.1 Instance Handle

The RTE uses an instance handle to identify different instances of the same component type. The definition of the instance handle type rte_sws_1148 is unique to each component type and therefore should be included in the application header file.

[rte_sws_1007]⌈ The application header file shall define the type of the instance handle for the component. ⌋*(RTE00012)*

All runnable entities for a component are passed the same instance handle type (as the first formal parameter rte_sws_1016) and can therefore use the same type definition from the component's application header file.

5.3.3.3.2 Runnable Entity Prototype

The application header file also includes a prototype for each runnable entity entry point (rte_sws_1132) and the API mapping (rte_sws_1274).

5.3.3.3.3 Initial Values

[rte_sws_5078]⌈ The *Application Header File* shall define the init value of non-queued `VariableDataPrototype`s of sender receiver or non volatile data ports and typed by an `ImplementationDataType` or `ApplicationDataType` of category `VALUE`.

```
#define Rte_InitValue_<Port>_<DEPType> <initValue><suffix>
```

where `<Port>` is the `PortPrototype shortName`, `<DEPType>` is the `shortName` of the `VariableDataPrototype`, and `<initValue>` is the `initValue` specified in the `NonqueuedReceiverComSpec` respectively `NonqueuedSenderComSpec`. `<suffix>` shall be "U" for unsigned data types and empty for signed data types. ⌋*(RTE00068, RTE00087, RTE00108)*

Note that the `initValue` defined may be subject to change due to the fact that for COM configuration it may be possible to change this value during ECU Configuration or even post-build time.

5.3.3.3.4 PerInstanceMemory

The *Application Header File* shall type definitions for PerInstanceMemory's as defined in Chapter 5.2.5, rte_sws_7133.

5.3.3.3.5 RTE-Component Interface

The application header file defines the “interface” between a component and the RTE. The interface consists of the RTE API for the component and the prototypes for runnable entities. The definition of the RTE API requires that both relevant data structures and API calls are defined.

The data structures required to support the API are defined in the Application Header file (CDS) (see chapter 5.3.3), in the Application Types Header file (see chapter 5.3.6), in the RTE Types Header file (see chapter 5.3.1) and in the RTE Data Handle Types Header file (see chapter 5.3.5).

The data structure types are declared in the header files whereas the instances are defined in the generated RTE. The necessary data structures for object-code software-components are defined in chapter 5.5.2 and chapter 5.4.2.

The RTE generator is required rte_sws_1004 to limit the contents of the application header file to only that information that is relevant to that component type. This requirement includes the definition of the API mapping. The API mapping is described in chapter 5.2.6.

[rte_sws_1276]⌈ Only RTE API calls that are valid for the particular software-component type shall be defined within the component’s application header file. ⌋*(RTE00051, RTE00017, RTE00167)*

Requirement rte_sws_1276 ensures that attempts to invoke invalid API calls will be rejected as a compile-time error [RTE00017].

5.3.3.3.6 Application Errors

The concept of client server supports application specific error codes. Symbolic names for Application Errors are defined in the application header file to avoid conflicting definitions between several `AtomicSoftwareComponentType`s mapped one ECU. See rte_sws_2575 and rte_sws_2576.

5.3.4 RTE Types Header File

The *RTE Types Header File* includes the RTE specific type declarations derived from the `ImplementationDataType`s created from the definitions of AUTOSAR meta-model classes within the RTE generator’s input. The available meta-model classes are defined by the AUTOSAR software-component template and include classes for defining primitive values, structures, arrays and pointers.

The types declared in the *RTE Types Header File* intend to be used for the implementation of RTE internal data buffers as well as for RTE API.

[rte_sws_1160]⌈ The RTE generator shall create the *RTE Types Header File* including the type declarations corresponding to the `ImplementationDataType`s defined in the input configuration as well as the RTE implementation types. ⌋*()*

The RTE Data Types header file should be output for “RTE Contract” and “RTE Generation” phases.

5.3.4.1 File Contents

[rte_sws_2648]⌈ The *RTE Types Header File* shall include the type declarations for all the AUTOSAR Data Types according to rte_sws_7104, rte_sws_7110, rte_sws_6706, rte_sws_6707, rte_sws_6708 rte_sws_7111, rte_sws_7114, rte_sws_7144, rte_sws_7109 and rte_sws_7148 depending on the values of attributes `typeEmitter` and `nativeDeclaration` but irrespective of their use by the generated RTE. ⌋*()*

The attribute `typeEmitter` controls which part of the AUTOSAR toolchain is supposed to provide data type definitions. For legacy reasons the RTE generator is supposed to generate the corresponding data type if the `ImplementationDataType` defines no `typeEmitter`.

[rte_sws_6709]⌈ The RTE generator shall generate the corresponding data type definition if the value of attribute `typeEmitter` is NOT defined. ⌋*()*

[rte_sws_6710]⌈ The RTE generator shall generate the corresponding data type definition if the value of attribute `typeEmitter` is set to "`RTE`". ⌋*()*

[rte_sws_6711]⌈ The RTE generator shall reject configurations where the value of the attribute `typeEmitter` is set to "`RTE`" and the `ImplementationDataType` references a `SwBaseType` without defined `nativeDeclaration`. ⌋*()*

[rte_sws_6712]⌈ The RTE generator shall silently not generate the corresponding data type definition if the value of attribute `typeEmitter` is set to anything else but "`RTE`". ⌋*()*

This requirement ensures the availability of `ImplementationDataType`s for the internal use in AUTOSAR software components.

Nevertheless the *RTE Types Header File* does not contain any data type belonging to an `ImplementationDataType` where `typeEmitter` is set to anything else but "`RTE`" regardless if the `ImplementationDataType` references `SwBaseType`s and if this `SwBaseType`s define `nativeDeclaration`s.

The types header file may need types in terms of BSW types (from the file `Std_Types.h`) or from the implementation specific RTE header file to declare types.

However, since the RTE header file includes the file `Std_Types.h` already so only the RTE header file needs direct inclusion within the types header file.

[rte_sws_1163]⌈ The *RTE Types Header File* shall include the *RTE Header File*. ⌋*(BSW00353)*

5.3.4.2 Classification of Implementation Data Types

The type model `ImplementationDataType`s is able to express following kinds of data types:

- `Primitive Implementation Data Type`
- `Array Implementation Data Type`
- `Structure Implementation Data Type`
- `Union Implementation Data Type`
- `Redefinition Implementation Data Type`
- `Pointer Implementation Data Type`

A *Primitive Implementation Data Type* is classified that it directly refers by its `swDataDefProps` to a `SwBaseType` in the role `baseType`. The `category` attribute is set to `VALUE`.

An *Array Implementation Data Type* is classified that it defines `ImplementationDataTypeElement`s for each dimension of the array. The `swArraySize` specifies the number of array elements of the dimension. The `category` attribute *Array Implementation Data Type* is set to `ARRAY`.

A *Structure Implementation Data Type* is categorized that it has `ImplementationDataTypeElement`'s. The `category` attribute of the `ImplementationDataType` is set to `STRUCTURE`. Each `ImplementationDataTypeElement` it self can be one of the listed kinds again.

A *Union Implementation Data Type* is categorized that it has `ImplementationDataTypeElement`'s. The `category` attribute of the `ImplementationDataType` is set to `UNION`. Each `ImplementationDataTypeElement` it self can be one of the listed kinds again.

A *Redefinition Implementation Data Type* is classified that it refers to other `ImplementationDataType`s. The `category` attribute of the referring `ImplementationDataType` has to be set to `TYPE_REFERENCE`.

A *Pointer Implementation Data Type* is classified that its `swDataDefProps` has a `swPointerTargetProps` attribute. The `swDataDefProps` in the role `swPointerTargetProps` specifying the target to which the pointer refers. The `category` attribute of the `ImplementationDataType` has to be set to `DATA_REFERENCE`.

5.3.4.3 Primitive Implementation Data Type

The *RTE Types Header File* declares C types for all `Primitive Implementation Data Types` where the referred `BaseType` has a `nativeDeclaration` attribute.

[rte_sws_7104]⌈ For each `Primitive Implementation Data Type` with a `nativeDeclaration` attribute, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef <nativeDeclaration> <name>;`

where `<nativeDeclaration>` is the `nativeDeclaration` attribute of the referred `BaseType` and `<name>` is the `Implementation Data Type symbol` of the `Primitive Implementation Data Type`. ⌋*(RTE00055, RTE00166, RTE00168, BSW00353)*

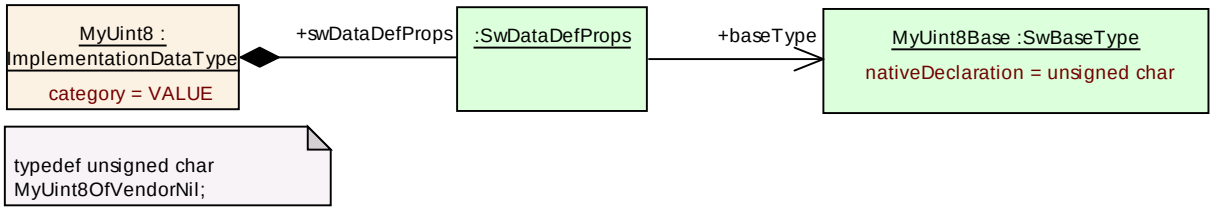


Figure 5.3: Primitive Implementation Data Type

Note: All `Primitive Implementation Data Types` where the referred `BaseType` has **no** `nativeDeclaration` attribute resulting not in a type declaration. This is intended to prevent the redeclaration of the predefined Standard Types and Platform Types.

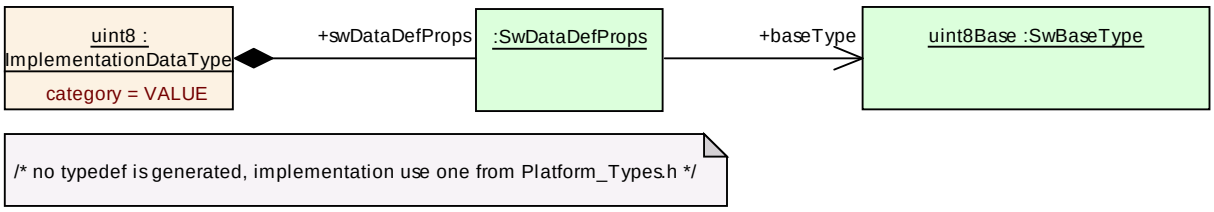


Figure 5.4: Primitive Implementation Data Type included from Platform_Types.h

[rte_sws_7105]⌈ If more than one `Primitive Implementation Data Type` with equal `shortName` and equal `nativeDeclaration` attribute of the referred `BaseType` are defined, the *RTE Types Header File* shall include only once the corresponding type declaration according to rte_sws_7104. ⌋*(RTE00165)*

Note: This avoids the redeclaration of C types due to the multiple descriptions of equivalent `Primitive Implementation Data Types` in the ECU extract.

5.3.4.4 Array Implementation Data Type

In addition to the primitive data-types defined in the previous section, it is also necessary for the RTE generator to declare composite data-types: arrays and records.

An array definition following information:

- the array type
- the number of dimensions
- the number of elements for each dimension.

[rte_sws_7110]⌈ For each `Array Implementation Data Type` which leaf `ImplementationDataTypeElement` is typed by a `BaseType`, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef <nativeDeclaration> <name>[<size 1>]{[<size 2>]...[<size n>]};`

where `<nativeDeclaration>` is the `nativeDeclaration` attribute of the referred `BaseType`,

`<name>` is the `Implementation Data Type symbol` of the `Array Implementation Data Type`,

`[<size x>]` is the `arraySize` of the Array's `ImplementationDataTypeElement`.

For each array dimension defined by one Array's `ImplementationDataTypeElement` one array dimension definition `[<size x>]` is defined. The array dimension definitions `[<size 1>], [<size 2>] ... [<size n>]` ordered from the root to the leaf `ImplementationDataTypeElement`.⌋*(RTE00055, RTE00164)*

[rte_sws_7111]⌈ For each `Array Implementation Data Type` which leaf `ImplementationDataTypeElement` is typed by an `ImplementationDataType`, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef <type> <name>[<size 1>]{[<size 2>]...[<size n>]};`

where `<type>` is the `shortName` of the referred `ImplementationDataType`,

`<name>` is the `Implementation Data Type symbol` of the `Array Implementation Data Type`,

`[<size x>]` is the `arraySize` of the Array's `ImplementationDataTypeElement`. For each array dimension defined by one Array's `ImplementationDataTypeElement` one array dimension definition `[<size x>]` is defined.

The array dimension definitions `[<size 1>], [<size 2>] ... [<size n>]` ordered from the root to the leaf `ImplementationDataTypeElement`.⌋*(RTE00055, RTE00164)*

[rte_sws_6706]⌈ For each `Array Implementation Data Type` which last `ImplementationDataTypeElement` is of category `STRUCTURE`, the RTE *Types Header File* shall include the corresponding type declaration as:

`typedef struct { <elements> } <name>[<size 1>]{[<size 2>]...[<size n>]};`

where `<elements>` is the record element specification and

`<name>` is the `Implementation Data Type symbol` of the `Array Implementation Data Type`.

For each record element defined by one `ImplementationDataTypeElement` one record element specification `<elements>` is defined. The record element specifications are ordered according the order of the related `ImplementationDataTypeElements` in the input configuration.
Sequent record elements are separated with a semicolon.

`[<size x>]` is the `arraySize` of the Array's `ImplementationDataTypeElement`. For each array dimension defined by one Array's `ImplementationDataTypeElement` one array dimension definition `[<size x>]` is defined.

The array dimension definitions `[<size 1>], [<size 2>] ... [<size n>]` ordered from the root to the last `ImplementationDataTypeElement` belonging to the array definition. ⌋*(RTE00055, RTE00164)*

The definition of the record element specification is defined in section 5.3.4.6.

[rte_sws_6707]⌈ For each `Array Implementation Data Type` which last `ImplementationDataTypeElement` is of category `UNION`, the RTE *Types Header File* shall include the corresponding type declaration as:

`typedef union { <elements> } <name>[<size 1>]{[<size 2>]...[<size n>]};`

where `<elements>` is the record element specification and

`<name>` is the `Implementation Data Type symbol` of the `Array Implementation Data Type`.

For each record element defined by one `ImplementationDataTypeElement` one record element specification `<elements>` is defined. The record element specifications are ordered according the order of the related `ImplementationDataTypeElements` in the input configuration.
Sequent record elements are separated with a semicolon.

`[<size x>]` is the `arraySize` of the Array's `ImplementationDataTypeElement`. For each array dimension defined by one Array's `ImplementationDataTypeElement` one array dimension definition `[<size x>]` is defined.

The array dimension definitions `[<size 1>], [<size 2>] ... [<size n>]` ordered from the root to the last `ImplementationDataTypeElement` belonging to the array definition. ⌋*(RTE00055, RTE00164)*

The definition of the record element specification is defined in section 5.3.4.6.

[rte_sws_6708]⌈ For each `Array Implementation Data Type` which last `ImplementationDataTypeElement` is of category `DATA_REFERENCE`, the RTE *Types Header File* shall include the corresponding type declaration as:

`typedef <tqlA> <addtqlA> <type> * <tqlB> <addtqlB> <name> [<size 1>]{[<size 2>]...[<size n>]};`

where `<name>` is the `Implementation Data Type symbol` of the `Array Implementation Data Type` and

`[<size x>]` is the `arraySize` of the Array's `ImplementationDataTypeElement`. For each array dimension defined by one Array's `ImplementationDataTypeElement` one array dimension definition `[<size x>]` is defined. The array dimension definitions `[<size 1>], [<size 2>] ... [<size n>]` ordered from the root to the last `ImplementationDataTypeElement` belonging to the array definition. ⌋*(RTE00055, RTE00164)*

For the definition of `<tqlA>` and `<tqlB>` see rte_sws_7149 and rte_sws_7166.

For the definition of `<addtqlA>` and `<addtqlB>` see rte_sws_7036 and rte_sws_7037.

[rte_sws_7112]⌈ If more than one `Array Implementation Data Type` with equal `shortName` of the `ImplementationDataType` and equal `nativeDeclaration` attribute of the referred `BaseType` are defined, the *RTE Types Header File* shall include only once the corresponding type declaration according to rte_sws_7110. ⌋*(RTE00165)*

[rte_sws_7113]⌈ If more than one `Array Implementation Data Type` with equal `shortName` of the `ImplementationDataType` and equal `shortName` of the referred `ImplementationDataType` are defined, the *RTE Types Header File* shall include only once the corresponding type declaration according to rte_sws_7111. ⌋*(RTE00165)*

Note: This avoids the redeclaration of C types due to the multiple descriptions of equivalent `Array Implementation Data Types` in the ECU extract.

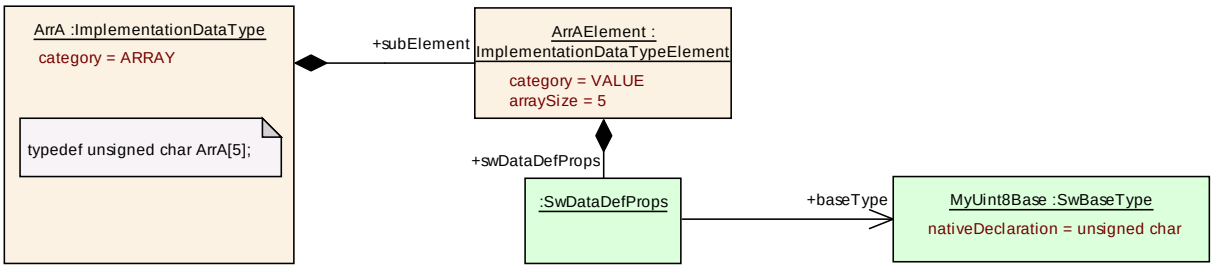


Figure 5.5: Example of a single dimension array typed by an `BaseType`

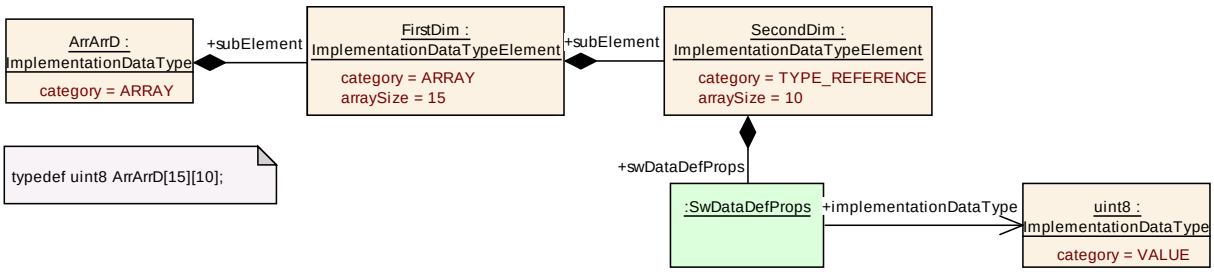


Figure 5.6: Example of a two dimension array typed by an `ImplementationDataType`

ANSI C does not allow a type declaration to have zero elements and therefore we require that the "number of elements" to be a positive integer.

[rte_sws_ext_1190] The `arraySize` defining number of elements in one dimension of an *Array Implementation Data Type* shall be an integer that is ≥ 1 for each dimension.

5.3.4.5 Structure Implementation Data Type and Union Implementation Data Type

[rte_sws_7114]⌈ For each `Structure Implementation Data Type`, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef struct { <elements> } <name>;`

where `<elements>` is the record element specification and `<name>` is the `Implementation Data Type symbol` of the `Structure Implementation Data Type`. For each record element defined by one `ImplementationDataTypeElement` one record element specification `<elements>` is defined. The record element specifications are ordered according the order of the related `ImplementationDataTypeElement`s in the input configuration. Sequent record elements are separated with a semicolon. ⌋*(RTE00055, RTE00164)*

5.3.4.6 Union Implementation Data Type

[rte_sws_7144]⌈ For each `Union Implementation Data Type`, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef union { <elements> } <name>;`

where `<elements>` is the union element specification and `<name>` is the `Implementation Data Type symbol` of the `Union Implementation Data Type`. For each union element defined by one `ImplementationDataTypeElement` one union element specification `<elements>` is defined. The union element specifications are ordered according the order of the related `ImplementationDataTypeElement`s in the input configuration. Sequent union elements are separated with a semicolon. ⌋*(RTE00055, RTE00164)*

[rte_sws_7115]⌈ Record and Union element specifications `<elements>` shall be generated as

`<nativeDeclaration> <name>;`

if the ImplementationDataTypeElement has the category attribute set to VALUE and if it refers to an BaseType. The meaning of the fields is identical to rte_sws_7104 ⌋*(RTE00055, RTE00164)*

[rte_sws_7116]⌈ Record and Union element specifications <elements> shall be generated as

<type> <name>;

if the ImplementationDataTypeElement has the category attribute set to TYPE_REFERENCE and if it refers to an ImplementationDataType. <type> is the Implementation Data Type symbol of the referred ImplementationDataType and <name> is the shortName of the ImplementationDataTypeElement. ⌋*(RTE00055, RTE00164)*

[rte_sws_7117]⌈ Record and Union element specifications <elements> shall be generated as

<nativeDeclaration> <name>[<size 1>]{[<size 2>]...[<size n>]};

if the ImplementationDataTypeElement has the category attribute set to ARRAY and which leaf ImplementationDataTypeElement has the category attribute set to VALUE and is typed by an BaseType. The meaning and order of the fields is identical to rte_sws_7110 ⌋*(RTE00055, RTE00164)*

[rte_sws_7118]⌈ Record and Union element specifications <elements> shall be generated as

<type> <name>[<size 1>]{[<size 2>]...[<size n>]};

if the ImplementationDataTypeElement has the category attribute set to ARRAY and which leaf ImplementationDataTypeElement has the category attribute set to TYPE_REFERENCE and is typed by an ImplementationDataType. The meaning and order of the fields is identical to rte_sws_7111 ⌋*(RTE00055, RTE00164)*

[rte_sws_7119]⌈ Record and Union element specifications <elements> shall be generated as

struct { <elements> } <name>;

if the ImplementationDataTypeElement has the category attribute set to STRUCTURE. The meaning and order of the fields is identical to rte_sws_7114 Sequent elements are separated with a semicolon. ⌋*(RTE00055, RTE00164)*

[rte_sws_7145]⌈ Record and Union element specifications <elements> shall be generated as

union { <elements> } <name>;

if the ImplementationDataTypeElement has the category attribute set to UNION. The meaning and order of the fields is identical to rte_sws_7144. Sequent elements are separated with a semicolon. ⌋*(RTE00055, RTE00164)*

[rte_sws_7146]⌈ Pointer element specifications <elements> shall be generated as

<tqlA> <addtqlA> <type> * <tqlB> <addtqlB> <name>;

if the ImplementationDataTypeElement has the category attribute set to DATA_REFERENCE where <name> is the shortName of the ImplementationDataTypeElement. ⌋*(RTE00055, RTE00164)*

For the definition of <tqlA> and <tqlB> see rte_sws_7149 and rte_sws_7166.

For the definition of <addtqlA> and <addtqlB> see rte_sws_7036 and rte_sws_7037.

For the definition of <type> see rte_sws_7162, rte_sws_7163.

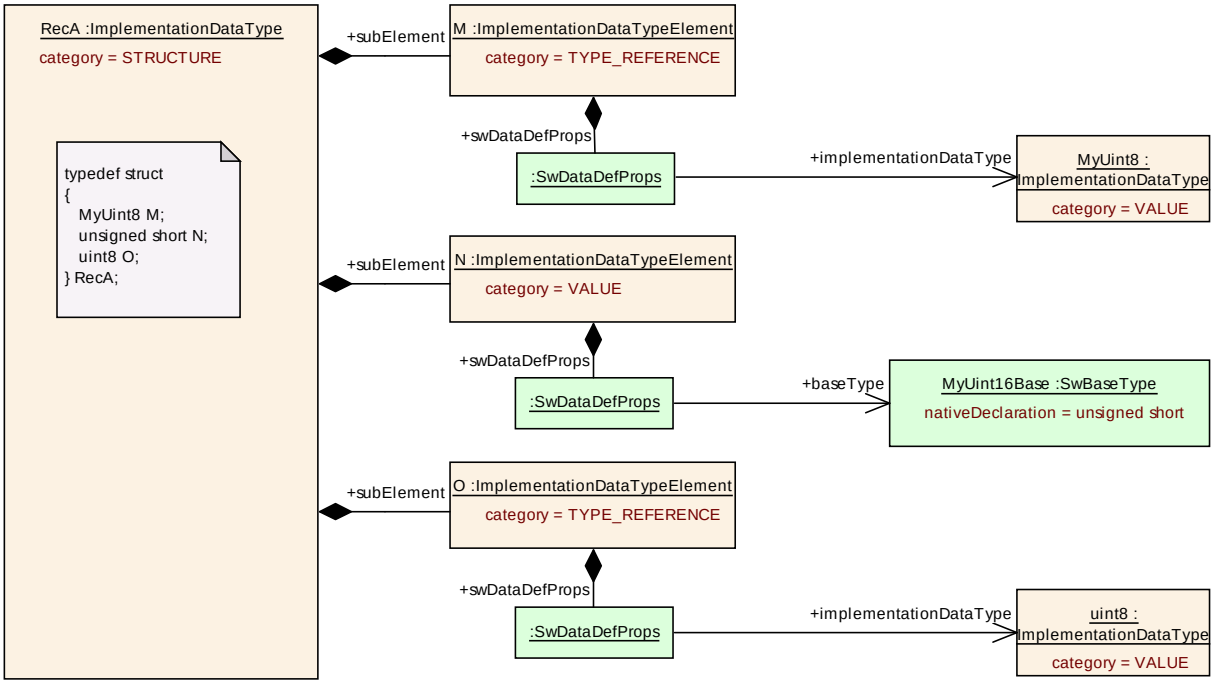


Figure 5.7: Example of a structure type

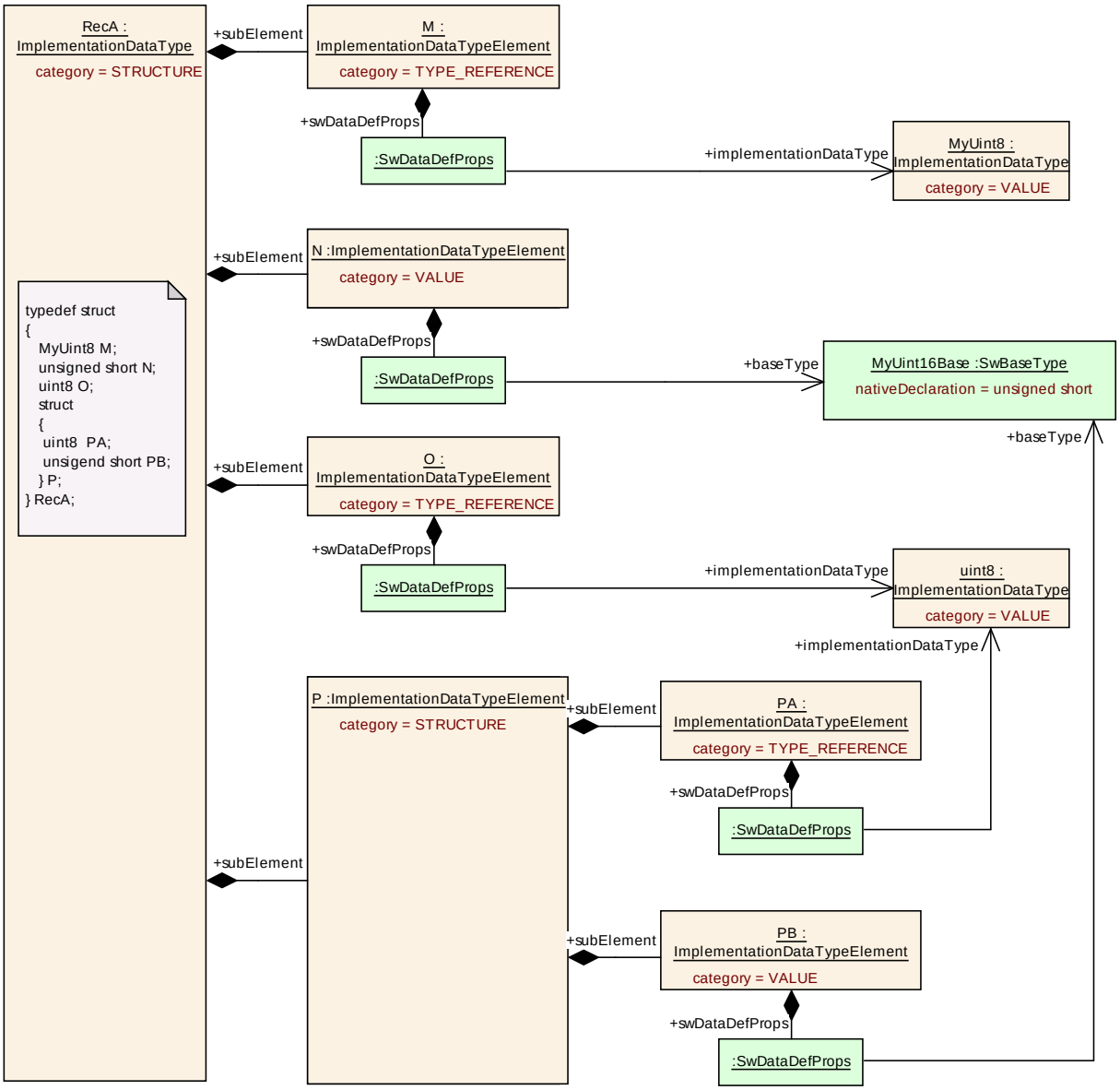


Figure 5.8: Example of a nested structure type

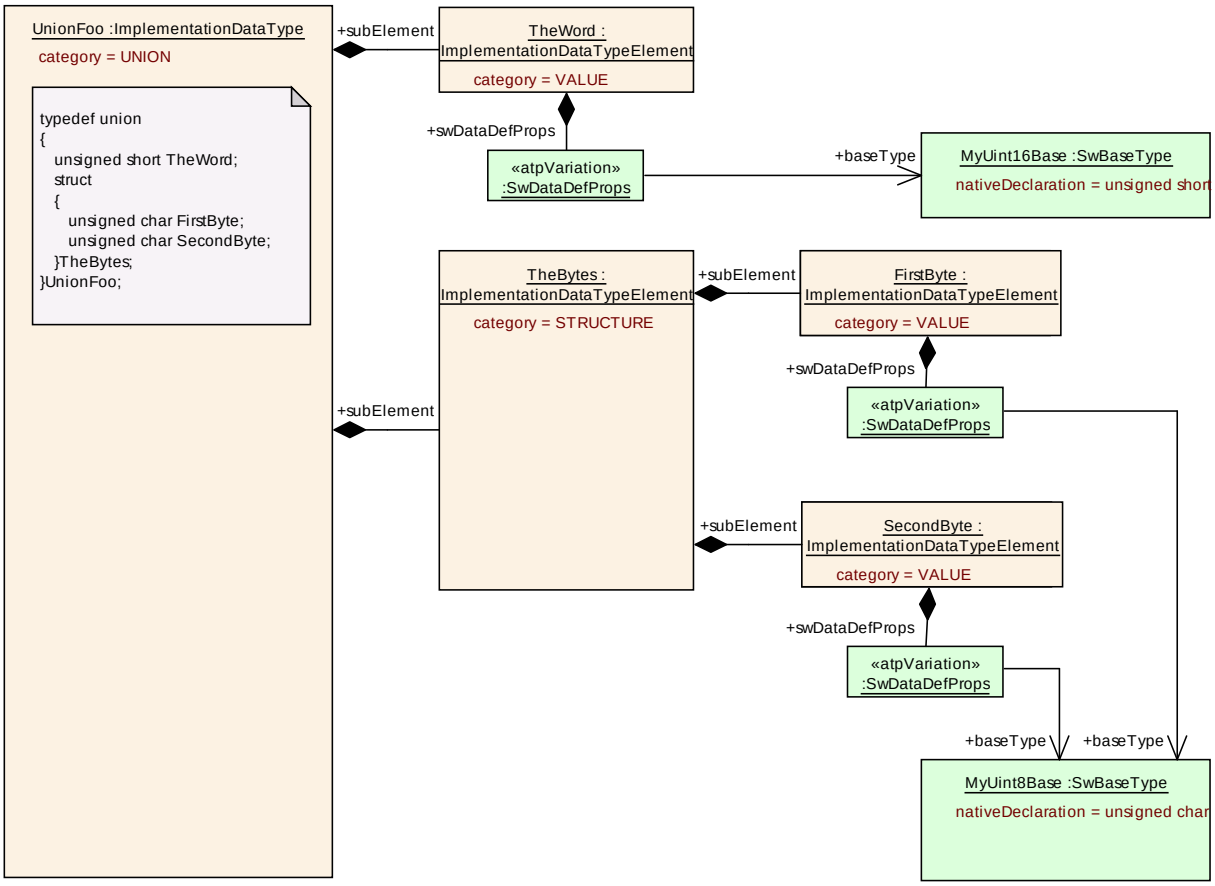


Figure 5.9: Example of a union type

[rte_sws_7107]⌈ If more than one `Structure Implementation Data Type` or `Union Implementation Data Type` with equal `shortName` of the `ImplementationDataType` are defined, the *RTE Types Header File* shall include only once the corresponding type declaration according to rte_sws_7114 or rte_sws_7144. ⌋*(RTE00165)*

Note: This avoids the redeclaration of C types due to the multiple descriptions of equivalent `Structure Implementation Data Types` and `Union Implementation Data Types` in the ECU extract.

ANSI C does not allow a `struct` to have zero elements and therefore we require that a record include at least one element.

[rte_sws_ext_1192] A structure shall include at least one element defined by a `ImplementationDataTypeElement`.

A union data type describes a kind of structural overlay. Defining only one sub element of a `union` ist therefore not reasonable and indicates an error.

[rte_sws_ext_7147] A *Union Implementation Data Type* shall include at least two elements defined by `ImplementationDataTypeElement`s.

5.3.4.7 Implementation Data Type redefinition

[rte_sws_7109]⌈ For each `Redefinition Implementation Data Type` which is typed by an `ImplementationDataType`, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef <type> <name>;`

where `<type>` is the `Implementation Data Type symbol` of the referred `ImplementationDataType` and `<name>` is the `Implementation Data Type symbol` of the `Primitive Implementation Data Type`. ⌋*(RTE00055, RTE00166)*

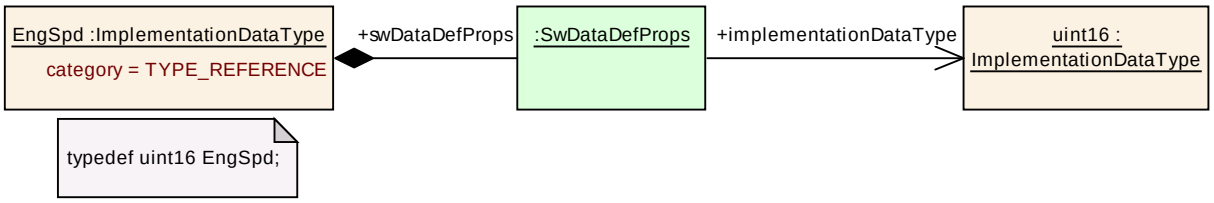


Figure 5.10: Example of an Implementation Data Type redefinition

[rte_sws_7167]⌈ If more than one `Redefinition Implementation Data Types` with equal `shortName`s which are referring to compatible `ImplementationDataType`s with identical `shortName`s are defined, the *RTE Types Header File* shall include only once the corresponding type declaration according to rte_sws_7109. ⌋*(RTE00165)*

Note: This avoids the redeclaration of C types due to the multiple descriptions of equivalent `Redefinition Implementation Data Type` in the ECU extract.

5.3.4.8 Pointer Implementation Data Type

[rte_sws_7148]⌈ For each `Pointer Implementation Data Type`, the *RTE Types Header File* shall include the corresponding type declaration as:

`typedef <tqlA> <addtqlA> <type> * <tqlB> <addtqlB> <name>;`

where `<name>` is the `Implementation Data Type symbol` of the `Pointer Implementation Data Type`. ⌋*(RTE00055, RTE00166)*

[rte_sws_7149]⌈ `<tqlA>` (type qualifier A) of a `Pointer Implementation Data Type` (rte_sws_7148) or *Pointer element specification*s (rte_sws_7146) shall be set to const if the `swImplPolicy` of the `swPointerTargetProps` is set to `const` and shall be omitted for all other values of `swImplPolicy`. ⌋*(RTE00055, RTE00166)*

[rte_sws_7166]⌈ `<tqlB>` (type qualifier B) of a `Pointer Implementation Data Type` (rte_sws_7148) or *Pointer element specification*s (rte_sws_7146) shall be set to const if the `swImplPolicy` of the `SwDataDefProps` of the `Implementation-`

DataType respectively ImplementationDataTypeElement is set to const and shall be omitted for all other values of swImplPolicy. ⌋*(RTE00055, RTE00166)*

[rte_sws_7036]⌈ <addtqlA> (additional type qualifier A) of a Pointer Implementation Data Type (rte_sws_7148) or *Pointer element specifications* (rte_sws_7146) shall be set to the content of the additionalNativeTypeQualifier attribute of the swPointerTargetProps if the attribute exists and shall be omitted if such additionalNativeTypeQualifier attribute dose not exist. ⌋*(RTE00055, RTE00166)*

[rte_sws_7037]⌈ <addtqlB> (additional type qualifier B) of a Pointer Implementation Data Type (rte_sws_7148) or *Pointer element specifications* (rte_sws_7146) shall be set to the content of the additionalNativeTypeQualifier attribute of the SwDataDefProps of the ImplementationDataType respectively ImplementationDataTypeElement and shall be omitted if such additionalNativeTypeQualifier attribute dose not exist. ⌋*(RTE00055, RTE00166)*

[rte_sws_7162]⌈ <type> shall be set to the nativeDeclaration attribute of the referred BaseType if the targetCategory of a Pointer Implementation Data Type (rte_sws_7148) or *Pointer element specifications* (rte_sws_7146) is set to VALUE ⌋*(RTE00055, RTE00166)*

[rte_sws_7163]⌈ <type> shall be the shortName of the referred ImplementationDataType if the targetCategory of a Pointer Implementation Data Type (rte_sws_7148) or *Pointer element specifications* (rte_sws_7146) is set to TYPE_REFERENCE ⌋*(RTE00055, RTE00166)*

[rte_sws_7169]⌈ If more than one Pointer Implementation Data Types with equal shortNames which are resulting in the same C pointer type declaration are defined, the *RTE Types Header File* shall include only once the corresponding type declaration according to rte_sws_7148. ⌋*(RTE00165)*

Note: This avoids the redeclaration of C types due to the multiple descriptions of equivalent Pointer Implementation Data Type in the ECU extract.

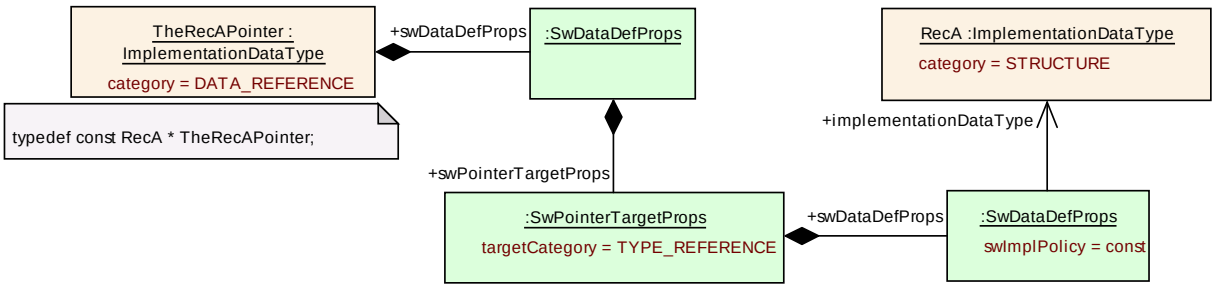


Figure 5.11: Example of a Pointer Implementation Data Type

5.3.4.9 ImplementationDataTypes with VariationPoints

[rte_sws_6539]⌈

The RTE Generator shall wrap each code related to ImplementationDataTypeElements which are subject to variability in Structure Implementation Data Type and Union Implementation Data Type (see 4.20 if the variability shall be implemented.

```
#if (<condition>)

<elements>

#endif
```

where <condition> are the *condition value macro*(s) of the VariationPoints according table 4.20 and

<elements> is the code according invariant ImplementationDataTypeElements (see also rte_sws_7115, rte_sws_7116, rte_sws_7117, rte_sws_7118, rte_sws_7119, rte_sws_7145, rte_sws_7146)

⌋*(RTE00201)*

[rte_sws_6540]⌈ The RTE Generator shall implement the <size x> of an Array Implementation Data Type for each arraySize which is subject to variability with the corresponding *attribute value macro* according table 4.20 if the variability shall be implemented. ⌋*(RTE00201)*

5.3.4.10 Naming of data types

[rte_sws_6716]⌈ The Implementation Data Type symbol shall be the shortName of the ImplementationDataType if no symbol attribute for this ImplementationDataTypeis defined. ⌋*(RTE00167)*

Example 5.19

The Primitive Implementation Data Type in example 5.3 results in the type definition:

```
/* RTE Types Header File */
typedef unsigned char MyUint8;
```

[rte_sws_6717]⌈ The Implementation Data Type symbol shall be the value of the SymbolProps.symbol attribute of the ImplementationDataType if the symbol attribute is defined. ⌋*(RTE00167)*

[rte_sws_6718]⌈ If the *RTE Types Header File* contains generated a C data type which Implementation Data Type symbol differs from the ImplementationDataType shortName the *Application Type Header File*s of each software component using the type shall contain a definition which redefines the Implementation Data Type symbol to the shortName of the ImplementationDataType. ⌋*(RTE00167)*

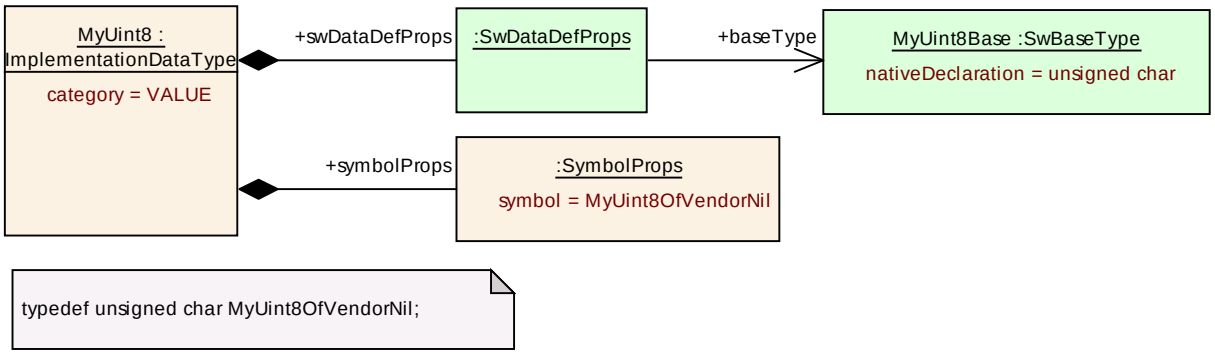


Figure 5.12: Primitive Implementation Data Type with `SymbolProps`

Example 5.20

If the input configuration contains a two `ImplementationDataType`s with same name but different definition the `SymbolProps` can be used to avoid the name clash. The `Primitive Implementation Data Type` in example 5.12 results in following definition:

```
1 /* RTE Types Header File */
2 typedef unsigned char MyUint8OfVendorNil;
```

The *Application Types Header File* an using component contain the remapping to the original name:

```
1 /* Application Types Header File */
2 define MyUint8 MyUint8OfVendorNil;
```

[rte_sws_6719]⌈ The RTE generator shall reject configurations where `ImplementationDataType`s result in the same `Implementation Data Type symbol` but whose definition would not resulting in the same type declaration. ⌋*(RTE00018)*

Note: This would result in compiler errors due to incompatible redefinition of C types.

[rte_sws_6724]⌈ The RTE generator shall reject configurations where the same software component uses `ImplementationDataType`s with equal `shortName`s which would result in the mapping to different `Implementation Data Type symbols`. ⌋*(RTE00018)*

Note: This would result in compiler errors due to incompatible redefinition of the mapping from `ImplementationDataType.shortName` to `Implementation Data Type symbol`

5.3.4.11 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1161]⌈ The name of the *RTE Types Header File* shall be `Rte_Type.h`. ⌋*(BSW00300)*

[rte_sws_1162]⌈ Within the *RTE Types Header File*, each data type shall be declared using `typedef`. ⌋*(RTE00126)*

A `typedef` is used when declaring a new data type instead of a `#define` even though C only provides weak type checking since other static analysis tools can then be used to overlay strong type checking onto the C before it is compiled and thus detect type errors before the module is even compiled.

5.3.5 RTE Data Handle Types Header File

The *RTE Data Handle Types Header File* contains the Data Handle type declarations necessary for the component data structures (see Section 5.4.2). The *RTE Data Handle Types Header File* code is not allowed to create objects in memory.

[rte_sws_7920]⌈ The RTE generator shall create the *RTE Data Handle Types Header File* including the type declarations of `Data Element without Status` (rte_sws_1363, rte_sws_1364, rte_sws_2607) and `Data Element with Status` (rte_sws_1365, rte_sws_1366, rte_sws_3734, rte_sws_2666, rte_sws_2589, rte_sws_2590). ⌋*()*

[rte_sws_7921]⌈ The *RTE Data Handle Types Header File* shall not contain code that creates object in memory. ⌋*(BSW00308)*

The *RTE Data Handle Types Header File* should be an output of the “RTE Contract” and “RTE Generation” phases.

5.3.5.1 File Name

[rte_sws_7922]⌈ The name of the *RTE Data Handle Types Header File* shall be `Rte_DataHandleType.h`. ⌋*(BSW00300)*

5.3.5.2 File Contents

The *RTE Data Handle Types Header File* contains the type declarations of `Data Element without Status` and `Data Element with Status` (see Section 5.4.2).

[rte_sws_7923]⌈ The *RTE Data Handle Types Header File* shall include the following mechanism to prevent multiple inclusions.

```
#ifndef RTE_DATA_HANDLE_TYPE_H
#define RTE_DATA_HANDLE_TYPE_H

/* File contents */

#endif /* RTE_DATA_HANDLE_TYPE_H */
```

⌋*(RTE00126)*

5.3.6 Application Types Header File

The *Application Types Header File* provides a component local name space for enumeration literals and range values. The *Application Types Header File* is not allowed to create objects in memory.

The *Application Types Header File* file should be identical output for "RTE Contract" and "RTE Generation" phases.

[rte_sws_7120]⌈ The RTE generator shall create an *Application Types Header File* for each software-component type (excluding *ParameterSwComponentType*s and *NvBlockSwComponentType*s) defined in the input. ⌋*(RTE00024, RTE00140, BSW00447)*

[rte_sws_7121]⌈ The *Application Types Header File* shall not contain code that creates objects in memory. ⌋*(BSW00308)*

5.3.6.1 File Name

[rte_sws_7122]⌈ The name of the *Application Types Header File* shall be formed by prefixing the AUTOSAR software-component type name with `Rte_` and appending the result with `_Type.h`. ⌋*(BSW00300, RTE00167)*

Example 5.21

The following declaration in the input XML:

```
<APPLICATION-SOFTWARE-COMPONENT-TYPE>
    <SHORT-NAME>Source</SHORT-NAME>
</APPLICATION-SOFTWARE-COMPONENT-TYPE>
```

should result in the *Application Types Header File* `Rte_Source_Type.h` being generated.

5.3.6.2 Scope

[rte_sws_7123]⌈ The *Application Types Header File* for a component shall contain only information relevant for that component. ⌋*(RTE00167, RTE00017)*

[rte_sws_7124]⌈ The *Application Types Header File* shall be valid for both C and C++ source. ⌋*(RTE00126, RTE00138)*

Requirement rte_sws_7124 is met by ensuring that all definitions within the *Application Types Header File* are defined using C linkage if a C++ compiler is used.

[rte_sws_7125]⌈ All definitions within in the *Application Types Header File* shall be preceded by the following fragment;

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

[rte_sws_7126]⌈ All definitions within the application types header file shall be suffixed by the following fragment;

```
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

[rte_sws_7678]⌈ The *Application Types Header File* shall be protected against multiple inclusions:

```
#ifndef RTE_<SWC>_TYPE_H
#define RTE_<SWC>_TYPE_H
...
/*
 * Contents of file
 */
...
#endif /* !RTE_<SWC>_TYPE_H */
```

Where <SWC> is the AUTOSAR software-component type name.[3] ⌋*(RTE00126)*

[3]No additional capitalization is applied to the names.

5.3.6.3 File Contents

In contrast to the *Application Header File* the *Application Types Header File* supports that multiple *Application Types Header File*'s are included in the same module. This is necessary if for instance a BSW module uses several AUTOSAR Services.

[rte_sws_7127]⌈ The *Application Types Header File* shall include the *RTE Types Header File*. ⌋*(RTE00087)*

The name of the *RTE Types Header File* is defined in Section 5.3.4.

5.3.6.4 RTE Modes

The *Application Types Header File* shall contain identifiers for the ModeDeclarations and type definitions for ModeDeclarationGroup's as defined in Chapter 5.5.3

5.3.6.5 Enumeration Data Types

The *Application Types Header File* shall contain the enumeration constants as defined in Chapter 5.5.4

5.3.6.6 Range Data Types

The *Application Types Header File* shall contain definitions of `Range` constants as defined in Chapter 5.5.5

5.3.6.7 Implementation Data Type symbols

The *Application Type Header File* may contain definitions to redefine the `Implementation Data Type symbol` to the `shortName` of the `ImplementationDataType` in order to provide the expected type name to the software component implementation. See section 5.3.4.10.

5.3.7 VFB Tracing Header File

The VFB Tracing Header File defines the configured VFB Trace events.

[rte_sws_1319]⌈ The VFB Tracing Header File shall be created by the RTE Generator during *RTE Generation Phase* only. ⌋*(RTE00045)*

The VFB Tracing Header file is included by the generated RTE and by the user in the module(s) that define the configured hook functions. The header file includes proto-

types for the configured functions to ensure consistency between the invocation by the RTE and the definition by the user.

5.3.7.1 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1250]⌈ The name of the VFB Tracing Header File shall be `Rte_Hook.h`. ⌋*(RTE00045)*

5.3.7.2 File Contents

[rte_sws_1251]⌈ The VFB Tracing header file shall include the *RTE Configuration Header File* (Section 5.3.8). ⌋*(RTE00045)*

[rte_sws_1357]⌈ The VFB Tracing header file shall include the *RTE Types Header file* (Section 5.3.4). ⌋*(RTE00003, RTE00004)*

[rte_sws_3607]⌈ The VFB Tracing header file shall include `Os.h`. ⌋*(RTE00005, RTE00008)*

[rte_sws_1320]⌈ The VFB Tracing header file shall contain the following code immediately after the include of the *RTE Configuration Header File*.

```
#ifndef RTE_VFB_TRACE
#define RTE_VFB_TRACE (FALSE)
#endif /* RTE_VFB_TRACE */
```

⌋*(RTE00008, RTE00005)*

Requirement rte_sws_1320 enables VFB tracing to be globally enabled/disabled within the RTE Configuration Header File and ensures that it defaults to 'disabled'.

[rte_sws_1236]⌈ For each trace event hook function defined in Section 5.11.4, the RTE generator shall define the following code sequence in the VFB Tracing header file:

```
#if defined(<trace event>) && (RTE_VFB_TRACE == FALSE)
#undef <trace event>
#endif
#if defined(<trace event>)
#undef <trace event>
extern void <trace event>(<params>);
#else
#define <trace event>(<params>) ((void)(0))
#endif /* <trace event> */
```

where `<trace event>` is the name of trace event hook function and `<params>` is the list of parameter names of the trace event hook function prototype as defined in Section 5.11.4. ⌋*(RTE00008)*

The code fragment within rte_sws_1236 benefits from a brief analysis of its structure. The first `#if` block ensures that an individually configured trace event in the RTE Configuration Header File rte_sws_1324 is disabled if tracing is globally disabled rte_sws_1323. The second `#if` block emits the prototype for the hook function only if enabled in the RTE Configuration file and thus ensures that only configured trace events are prototyped. The `#undef` is required to ensure that the trace event function is invoked as a function by the generated RTE. The `#else` block comes into effect if the trace event is disabled, either individually rte_sws_1325 or globally, and ensures that it has no run-time effect. Within the `#else` block the definition to `((void)(0))` enables the hook function to be used within the API Mapping in a comma-expression.

An individual trace event defined in Section 5.11.4 actually defines a class of hook functions. A member of the class is created for each RTE object created (e.g. for each API function, for each task) and therefore an individual trace event may give rise to many hook function definitions in the VFB Tracing header file.

Example 5.22

Consider an API call `Rte_Write_p1_a` for an instance of SW-C `c`. This will result in two trace event hook functions being created by the RTE generator:

```
Rte_WriteHook_c_p1_a_Start
```

and

```
Rte_WriteHook_c_p1_a_Return
```

5.3.8 RTE Configuration Header File

The *RTE Configuration Header File* contains user definitions that affect the behavior of the generated RTE.

The directory containing the required *RTE Configuration Header File* should be included in the compiler's include path when using the VFB tracing header file. The *RTE Configuration Header File* is generated by the RTE generator.

5.3.8.1 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1321]⌈ The name of the *RTE Configuration Header File* shall be `Rte_Cfg.h`. ⌋*(RTE00008, RTE00045)*

5.3.8.2 File Contents

[rte_sws_7641]⌈ The *RTE Configuration Header File* shall include the file `Std_Types.h`. ⌋*(RTE00149, RTE00150, BSW00353)*

5.3.8.2.1 VFB tracing configuration

[rte_sws_1322]⌈ The RTE generator shall globally enable VFB tracing when `RTE_VFB_TRACE` is defined in the *RTE Configuration Header File* as a vale which does not evaluate as `FALSE`. ⌋*(RTE00008, RTE00005)*

Note that, as observed in Section 5.11, VFB tracing enables debugging of software components, not the RTE itself.

[rte_sws_1323]⌈ The RTE generator shall globally disable VFB tracing when `RTE_VFB_TRACE` is defined in the RTE configuration header file as `FALSE`. ⌋*(RTE00008, RTE00005)*

As well as globally enabling or disabling VFB tracing, the RTE Configuration header file also configures those individual VFB tracing events that are *enabled*.

[rte_sws_1324]⌈ The RTE generator shall enable VFB tracing for a given hook function when there is a `#define` in the *RTE Configuration Header File* for the hook function name and tracing is globally enabled. ⌋*(RTE00008)*

Note that the particular value assigned by the `#define`, if any, is not significant.

[rte_sws_1325]⌈ The RTE generator shall disable VFB tracing for a given hook function when there is no `#define` in the *RTE Configuration Header File* for the hook function name even if tracing is globally enabled. ⌋*(RTE00008)*

Example 5.23

Consider the trace events from Example 5.22. The trace event for API start is enabled by the following definition;

```
#define Rte_WriteHook_i1_p1_a_Start
```

And the trace event for API termination is enabled by the following definition;

```
#define Rte_WriteHook_i1_p1_a_Return
```

5.3.8.2.2 Condition Value Macros

The *Condition Value Macros* are generated in the *PreBuild Data Set Contract Phase* and *PreBuild Data Set Generation Phase*. To do this a particular variant out of the

PreBuild Variability of the input configuration has to be chosen by the means described in by rte_sws_6500.

[rte_sws_6514]⌈ If evaluated BooleanValueVariationPoints or ConditionByFormulas are resulting to true the <value> for *Condition Value Macros* shall be coded as TRUE and if these are resulting to false the value shall be coded as FALSE. ⌋*(RTE00201, RTE00203)*

[rte_sws_6513]⌈ For each VariationPointProxy which bindingTime = PreCompileTime the *RTE Configuration Header File* shall contain a definition of a *Condition Value Macro* in the *RTE PreBuild Data Set Contract Phase* and *RTE PreBuild Data Set Generation Phase*

```
#define Rte_SysCon_<cts>_<name> <value>
```

Where <cts> is the component type symbol of the AtomicSwComponentType,

<name> is the shortName of the VariationPointProxy and

<value> is the evaluated value of the AttributeValueVariationPoint or ConditionByFormula. ⌋*(RTE00203, RTE00167)*

This requirements makes the SwSystemconst values available to resolve the PreBuild Variability in the software components via the Preprocessor. This might be used to

- read the actual value of the value assigned to a SwSystemconst
- read the setting of an attribute (e.g. array size) dependent from a SwSystemconst
- check the existence of a conditional existent object, e.g. an code fragment implementing a particular functionality

[rte_sws_3854]⌈ For each VariationPointProxy which bindingTime = PreCompileTime the *RTE Application Header File* shall contain a definition

```
#define Rte_SysCon_<name> Rte_SysCon_<cts>_<name>
```

where <cts> is the component type symbol of the AtomicSwComponentType and

<name> is the shortName of the VariationPointProxy. ⌋*(RTE00203, RTE00167)*

[rte_sws_6515]⌈ For each RTE API which is subject to variability and following the form *component port* or *entity port* in table 4.13 the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define Rte_VPCon_<cts>_<re>[_<resl>]_<p>_<o>[_<psl>] <value>
```

where <cts> is the component type symbol of the AtomicSwComponentType,

<re> is the short name of the RunnableEntity,

<resl> is the shortLabel of the RunnableEntity's VariationPoint containing the reference element (e.g. a VariableAccess) to the PortInterface element,

<p> is the name of the PortPrototype,

<o> is the short name of the PortInterface element and

<psl> is the shortLabel of the PortPrototype's VariationPoint which is referred by the VariableAccess

If there is no VariationPoint at the RunnableEntity owning the VariableAccess the <resl> with leading underscore is omitted ([_<resl>]).

If there is no VariationPoint at the PortPrototype referred by the VariableAccess the <psl> with leading underscore is omitted ([_<psl>]).

<value> is the evaluated value of the ConditionByFormula of the VariationPoint vary the existence of the RTE API in table 4.13. ⌋*(RTE00201, RTE00167)*

[rte_sws_6518]⌈ For each RTE API which is subject to variability and following the form *component internal* in table 4.13 the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define Rte_VPCon_<cts>_<ki>_<name>_<sl> <value>
```

where <cts> is the component type symbol of the AtomicSwComponentType,

<ki> is the *kind infix* according table 4.13,

<name> is the short name of the element which is subject to variability in table 4.13 and is defining the API name infix,

<sl> is the shortLabel of the elements' VariationPoint defining the API name infix.

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the RTE API in table 4.13. ⌋*(RTE00201, RTE00167)*

[rte_sws_6519]⌈ For each RTE API which is subject to variability and which variability shall be implemented and which is following the form *entity internal* in table 4.13 the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define Rte_VPCon_<cts>_<re>[_<resl>]_<ki>_<name>_<sl> <value>
```

where <cts> is the component type symbol of the AtomicSwComponentType,

<re> is the short name of the RunnableEntity,

<resl> is the shortLabel of the RunnableEntity's VariationPoint containing the reference element (e.g. a VariableAccess) to the PortInterface element,

<ki> is the *kind infix* according table 4.13 and

<name> is the short name of the element which is subject to variability in table 4.13 and is defining the API name infix.

<sl> is the shortLabel of the elements' VariationPoint defining the API name infix.

If there is no VariationPoint at the RunnableEntity owning the reference element (e.g. a VariableAccess) to the PortInterface element the <resl> with leading underscore is omitted ([_<resl>]).

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the RTE API in table 4.13. ⌋*(RTE00201, RTE00167)*

[rte_sws_6520]⌈ For each PortPrototype which is subject to variability and which variability shall be implemented the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define Rte_VPCon_<cts>_<p>_<psl> <value>
```

where <cts> is the component type symbol of the AtomicSwComponentType,

<p> is the short name of the PortPrototype and

<psl> is the shortLabel of the PortPrototype's VariationPoint and

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the PortPrototype in table 4.13. ⌋*(RTE00201, RTE00167)*

[rte_sws_6530]⌈ For each RunnableEntity which is subject to variability and which variability shall be implemented the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define Rte_VPCon_<cts>_<re>_<resl> <value>
```

where <cts> is the component type symbol of the AtomicSwComponentType,

<re> is the short name of the RunnableEntity

<resl> is the shortLabel of the RunnableEntity's VariationPoint containing the reference element (e.g. a VariableAccess) to the PortInterface element,

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the RunnableEntity in table 4.16. ⌋*(RTE00201, RTE00167)*

[rte_sws_6541]⌈ For each arraySize which subject to variability the *RTE Configuration Header File* shall contain one definition of a *Attribute Value*

```
#define Rte_VPVal_<t>_<e 1>[_<e 2> ... _<e n>] <value>
```

where <t> is the shortName of the ImplementationDataType,

[<e x>] are the shortNames of the Array's ImplementationDataTypeElements with a leading underscore ordered from the root to the Array's ImplementationDataTypeElement with the arraySize being subject to variability and

<value> is the evaluated value of the AttributeValueVariationPoint of the arraySize ⌋*(RTE00201, RTE00167)*

[rte_sws_6542]⌈ For each Array's ImplementationDataTypeElement which subject to variability the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define Rte_VPCon_<t>_<e 1>[_<e 2> ... _<e n>] <value>
```

where <t> is the shortName of the ImplementationDataType,

[<e x>] are the shortNames of the Array's ImplementationDataTypeElements with a leading underscore ordered from the root to the Array's ImplementationDataTypeElement being subject to variability and

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the conditional existence of the Array's ImplementationDataTypeElement ⌋*(RTE00201, RTE00167)*

[rte_sws_6535]⌈ For each *Basic Software Scheduler* API which is subject to variability and following the form *module internal* in table 4.22 the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define SchM_VPCon_<bsnp>[_<vi>_<ai>]_<ki>_<name>_<sl> <value>
```

where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the BSW module,

<ai> is the vendorApiInfix of the BSW module,

<ki> is the *kind infix* according table 4.22,

<name> is the short name of the element which is subject to variability in table 4.22 defining the *Basic Software Scheduler* API name infix and

<sl> is the shortLabel of the elements' VariationPoint defining the API name infix.

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the *Basic Software Scheduler* API in table 4.22.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00229, BSW00347)*

[rte_sws_6536]⌈ For each *Basic Software Scheduler* API which is subject to variability and which variability shall be implemented and which is following the form *module external* and *entity internal* in table 4.22 the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define SchM_VPCon_<bsnp>[_<vi>_<ai>]_<ki>_
                      <entity>[_<esl>]_<name>[_<sl>] <value>
```

where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the BSW module,

<ai> is the vendorApiInfix of the BSW module,

<ki> is the *kind infix* according table 4.22,

entity is the short name of the BswModuleEntity

<esl> is the shortLabel of the BswModuleEntity's VariationPoint containing the subject to variability,

<name> is the short name of the element which is subject to variability in table 4.22 defining the *Basic Software Scheduler* API name infix and

<sl> is the shortLabel of the elements's VariationPoint defining the API name infix.

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the *Basic Software Scheduler* API in table 4.22.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528.

If there is no VariationPoint at the BswModuleEntity referring to the subject to variability in table 4.22 the <esl> with leading underscore is omitted ([_<esl>]).

If there is no VariationPoint at the elements defining the *Basic Software Scheduler* API name infix 4.22 the <sl> with leading underscore is omitted ([_<sl>]). ⌋*(RTE00229, BSW00347)*

[rte_sws_6532]⌈ For each BswSchedulableEntity which is subject to variability and which variability shall be implemented the *RTE Configuration Header File* shall contain one definition of a *Condition Value*

```
#define SchM_VPCon_<bsnp>[_<vi>_<ai>]_<entry>_<esl> <value>
```

where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the BSW module,

<ai> is the vendorApiInfix of the BSW module,

<entry> is the *short name* of the implemented (implementedEntry) entry point and

<esl> is the shortLabel of the BswModuleEntity's VariationPoint

<value> is the evaluated value of the ConditionByFormula of the VariationPoint defining the variant existence of the BswSchedulableEntity in table 4.24.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00229, BSW00347)*

5.3.9 Generated RTE

Figure 5.1 defines the relationship between generated and standardized header files. It is **not** necessary to standardize the relationship between the C module, Rte.c, and the header files since when the RTE is generated the application header files are created anew along with the RTE. This means that details of which header files are included by Rte.c can be left as an implementation detail.

5.3.9.1 Header File Usage

[rte_sws_1257]⌈ In compatibility mode, the Generated RTE module shall include Os.h. ⌋*(RTE00145)*

[rte_sws_3794]⌈ In compatibility mode, the generated RTE module shall include Com.h. ⌋*(RTE00145)*

[rte_sws_1279]⌈ In compatibility mode, the Generated RTE module shall include Rte.h. ⌋*(RTE00145)*

[rte_sws_1326]⌈ In compatibility mode, the Generated RTE module shall include the VFB Tracing header file. ⌋*(RTE00045, RTE00145)*

[rte_sws_3788]⌈ Except for the declaration of entry points for components (see rte_sws_7194), the RTE shall map its memory objects with the file MemMap.h, using the AUTOSAR memory mapping mechanism (see [26]). ⌋*(RTE00148)*

[rte_sws_7692]⌈ The Generated RTE module shall perform Inter Module Checks to avoid integration of incompatible files. The imported included files shall be checked by preprocessing directives.

The following version numbers shall be verified:

- <MODULENAME>_AR_RELEASE_MAJOR_VERSION

- <MODULENAME>_AR_RELEASE_MINOR_VERSION

Where <MODULENAME> is the module short name of the other (external) modules which provide header files included by the Generated RTE module.

If the values are not identical to the expected values, an error shall be reported. ⌋*(BSW004)*

Figure 5.13 provides an example of how the RTE header and generated header files could be used by a generated RTE.

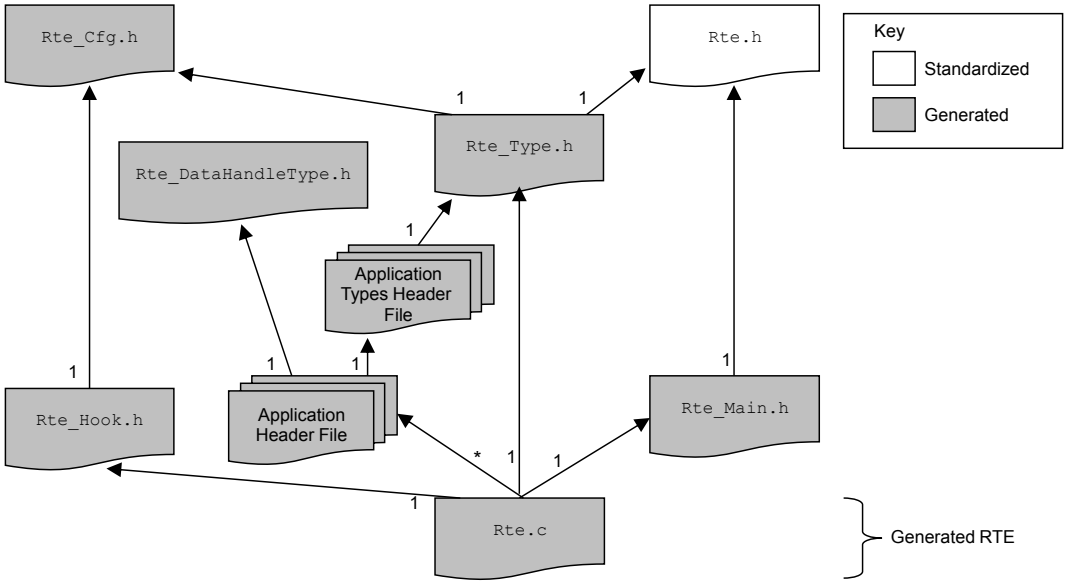


Figure 5.13: Example of header file use by the generated RTE.

In the example in Figure 5.13, the generated RTE C module requires access to the data structures created for each AUTOSAR software-component and therefore includes each application header file[4]. In the example, the generated RTE also includes the RTE header file and the lifecycle header file in order to obtain access to RTE and lifecycle related definitions.

Note: Inclusion of *Application Header File*s of different software components into the RTE C module needs support in the *Application Header File*s in order to avoid that some local definitions of software components are producing name clashes. If the RTE C module does not include any *Application Header File*, some type definitions (e.g. component data structure) might have to be generated twice.

5.3.9.2 C/C++

The following requirements apply to RTEs generated for C and C++.

[4]The requirement that a software module include at most one application header file applies only to modules that actually implement a software-component and therefore does not apply to the generated RTE.

Note: The <CoreID>s referred to in requirements rte_sws_2712, rte_sws_2713 and rte_sws_2740 are implementation-specific identifiers for the modules. They need not be the same as the CoreId identifiers configured for the multi core OS. Refer to section 4.3.4 for a discussion of the allocation of ECU execution logic to partitions and the allocation of partitions to cores.

[rte_sws_1169]⌈ The name of the C module containing the generated RTE code that is shared by all cores of an ECU shall be Rte.c. ⌋*(BSW00300, RTE00126)*

[rte_sws_2711]⌈ On a multi core ECU, RTE shall only use global and static variables in the Rte.c module, if it is used in a single image system that supports shared memory. In this case, RTE shall guarantee consistency of this memory, e.g. by using OS mechanisms. ⌋*()*

[rte_sws_2712]⌈ On a multi core ECU, there shall be additional modules named Rte_Core<CoreID> for the core specific code parts of RTE. ⌋*()*

[rte_sws_2713]⌈ There shall not be symbol redefinitions between different Rte_Core<CoreID>.c modules. ⌋*()*

These requirements makes sure, that all Rte modules can be linked in one image. On a multi core ECU, the RTE may be linked in one image or distributed over separate images, one per core.

An RTE that includes configured code from an object-code or source-code library may use additional modules. Further on due to the encapsulation of a component local name space RTE00167, it might be required to encapsulate part of the generated RTE code in component specific files as well to avoid name clashes in the RTE's implementation.

[rte_sws_7140]⌈ The RTE generator is allowed to partition the generated RTE module in several files additionally to Rte.c and Rte_Core<CoreID>.c. ⌋*(RTE00167)*

5.3.9.3 File Contents

By its very nature the contents of the generated RTE is largely vendor specific. It is therefore only possible to define those common aspects that are visible to the "outside world" such as the names of generated APIs and the definition of component data structures that apply any operating mode.

5.3.9.3.1 Component Data Structures

The *Component Data Structure* (Section 5.4.2) is a per-component data type used to define instance specific information required by the generated RTE.

[rte_sws_3711]⌈ The generated RTE shall contain an instance of the relevant Component Data Structure for each software-component instance on the ECU for which the RTE is generated. ⌋*(RTE00011)*

[rte_sws_3712]⌈ The name of a Component Data Structure instantiated by the RTE generator shall be `Rte_Instance_<name>` where `<name>` is an automatically generated name, created in some manner such that all instance data structure names are unique. ⌋*(BSW00307)*

The software component instance name referred to in rte_sws_3712 is never made visible to the users of the generated RTE. There is therefore no need to specify the precise form that the unique name takes. The `Rte_Instance_` prefix is mandated in order to ensure that no name clashes occur and also to ensure that the structures are readily identifiable in map files, debuggers, etc.

5.3.9.3.2 Generated API

[rte_sws_1266]⌈ The RTE module shall define the generated functions that will be invoked when an AUTOSAR software-component makes an RTE API call. ⌋*(RTE00051)*

The semantics of the generated functions are not defined (since these will obviously vary depending on the RTE API call that it is implementing) nor are the implementation details (which are vendor specific). However, the names of the generated functions defined in Section 5.2.6.1.

The signature of a generated function is the same as the signature of the relevant RTE API call (see Section 5.6) with the exception that the instance handle can be omitted since the generated function is applicable to a specific software-component instance.

5.3.9.3.3 Callbacks

In addition to the generated functions for the RTE API, the RTE module includes callbacks invoked by COM when signal events (receptions, transmission acknowledgement, etc.) occur.

[rte_sws_1264]⌈ The RTE module shall define COM callbacks for relevant signals. ⌋*(RTE00019)*

The required callbacks are defined in Section 5.9.

[rte_sws_3795]⌈ The RTE generator shall generate a separate header file containing the prototypes of the COM callback functions. ⌋*(RTE00019)*

[rte_sws_3796]⌈ The name of the header file containing the callback prototypes shall be `Rte_Cbk.h` in a C/C++environment. ⌋*(RTE00019)*

rte_sws_3796 refers to the callbacks defined in section 5.9.

5.3.9.3.4 Task bodies

The RTE module define task bodies for tasks created by the RTE generator only in compatibility mode.

[rte_sws_1277]⌈ In compatibility mode rte_sws_1257, the RTE module shall define all task bodies created by the RTE generator. ⌋*(RTE00145)*

Note that in vendor mode it is assumed that greater knowledge of the OS is available and therefore the above requirement does *not* apply so that specific optimizations, such as creating each task in a separate module, can be applied.

5.3.9.3.5 Lifecycle API

[rte_sws_1197]⌈ The RTE module shall define the RTE lifecycle API. ⌋*(RTE00051)*

The RTE lifecycle API is defined in Section 5.8.

5.3.9.4 Reentrancy

All code invoked by generated RTE code that can be subject to concurrent execution must be reentrant. This requirement for reentrancy can be overridden if the generated code is not subject to concurrent execution, for example, if protected by a data consistency mechanism to ensure that access to critical regions is serialized.

5.3.10 RTE Post Build Variant Sets

[rte_sws_6620]⌈ The RTE generator shall generate in the *Rte_PBCfg.h* file the `SchM_ConfigType` type declaration of the predefined post build variants data structure. This header file must be used by other RTE modules to resolve their runtime variabilities. ⌋*(RTE00201)*

[rte_sws_6638]⌈ The RTE generator must generate a *Rte_PBCfg.c* file containing the declarations and initializations of one or more RTE post build variants. Only one of these variants can be active at runtime. ⌋*(RTE00201, BSW00346)*

Within an RTE with post build variants, one active RtePostBuildVariantConfiguration will exist. It is a pointer to this structure that shall be passed to SchM_Init. Also note that the container PredefinedVariant is only a Meta Model construct to allow the designer to create a validated collection of values assigned to a criterion. It is up to the implementer of the RTE generator to optimize variant configurations either for size and/or performance by using different levels of indirection to the PostBuildVariantCriterionValues. For the least amount of indirection for example one can have the criterion values at the level of the Sch_ConfigType. If you use post build loadable then you may want to reduce memory storage by reusing variant sets if they remain unchanged across two or more predefined variants.

The following subsections provide examples for the SchM_ConfigType declaration and instantiation only for demonstration purposes. No requirement what so ever is implied.

5.3.10.1 Example 1: File Contents Rte_PBCfg.h

An example of a flat data structure to represent the criterion values defined in the *Rte_PBCfg.h* file containing theSchM_ConfigType type which can contain the list of unique PostBuildVariantCriterion members. This approach immediately enforces that only one single criterion assignment can exist. The member names can, for example, follow the template defined below where <sn> is the PostBuildVariantCriterion shortName.

```
1  struct SchM_ConfigType {
2    /* The PostBuildVariantCriterion shortname */
3      int VarCri_<sn>;
4      .
5      .
6      .
7  };
```

5.3.10.2 Example 2: File Contents Rte_PBCfg.h

An example showing an additional level of indirection and as such allows for reuse of variant sets to optimize memory storage acorss for example several predefined variants is shown below. The RTE generator in this case can reuse some PostBuildVariantSets to reduce the memory resource consumption of an ECU. The RTE generator can declare in the *Rte_PBCfg.h* file a structure type for each **distinct unique** collection of PostBuildVariantSets containing the PostBuildVariantCriterions as members. This implies that if two PredefinedVariants are defined each referring to a named PostBuildVariantSet and the list of PostBuildVariantCriterions in each of these PostBuildVariantSets is identical that only one type is defined for these two named PostBuildVariantSets. The name of the type can,

for example, follow the pattern below where the <id> is a unique identifier for that type (e.g. a counter).

```
struct Rte_VarSet_<id>_t {
  /* The PostBuildVariantCriterion shortname */
    int VarCri_<sn>;
    .
    .
    .
};
```

Now the SchM_ConfigType type can bedeclared with pointers to these variant sets. The member names of this struct can, for example, follow the template below where <id> is a unique identifier.

```
struct SchM_ConfigType {
  /* The PostBuildVariantCriterion shortname */
    Rte_VarSet_<id>_t* VarSet_<id>_Ptr;
    .
    .
    .
};
```

5.3.10.3 Examples: File Contents Rte_PBCfg.c

In correlation with example 1 of the header file the RTE generator can declare and optionaly initialize a default variant configuration named Rte_VarCfg in the *Rte_PBCfg.c* file of the SchM_ConfigType type.

For example (the initializers are the criterion values):

```
const struct SchM_ConfigType Rte_VarCfg = {1,2,3,4,5};
```

And likewise for the example 2 header file the RTE generator can declare and initialize in the *Rte_PBCfg.c* file all possible PostBuildVariantSets and the RtePostBuildVariantConfigurations using references to these variant sets.

For example:

```
const struct Rte_VarSet_1_t Rte_VarSet_1a = {1,2,3};
const struct Rte_VarSet_1_t Rte_VarSet_1b = {1,4,1};
const struct Rte_VarSet_2_t Rte_VarSet_2 = {2,5,7,3,2};
.
.
.
```

```
const struct SchM_ConfigType Rte_VarCfg_1 =
    {&Rte_VarSet_1a,&Rte_VarSet_2};
const struct SchM_ConfigType Rte_VarCfg_2 =
    {&Rte_VarSet_1b,&Rte_VarSet_2};
.
```

```
.
.
```

When `SchM_Init` is called, a pointer to the active `SchM_ConfigType` will be passed along which shall be assigned to the named Rte_VarCfgPtr which is of type `SchM_ConfigType*`. This pointer shall be used to determine the values for actual used `PostBuildVariantCriterion`s and for variant validation when the DET is enabled.

Example 1 pseudo code evaluating the criterions

```
switch(Rte_VarCfg->VarCri_1)
{
    case 1:
        /* DO SOMETHING */
        break;
    case 2:
        /* DO SOMETHING ELSE */
}
```

Example 2 pseudo code evaluating the criterions

```
switch(Rte_VarCfgPtr->VarSet_1_Ptr->VarCri_1)
{
    case 1:
        /* DO SOMETHING */
        break;
    case 2:
        /* DO SOMETHING ELSE */
}
```

Another type of optimization strategy (besides flattening) that can be applied is double buffering for frequently used variant criterion values. The additional buffer can then be used in the conditions to optimize the performance of the RTE (e.g. `BufferedVarCri_1 = Rte_VarCfgPtr->VarSet_1->VarCri_1`).

5.4 RTE Data Structures

Object-code software components are compiled against an application header file created during the “RTE Contract” phase but are linked against an RTE (and application header file) created during the “RTE Generation” phase. When generated in compatibility mode, an RTE has to work for object-code components compiled against an application header file created in compatibility mode, even if the application header file was created by a different RTE generator. It is thus necessary to define the data structures and naming conventions for the compatibility mode to ensure that the object-code is compatible with the generated RTE. An RTE generated in vendor mode only has to work for those object-code components that were compiled against application header files created in vendor mode by a compatible RTE generator (which in general would mean an RTE generator supplied by the same vendor).

The use of standardized data structures imposes tight constraints on the RTE implementation and therefore restricts the freedom of RTE vendors to optimize the solution of object-code components but has the advantage that RTE generators from different vendors can be used to compile an object-code software-component and to generate the RTE. No such restrictions apply for the vendor mode. If an RTE generator operating in vendor mode is used for an object-code component in both phases, vendor-specific optimizations can be used.

Note that with the exception of data structures required for support object-code software components in compatibility mode, the data structures used for “RTE Generation” phase are not defined. This permits vendor specific API mappings and data structures to be used for a generated RTE without loss of portability.

The following definitions only apply to RTE generators operating in compatibility mode – in this mode the instance handle and the component data structure have to be defined even for those (object-code) software components for which multiple instantiation is forbidden to ensure compatibility.

5.4.1 Instance Handle

The RTE is required to support object-code components as well as multiple instances of the same AUTOSAR software-component mapped to an ECU [RTE00011]. To minimise memory overhead all instances of a component on an ECU share code [RTE00012] and therefore both the RTE and the component instances require a means to distinguish different instances.

Support for both object-code components and multiple instances requires a level of indirection so that the correct generated RTE custom function is invoked in response to a component action. The indirection is supplied by the instance handle in combination with the API mapping defined in Section 5.2.6.

[rte_sws_1012]⌈ The component instance handle shall identify particular instances of a component. ⌋*(BSW00312, RTE00011)*

The instance handle is passed to each runnable entity in a component when it is activated by the RTE as the first parameter of the function implementing the runnable entity rte_sws_1016. The instance handle is then passed back by the runnable entity to the RTE, as the first parameter of each direct RTE API call, so that the RTE can identify the correct component instance making the call. This scheme permits multiple instances of a component on the same ECU to share code.

The instance handle indirection permits the name of the RTE API call that is used within the component to be unique within the scope of a component as well as independent of the component’s instance name. It thus enables object-code AUTOSAR software-

components to be compiled before the final “RTE Generation” phase when the instance name is fixed.

[rte_sws_1013]⌈ For the RTE C/C++ API, any call that can operate on different instances of a component that supports multiple instantiation `supportsMultipleInstantiation` shall have an instance handle as the first formal parameter. ⌋*(RTE00011)*

[rte_sws_3806]⌈ If a component does not support multiple instantiation, the instance handle parameter shall be omitted in the RTE C/C++ API and in the signature of the RTE Hook functions. ⌋*(RTE00011)*

If the component does not support multiple instantiation, the name of the instance handle must be specified, since it is not passed to the API calls and runnable entities as parameters.

[rte_sws_3793]⌈ If a software component does not support multiple instantiation, the name of the instance handle shall be `Rte_Inst_<cts>`, where `<cts>` is the `component type symbol` of the `AtomicSwComponentType`. ⌋*(RTE00011)*

The data type of the instance handle is defined in Section 5.5.2.

5.4.2 Component Data Structure

Different component instances share many common features - not least of which is support for shared code. However, each instance is required to invoke different RTE API functions and therefore the instance handle is used to access the component data structure that defines all instance specific data.

It is necessary to define the component data structure to ensure compatibility between the two RTE phases when operating in compatibility mode – for example, a “clever” compiler and linker may encode type information into a pointer type to ensure type-safety. In addition, the structure definition cannot be empty since this is an error in ANSI C.

[rte_sws_7132]⌈ The component data structure type shall be defined in the *Application Header* file. ⌋*(RTE00011, RTE00167)*

[rte_sws_3714]⌈ The type name of the component data structure shall be `Rte_CDS_<cts>` where `<cts>` is the `component type symbol` of the `AtomicSwComponentType`. ⌋*(BSW00305)*

The members of the component data structure include function pointers. It is important that such members are not subject to run-time modification and therefore the component data structure is required to be placed in read-only memory.

[rte_sws_3715]⌈ All instances of the component data structure shall be defined as “const” (i.e. placed in read-only memory). ⌋*(BSW007)*

The elements of the component data structure are sorted into sections, each of which defines a logically related section. The sections defined within the component data structure are:

- **[rte_sws_3718]**⌈ Data Handles section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_3719]**⌈ Per-instance Memory Handles section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_1349]**⌈ Inter-runnable Variable Handles section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_3720]**⌈ Calibration Parameter Handles section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_3721]**⌈ Exclusive-area API section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_3716]**⌈ Port API section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_3717]**⌈ Inter Runnable Variable API section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_7225]**⌈ Inter Runnable Triggering API section. ⌋*(RTE00011, RTE00051)*
- **[rte_sws_3722]**⌈ Vendor specific section. ⌋*(RTE00011)*

The order of elements within each section of the component data structure is defined as follows;

[rte_sws_3723]⌈ Section entries shall be sorted alphabetically (ASCII / ISO 8859-1 code in ascending order) unless stated otherwise. ⌋*(RTE00051)*

The sorting of entries is applied to each section in turn.

Note that there is *no* prefix associated with the name of each entry within a section; the component data structure as a whole has the prefix and therefore there is no need for each member to have the same prefix.

ANSI C does not permit empty structure definitions yet an instance handle is required for the RTE to function. Therefore if there are no API calls then a single dummy entry is defined for the RTE.

[rte_sws_3724]⌈ If all sections of the Component Data Structure are empty the Component Data Structure shall contain a `uint8` with name `_dummy`. ⌋*(RTE00126)*

5.4.2.1 Data Handles Section

The data handles section is required to support the `Rte_IRead` and `Rte_IWrite` calls (see Section 5.2.4).

[rte_sws_3733]⌈ Data Handles shall be named `<re>_<p>_<o>` where `<re>` is the runnable entity name that reads (or writes) the data item, `<p>` the port name, `<o>` the data element. ⌋*(BSW00305, RTE00051)*

A runnable cannot read *and* write to the same port/data element since the port is inherently uni-directional (a provide port can only be written, a require port can only be read).

[rte_sws_2608]⌈ The Data Handle shall be a pointer to a `Data Element with Status` if and only if either

- the runnable has read access and either
  - `data element outdated` notification or
  - `data element invalidation`

  is activated for this `data element`, or
- the runnable has write access and acknowledgement is enabled for this `data element`.

⌋*(RTE00051, RTE00185)*

[rte_sws_2588]⌈ Otherwise, the data type for a Data Handle shall be a pointer to a `Data Element without Status`. ⌋*(RTE00051)*

See below for the definitions of these terms.

[rte_sws_6529]⌈ The RTE Generator shall wrap each entry of *Data Handles Section* in the component data structure of a variant existent `Rte_IRead` or `Rte_IWrite` API if the variability shall be implemented.

```
#if (<condition>)

<Data Handles Section Entry>

#endif
```

where `condition` is the condition value macro of the `VariationPoint` relevant for the variant existence of the `Rte_IRead` or `Rte_IWrite` API (see rte_sws_6515), `Data Handles Section Entry` is the code according an invariant *Data Handles Section Entry* (see also rte_sws_3733, rte_sws_2608, rte_sws_2588) ⌋*(RTE00201)*

5.4.2.1.1 Data Element without Status

[rte_sws_1363]⌈ The data type for a “Data Element without Status” shall be named `Rte_DE_<dt>` where `<dt>` is the data element’s `ImplementationDataType` name. ⌋*(RTE00051)*

[rte_sws_1364]⌈ A `Data Element without Status` shall be a structure containing a single member named `value`. ⌋*(RTE00051)*

[rte_sws_2607]⌈ The `value` member of a `Data Element without Status` shall have the same data type as the corresponding DataElement. ⌋*(RTE00051, RTE00147, RTE00078)*

Note that requirements rte_sws_1364 and rte_sws_2607 together imply that creating a variable of data type `Rte_DE_<dt>` allocates enough memory to store the data copy.

5.4.2.1.2 Data Element with Status

[rte_sws_1365]⌈ The data type for a “Data Element with Status” shall be named `Rte_DES_<dt>` where `<dt>` is the data element’s `ImplementationDataType` name. ⌋*(RTE00051)*

[rte_sws_1366]⌈ A `Data Element with Status` shall be a structure containing two members. ⌋*(RTE00051)*

[rte_sws_3734]⌈ The first member of each `Data Element with Status` shall be named ’value’ ⌋*(RTE00051)*

[rte_sws_2666]⌈ The value member of a `Data Element with Status` shall have the type of the corresponding DataElement. ⌋*(RTE00051, RTE00147, RTE00078, RTE00185)*

[rte_sws_2589]⌈ The second member of each `Data Element with Status` shall be named ’status’. ⌋*(RTE00051, RTE00147, RTE00078, RTE00185)*

[rte_sws_2590]⌈ The status member of a `Data Element with Status` shall be of the `Std_ReturnType` type. ⌋*(RTE00147, RTE00078, RTE00185)*

[rte_sws_2609]⌈ In case of read access, the status member of a `Data Element with Status` shall contain the error status corresponding to the value member. ⌋*(RTE00147, RTE00078)*

[rte_sws_3836]⌈ In case of write access, the status member of a `Data Element with Status` shall contain the transmission status corresponding to the value member. ⌋*(RTE00185)*

5.4.2.1.3 Usage

[rte_sws_7136]⌈ A definition for every required `Data Element with Status` and every `Data Element without Status` must be emitted in the *RTE Data Handle Types Header File* (see Section 5.3.5). ⌋*(RTE00051)*

Example 5.24

Consider a `uint8` data element, `a`, of port `p` which is accessed using a `VariableAccess` in the `dataWriteAccess` role by runnables `re1` and `re2` and a `VariableAccess` in the `dataReadAccess` role by runnable `re2` within component `c`. `data element outdated` is defined for this `dataElement`.

The required data types within the *RTE Data Handle Types Header File* would be:

```
typedef struct {
  uint8 value;
} Rte_DE_uint8;

typedef struct {
  uint8 value;
  Std_ReturnType status;
} Rte_DES_uint8;
```

The component data structure for `c` would also include:

```
Rte_DE_uint8*  re1_p_a;
Rte_DES_uint8* re2_p_a;
```

A software-component that is supplied as object-code or is multiple instantiated requires "general purpose" definitions of `Rte_IRead`, `Rte_IWrite`, and `Rte_IStatus` that use the data handles to access the data copies created within the generated RTE. For example:

```
#define Rte_IWrite_re1_p_a(s,v) ((s)->re1_p_a->value = (v))
#define Rte_IWrite_re2_p_a(s,v) ((s)->re2_p_a->value = (v))
#define Rte_IRead_re2_p_a(s,v)  ((s)->re2_p_a->value)
#define Rte_IStatus_re2_p_a(s)  ((s)->re2_p_a->status)
```

The definitions of `Rte_IRead`, `Rte_IWrite`, and `Rte_IStatus` are type-safe since an attempt to assign an incorrect type will be detected by the compiler.

For source code component that does **not** use multiple instantiation the definitions of `Rte_IRead`, `Rte_IWrite`, and `Rte_IStatus` can remain as above or vendor specific optimizations can be applied without loss of portability.

The values assigned to data handles within *instances* of the component data structure created within the generated RTE depend on the mapping of tasks and runnables – See Section 5.2.4.

5.4.2.2 Per-instance Memory Handles Section

The Per-instance Memory Section Handles section enables to access instance specific memory (sections).

[rte_sws_2301]⌈ The CDS shall contain a handle for each Per-instance Memory. This handle member shall be named `Pim_<name>` where `<name>` is the per-instance memory name. ⌋*(BSW00305, RTE00051, RTE00013)*

The Per-instance Memory Handles are typed; **[rte_sws_2302]**⌈ The data type of each Per-instance Memory Handle shall be a pointer to the type of the per instance memory that is defined in the *Application Header* file. ⌋*(RTE00051, RTE00013)*

The RTE supports the access to the per-instance memories by the `Rte_Pim` API.

[rte_sws_6527]⌈ The RTE Generator shall wrap each entry of *Per-instance Memory Handles Section* in the component data structure of a variant existent `PerInstanceMemory` or `arTypedPerInstanceMemory` if the variability shall be implemented.

```
#if (<condition>)

<Per-instance Memory Handles Section Entry>

#endif
```

where `condition` is the condition value macro of the `VariationPoint` relevant for the variant existence of the `Rte_Pim` API (see rte_sws_6518), `Per-instance Memory Handles Section Entry` is the code according an invariant *Per-instance Memory Handles Section Entry* (see also rte_sws_2301, rte_sws_2302) ⌋*(RTE00201)*

Example 5.25

Referring to the specification items rte_sws_2301, rte_sws_2302, and rte_sws_7133 Example 5.4 can be extended –

with respect to the software-component header:

```
struct Rte_CDS_c {
  ...
  /* per-instance memory handle section */
  Rte_PimType_c_MyMemType *Pim_mem;

  ...
};

#define Rte_Pim_mem(s) ((s)->Pim_mem)
```

and in Rte.c:

```
Rte_PimType_c_MyMemType mem1;

```

```
const struct Rte_CDS_c Rte_Instance_c1 = {
  ...
  /* per-instance memory handle section */
  /* Rte_PimType_c_MyMemType Pim_mem */
  &mem1
  ...
};
```

5.4.2.3 Inter Runnable Variable Handles Section

Each runnable may require separate handling for the inter runnable variables that it accesses. The indirection required for explicit access to inter runnable variables is described in section 5.4.2.7. The inter runnable variable handles section within the component data structure contains pointers to the (shadow) memory of inter runnable variables that can be directly accessed with the implicit API macros. The inter runnable variable handles section does not contain pointers for memory to handle inter runnable variables that are accessed with explicit API only.

[rte_sws_2636]⌈ For each runnable and each inter runnable variable that is accessed implicitly by the runnable, there shall be exactly one inter runnable handle member within the component data structure and this inter runnable variable handle shall point to the (shadow) memory of the inter runnable variable for the runnable. ⌋*(RTE00142)*

[rte_sws_1350]⌈ The name of each inter runnable variable handle member within the component data structure shall be `Irv_<re>_<o>` where `<o>` is the Inter-Runnable Variable short name and `<re>` is short name of the runnable name. ⌋*(RTE00142)*

[rte_sws_1351]⌈ The data type of each inter runnable variable handle member shall be a pointer to the type of the inter runnable variable. ⌋*(RTE00142)*

[rte_sws_6528]⌈ The RTE Generator shall wrap each entry of *Inter Runnable Variable Handles Section* in the component data structure of a variant existent `Rte_IrvRead` or `Rte_IrvWrite` if the variability shall be implemented.

```
#if (<condition> [|| <condition>])

<Inter Runnable Variable Handles Section Entry>

#endif
```

where `condition` are the condition value macro(s) of the `Variation-Point` relevant for the variant existence of the `Rte_IrvRead` or `Rte_IrvWrite` API accessing the same *Inter Runnable Variable* (see rte_sws_6519), `Inter Runnable Variable Handles Section Entry` is the code according an invariant *Inter Runnable Variable Handles Section Entry* (see also rte_sws_2636, rte_sws_1350, rte_sws_1351) ⌋*(RTE00201)*

5.4.2.4 Exclusive-area API Section

The exclusive-area API section includes exclusive areas that are accessed explicitly, using the RTE API, by the SW-C. Each entry in the section is a function pointer to the relevant RTE API function generated for the SW-C instance.

[rte_sws_3739]⌈ The name of each Exclusive-area API section entry shall be `<root>_<name>` where `<root>` is either `Entry` or `Exit` and `<name>` is the Exclusive-area name. ⌋*(RTE00051, RTE00032)*

[rte_sws_3740]⌈ The data type of each Exclusive-area API section entry shall be a function pointer that points to the generated RTE API function. ⌋*(RTE00051, RTE00032)*

[rte_sws_6521]⌈ The RTE Generator shall wrap each definition of a variant existent `Rte_Enter` and `Rte_Exit` in the Exclusive-area API section according table 4.13 if the variability shall be implemented.

```
#if (<condition>)

<Exclusive-area API section entry>

#endif
```

where `condition` is the condition value macro of the `VariationPoint` relevant for the variant existence of the `Rte_Enter` and `Rte_Exit` API (see rte_sws_6518), `Exclusive-area API section entry` is the code according an invariant Exclusive-area section entry (see also rte_sws_3739, rte_sws_3740) ⌋*(RTE00201)*

[rte_sws_3812]⌈ Entries in the Exclusive-area API section shall be sorted alphabetically. ⌋*(RTE00051, RTE00032)*

Note that two function pointers will be required for each accessed exclusive area; one for the Entry function and one for the Exit function.

5.4.2.5 Port API Section

Port API section comprises zero or more *function references* within the component data structure type that defines all API functions that access a port and can be invoked by the software-component (instance).

[rte_sws_2616]⌈ The function table entries for port access shall be grouped by the port names into port data structures. ⌋*(RTE00051)*

Each entry in the port API section of the component data structure is a “port data structure”.

[rte_sws_2617]⌈ The name of each *port data structure* in the component data structure shall be <p> where <p> is the port short-name. ⌋*(RTE00051)*

[rte_sws_3799]⌈ The component data structure shall contain a port data structure for port p only if the component supports multiple instantiation or if the indirectAPI attribute for p is set to ’true’. ⌋*(RTE00051)*

[rte_sws_6522]⌈ The RTE Generator shall wrap each *port data structure* of a variant existent PortPrototype if the variability shall be implemented.

```
#if (<condition>)

<port data structure>

#endif
```

where condition is the condition value macro of the VariationPoint relevant for the variant existence of the PortPrototype (see rte_sws_6520, port data structure is the code according an invariant *port data structure*s (see also rte_sws_2617, rte_sws_3799) ⌋*(RTE00201)*

[rte_sws_3731]⌈ The data type name for a port data structure shall be
struct Rte_PDS_<cts>_<i>_<P/R>

where <cts> is the component type symbol of the AtomicSwComponentType,

<i> is the port interface name and

‘P’ or ‘R’ are literals to indicate provide or require ports respectively. ⌋*(BSW00305, RTE00051)*

[rte_sws_7137]⌈ The port data structure type(s) shall be defined in the *Application Header* file. ⌋*(RTE00051)*

A port data structure type is defined for each port interface that types a port. Thus different ports typed by the same port interface structure share the same port data structure type.

[rte_sws_7138]⌈ The *Application Header* file shall contain a definition of a port data structure type for interface i and port type R or P only if the component supports multiple instantiation or at least one require or provide port exists that has the indirectAPI attribute set to ’true’. ⌋*(RTE00051)*

[rte_sws_6523]⌈ The RTE Generator shall wrap each *port data structure type* related to variant existent PortPrototypes if the variability shall be implemented and if all require PortPrototypes or all provide PortPrototypes are variant.

```
#if (<condition> [|| <condition>])

<port data structure type>
```

```
4
5 #endif
```

where `condition` are the condition value macro(s) of the `VariationPoint`s relevant for the variant existence of the `PortPrototype`s requiring the *port data structure type* (see rte_sws_6520), `port data structure type` is the code according an invariant *port data structure type* (see also rte_sws_3731, rte_sws_7138, rte_sws_3730 rte_sws_2620) ⌋*(RTE00201)*

Note: If any invariant `PortPrototype` requires the *port data structure type* it shall be defined unconditional.

[rte_sws_7677]⌈ The RTE shall support an indirect API for the port access functions listed in table 5.1. ⌋*(RTE00051)*

[rte_sws_3730]⌈ A port data structure shall contain a function table entry for each API function associated with the port as referenced in table 5.1. Pure API macros, like `Rte_IRead` and other implicit API functions, do not have a function table entry. ⌋*(RTE00051)*

API function	reference
`Rte_Send_<p>_<o>`	5.6.5
`Rte_Write_<p>_<o>`	5.6.5
`Rte_Switch_<p>_<o>`	5.6.6
`Rte_Invalidate_<p>_<o>`	5.6.7
`Rte_Feedback_<p>_<o>`	5.6.8
`Rte_SwitchAck_<p>_<o>`	5.6.9
`Rte_Read_<p>_<o>`	5.6.10
`Rte_DRead_<p>_<o>`	5.6.10
`Rte_Receive_<p>_<o>`	5.6.12
`Rte_Call_<p>_<o>`	5.6.13
`Rte_Result_<p>_<o>`	5.6.14
`Rte_Prm_<p>_<o>`	5.6.17
`Rte_Mode_<p>_<o>`	5.6.29
`Rte_Trigger_<p>_<o>`	5.6.31
`Rte_IsUpdated_<p>_<o>`	5.6.34

Table 5.1: Table of API functions that are referenced in the port API section.

[rte_sws_2620]⌈ An API function shall only be included in a port data structure, if it is required at least by one port. ⌋*(RTE00051)*

[rte_sws_2621]⌈ If a function table entry is available in a port data structure, the corresponding function shall be implemented for all ports that use this port data structure type. API functions related to ports that are not required by the AUTOSAR configuration shall behave like those for an unconnected port. ⌋*(RTE00051)*

APIs may be required only for some ports of a software component instance due to differences in for example the need for transmission acknowledgement. rte_sws_2621 is necessary for the concept of the indirect API. It allows iteration over ports.

[rte_sws_1055]⌈ The name of each function table entry in a port data structure shall be `<name>_<o>` where `<name>` is the API root (e.g. Call, Write) and `<o>` the data element or operation name. ⌋*(BSW00305, RTE00051)*

Requirement rte_sws_1055 does *not* include the port name in the function table entry name since the port is implicit when using a port handle.

[rte_sws_3726]⌈ The data type of each function table entry in a port data structure shall be a function pointer that points to the generated RTE function. ⌋*(RTE00051)*

The signature of a generated function, and hence the definition of the function pointer type, is the same as the signature of the relevant RTE API call (see Section 5.6) with the exception that the instance handle is omitted.

Example 5.26

This example shows a port data structure for the provide ports of the interface type `i2` in an AUTOSAR SW-C `c`.

`i2` is a SenderReceiverInterface which contains a data element prototype of type `uint8` with data semantics.

If one of the provide ports of `c` for the interface `i2` has a transmission acknowledgement defined and `i2` is not used with `data element invalidation`, the *Application Header* file would include a port data structure type like this:

```
1 struct Rte_PDS_c_i2_P {
2   Std_ReturnType (*Feedback_a)(uint8);
3   Std_ReturnType (*Write_a)(uint8);
4 }
```

If the provide port `p1` of the AUTOSAR SW-C `c` is of interface `i2`, the generated *Application Header* file would include the following macros to provide the direct API functions `Rte_Feedback_p1_a` and `Rte_Write_p1_a`:

```
1 /*direct API*/
2 #define Rte_Feedback_p1_a(inst,data)
3         ((inst)->p1.Feedback_a)(data)
4 #define Rte_Write_p1_a(inst,data) ((inst)->p1.Write_a)(data)
```

[rte_sws_2618]⌈ The port data structures within a component data structure shall first be sorted on the port data structure type name and then on the short name of the port. ⌋*(RTE00051)*

The requirements rte_sws_3731 and rte_sws_2618 guarantee, that all port data structures within the component data structure are grouped by their interface type and require/provide-direction.

Example 5.27

This example shows the grouping of port data structures within the component data structure.

The *Application Header* file for an AUTOSAR SW-C `c` with three provide ports `p1`, `p2`, and `p3` of interface `i2` would include a block of port data structures like this:

```
struct Rte_CDS_c {
  ...
  struct Rte_PDS_c_i1_R z;

/* component data structures          *
 * for provide ports of interface i2  */
  struct Rte_PDS_c_i2_P p1;
  struct Rte_PDS_c_i2_P p2;
  struct Rte_PDS_c_i2_P p3;

/*further component data structures*/
  struct Rte_PDS_c_i2_R c;
  ...
}

```

If `inst` is a pointer to a component data structure, and `ph` is defined by

```
struct Rte_PDS_c_i2_P *ph = &(inst->p1);
```

`ph` points to the port data structure `p1` of the instance handle `inst`. Since the three provide port data structures `p1`, `p2`, and `p3` of interface `i2` are ordered sequentially in the component data structure, `ph` can also be interpreted as an array of port data structures. E.g., `ph[2]` is equal to `inst->p3`.

In the following, `ph` will be called a port handle.

[rte_sws_1343]⌈ RTE shall create *port handle type*s for each port data structure using `typedef` to a pointer to the appropriate port data structure. ⌋*(RTE00051)*

[rte_sws_1342]⌈ The *port handle type* name shall be `Rte_PortHandle_<i>_<P/R>` where `<i>` is the port interface name and ‘P’ or ‘R’ are literals to indicate provide or receive ports respectively. ⌋*(RTE00051)*

[rte_sws_6524]⌈ The RTE Generator shall wrap each *port handle type* related to variant existent `PortPrototype`s if the variability shall be implemented and if all require `PortPrototype`s or all provide `PortPrototype`s are variant.

```
#if (<condition> [|| <condition>])

<port handle type>

#endif
```

where condition are the condition value macro(s) of the VariationPoints relevant for the variant existence of the PortPrototypes requiring the *port data structure type* (see rte_sws_6520), port data structure type is the code according an invariant *port data structure type* (see also rte_sws_1343, rte_sws_1342) ⌋*(RTE00201)*

[rte_sws_1053]⌈ The port handle types shall be written to the application header file. ⌋*(RTE00051)*

RTE provides port handles for access to the arrays of port data structures of the same interface type and provide/receive direction by the macro Rte_Ports, see section 5.6.1, and to the number of similar ports by the macro Rte_NPorts, see 5.6.1.

Example 5.28

For the provide port i2 of AUTOSAR SW-C c from example 5.26, the following port handle type will be defined in the *Application Header* file:

```
typedef struct Rte_PDS_c_i2_P *Rte_PortHandle_i2_P;
```

The macros to access the port handles for the indirect API might look like this in the generated *Application Header* file:

```
/*indirect (port oriented) API*/
#define Rte_Ports_i2_P(inst) &((inst)->p1)
#define Rte_NPorts_i2_P(inst) 3
```

So, the port handle ph of the previous example 5.27 could be defined by a user as:

```
Rte_PortHandle_i2_P ph = Rte_Ports_i2_P(inst);
```

To write ‘49’ on all ports p1 to p3, the indirect API can be used within the software component as follows:

```
uint8 p;
Rte_PortHandle_i2_P ph = Rte_Ports_i2_P(inst);
for(p=0;p<Rte_NPorts_i_P(inst);p++) {
  ph[p].Write_a(49);
}
```

Software components may also want to set up their own port handle arrays to iterate over a smaller sub group than all ports with the same interface and direction. Rte_Port can be used to pick the port handle for one specific port, see 5.6.3.

5.4.2.6 Calibration Parameter Handles Section

The RTE is required to support access to calibration parameters derived by *per-instance* ParameterDataPrototypes (see 4.2.8.3) using the Rte_CData (see section 5.6.16).

[rte_sws_3835]⌈ The name of each Calibration parameter handle shall be CData_<name> where <name> is the ParameterDataPrototype name. ⌋*(RTE00051, RTE00154, RTE00155)*

[rte_sws_3949]⌈ The type of each calibration parameter handle shall be a function pointer that points to the generated RTE function. ⌋*(RTE00051, RTE00154, RTE00155)*

Note that accesses to ParameterDataPrototypes within ParameterSwComponentTypes do not require handles within this section since the generated Rte_Prm (see section 5.6.17) API is accessed either directly (single instantiation) or through handles in the port API section (multiple instantiation). Likewise, access to *shared* ParameterDataPrototypes does not require a handle since, by definition, no per-instance data is present.

5.4.2.7 Inter Runnable Variable API Section

The Inter Runnable Variable API section comprises zero or more *function table entries* within the component data structure type that defines all explicit API functions to access an inter runnable variable by the software-component (instance). The API for implicit access of inter runnable variables does not have any *function table entries*, since the implicit API uses macro's to access the inter runnable variables or their shadow memory directly, see section 5.4.2.3.

Since the entries of this section are only required to access the explicit InterRunnableVariable API if a software component supports multiple instantiation, it shall be omitted for software components which do not support multiple instantiation.

[rte_sws_3725]⌈ If the component supports multiple instantiation, the member name of each function table entry within the component data structure shall be <name>_<re>_<o> where <name> is the API root (e.g. IrvRead), <re> the runnable name, and <o> the inter runnable variable name. ⌋*(RTE00051)*

[rte_sws_3752]⌈ The data type of each function table entry shall be a function pointer that points to the generated RTE function. ⌋*(RTE00051)*

The signature of a generated function, and hence the definition of the function pointer type, is the same as the signature of the relevant RTE API call (see Section 5.6) with the exception that the instance handle is omitted.

[rte_sws_2623]⌈ If the component supports multiple instantiation, the *Inter Runnable Variable API Section* shall contain pointers to the following API functions:

API function	reference
Rte_IrvRead_<re>_<o>	5.6.25
Rte_IrvWrite_<re>_<o>	5.6.26

Table 5.2: Table of API functions that are referenced in the inter runnable variable API section

⌋*(RTE00051)*

[rte_sws_6525]⌈ The RTE Generator shall wrap each entry of *Inter Runnable Variable API Section* in the component data structure of a variant existent `Rte_IrvRead` or `Rte_IrvWrite` API if the variability shall be implemented.

```
#if (<condition>)

<Inter Runnable Variable API Section Entry>

#endif
```

where `condition` is the condition value macro of the `VariationPoint` relevant for the variant existence of the `Rte_IrvRead` or `Rte_IrvWrite` API (see rte_sws_6519), `Inter Runnable Variable API Section Entry` is the code according an invariant *Inter Runnable Variable API Section Entry* (see also rte_sws_3725, rte_sws_3752, rte_sws_2623) ⌋*(RTE00201)*

[rte_sws_3791]⌈ If the software component does not support multiple instantiation, the inter runnable variable API section shall be empty. ⌋*(RTE00051)*

5.4.2.8 Inter Runnable Triggering API Section

The Inter Runnable Triggering API Section includes the Inter Runnable Triggering API handles. Each entry in the section is a function pointer to the relevant RTE API function generated for the SW-C instance.

[rte_sws_7226]⌈ The name of each *Inter Runnable Triggering handle* shall be `Rte_IrTrigger_<re>_<name>` where `<re>` is the name of the runnable entity the API might be used and `<name>` is the name of the `InternalTriggeringPoint`. ⌋*(RTE00051, RTE00163)*

[rte_sws_7227]⌈ The data type of each *Inter Runnable Triggering handle entry* shall be a function pointer that points to the generated RTE API function defined in 5.6.32. ⌋*(RTE00051, RTE00163)*

[rte_sws_6526]⌈ The RTE Generator shall wrap each entry of *Inter Runnable Triggering handle* in the component data structure of a variant existent `Rte_IrTrigger` API if the variability shall be implemented.

```
#if (<condition>)

```

```
3  <Inter Runnable Variable API Section Entry>
4
5  #endif
```

where `condition` is the condition value macro of the `VariationPoint` relevant for the variant existence of the `Rte_IrTrigger` API (see rte_sws_6519, `Inter Runnable Variable API Section Entry` is the code according an invariant *Inter Runnable Variable API Section Entry* (see also rte_sws_3725, rte_sws_3752, rte_sws_2623) ⌋*(RTE00201)*

[rte_sws_7228]⌈ Entries in the Inter Runnable Triggering handles section shall be sorted alphabetically. ⌋*(RTE00051, RTE00163)*

5.4.2.9 Vendor Specific Section

The vendor specific section is used to contain any vendor specific data required to be supported for each instances. By definition the contents of this section are outside the scope of this chapter and only available for use by the RTE generator responsible for the “RTE Generation” phase.

5.5 API Data Types

Besides the API functions for accessing RTE services, the API also contains RTE-specific data types.

5.5.1 Std_ReturnType

The specification in [29] specifies a standard API return type `Std_ReturnType`. The `Std_ReturnType` defines the "‘status"’ and "‘error values"’ returned by API functions. It is defined as a `uint8` type. The value “0” is reserved for “No error occurred”.

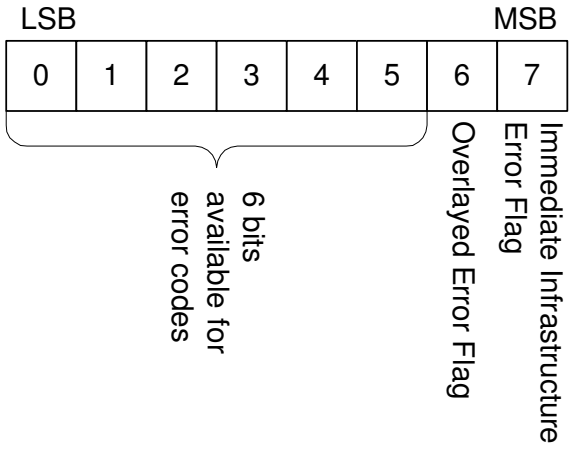


Figure 5.14: Bit-Layout of the Std_ReturnType

Figure 5.14 shows the general layout of `Std_ReturnType`.

The two most significant bits of the `Std_ReturnType` are reserved flags:

- The most significant bit 7 of `Std_ReturnType` is the “Immediate Infrastructure Error Flag” with the following values
  - “1” the error code indicates an immediate infrastructure error.
  - “0” the error code indicates no immediate infrastructure error.
- The second most significant bit 6 of `Std_ReturnType` is the Overlayed Error Flag. The use of this flag depends on the context and will be explained in table 5.4.

In order to avoid explicit access to bit numbers in the code, the RTE provides the three following macros that enables an application to check the return value of an API:

- **[rte_sws_7404]**⌈ For infrastructure errors, this macro is a boolean expression that is true if the corresponding bit is set:

```
#define Rte_IsInfrastructureError(status)  ((status & 128U) != 0)
```

⌋()

- **[rte_sws_7405]**⌈For overlayed errors, this macro is a boolean expression that is true if the corresponding bit is set:

```
#define Rte_HasOverlayedError(status)  ((status & 64U) != 0)
```

⌋()

- **[rte_sws_7406]**⌈For reading only the application error code without the eventual overlayed error, the following macro returns the lower 6 bits of the error code:

```
#define Rte_ApplicationError(status) (status & 63U)
```

⌋()

5.5.1.1 Infrastructure Errors

Infrastructure errors are split into two groups:

- "Immediate Infrastructure Errors" can be associated with the currently available data set. These `Immediate Infrastructure Errors` are mutually exclusive. Only one of these errors can be notified to a SW-C with one API call.

  [rte_sws_2593]⌈ `Immediate Infrastructure Errors` shall override any application level error. ⌋*(RTE00084, RTE00123)*

  `Immediate Infrastructure Error` codes are used on the receiver side for errors that result in no reception of application data and application errors.

  An `Immediate Infrastructure Error` is indicated in the `Std_ReturnType` by the `Immediate Infrastructure Error Flag` being set.

- "Overlayed Errors" are associated with communication events that happened after the reception of the currently available data set, e.g., `data element outdated` notification, or loss of data elements due to queue overflow.

  [rte_sws_1318]⌈ `Overlayed Error Flags` shall be reported using the unique bit of the `Overlayed Error Flag` within the `Std_ReturnType` type. ⌋*(RTE00084, RTE00094)*

  An `Overlayed Error` can be combined with any other application or infrastructure error code.

5.5.1.2 Application Errors

[rte_sws_2573]⌈ RTE shall support application errors with the following format definition: Application errors are coded in the least significant 6 bits of `Std_ReturnType` with the `Immediate Infrastructure Error Flag` set to "0". The application error code does not use the `Overlayed Error Flag`. ⌋*(RTE00124)*

This results in the following value range for application errors:

range	minimum value	maximum value
application errors	1	63

Table 5.3: application error value range

In client server communication, the server may return any value within the application error range. The client will then receive one of the following:

- An `Immediate Infrastructure Error` to indicate that the communication was not successful, or

- The server return code, or
- The server return code might be overlayed by the `Overlayed Error Flag` in a future release of RTE. In this release, there is no overlayed error defined for client server communication.

The client can filter the return value, e.g., by using the following code:

```
Std_ReturnType status;
status = Rte_Call_<p>_<o>(<instance>, <parameters>);
if (Rte_HasOverlayedError(status)) {
    /* handle overlayed error flag                          *
     * in this release of the RTE, the flag is reserved     *
     * but not used for client server communication         */

}

if(Rte_IsInfrastructureError(status)) {
    /* handle infrastructure error                          */
}
else {
    /* handle application error with error code status      */
    status = Rte_ApplicationError(status);
}
```

5.5.1.3 Predefined Error Codes

For client server communication, application error values are defined per client server interface and shall be passed to the RTE with the interface configuration.

The following standard error and status identifiers are defined:

Symbolic name	Value	Comments
[rte_sws_1058] `RTE_E_OK`	0	No error occurred.

Standard Application Error Values:		
[rte_sws_2594] `RTE_E_INVALID`	1	Generic application error indicated by signal invalidation in sender receiver communication with data semantics on the receiver side.
To be defined by the corresponding AUTOSAR Service	1	Returned by AUTOSAR Services to indicate a generic application error.

`Immediate Infrastructure Error` codes

Symbolic name	Value	Comments
[rte_sws_1060] RTE_E_COM_STOPPED	128	An IPDU group was disabled while the application was waiting for the transmission acknowledgment. No value is available. This is not considered a fault, since the IPDU group is switched off on purpose. This semantics are as follows: • The OUT buffers of a client or of explicit read APIs are not modified • no runnable with startOnEvent on a DataReceivedEvent for this VariableDataPrototype is triggered. • the buffers for implicit read access will keep the previous value.
[rte_sws_1064] RTE_E_TIMEOUT	129	A blocking API call returned due to expiry of a local timeout rather than the intended result. OUT buffers are not modified. The interpretation of this being an error depends on the application.
[rte_sws_1317] RTE_E_LIMIT	130	A internal RTE limit has been exceeded. Request could not be handled. OUT buffers are not modified.
[rte_sws_1061] RTE_E_NO_DATA	131	An explicit read API call returned no data. (This is no error.)
[rte_sws_1065] RTE_E_TRANSMIT_ACK	132	Transmission acknowledgement received.
[rte_sws_7384] RTE_E_NEVER_RECEIVED	133	No data received for the corresponding unqueued data element since system start or partition restart.
[rte_sws_7655] RTE_E_UNCONNECTED	134	The port used for communication is not connected.

Symbolic name	Value	Comments
[rte_sws_2739] RTE_E_IN_EXCLUSIVE_AREA	135	The error is returned by a blocking API and indicates that the runnable could not enter a wait state, because one ExecutableEntity of the current task's call stack has entered or is running in an ExclusiveArea.
[rte_sws_2757] RTE_E_SEG_FAULT	136	The error can be returned by an RTE API, if the parameters contain a direct or indirect reference to memory that is not accessible from the callers partition.
Overlayed Errors These errors do not refer to the data returned with the API. They can be overlayed with other Application- or Immediate Infrastructure Errors.		
[rte_sws_2571] RTE_E_LOST_DATA	64	An API call for reading received data with event semantics indicates that some incoming data has been lost due to an overflow of the receive queue or due to an error of the underlying communication stack.
[rte_sws_2702] RTE_E_MAX_AGE_EXCEEDED	64	An API call for reading received data with data semantics indicates that the available data has exceeded the aliveTimeout limit. A COM signal outdated callback will result in this error.

Table 5.4: RTE Error and Status values

The underlying type for Std_ReturnType is defined as a uint8 for reasons of compatibility – it avoids RTEs from different vendors assuming a different size if an enum was the underlying type. Consequently, #define is used to declare the error values:

```
1 typedef uint8 Std_ReturnType;
2
3 #define RTE_E_OK 0U
```

[rte_sws_1269]⌈ The standard errors as defined in table 5.4 including `RTE_E_OK` shall be defined in the RTE Header File. ⌋*(RTE00051)*

[rte_sws_2575]⌈ Application Error Identifiers with exception of `RTE_E_INVALID` shall be defined in the Application Header File. ⌋*(RTE00124, RTE00167)*

[rte_sws_2576]⌈ The application errors shall have a symbolic name defined as follows:

```
#define RTE_E_<interface>_<error> <error value>U
```

where `<interface>` `PortInterface` and `<error>` `ApplicationError` are the interface and error names from the configuration.[5] ⌋*(RTE00123)*

An `Std_ReturnType` value can be directly compared (for equality) with the above pre-defined error identifiers.

[rte_sws_7143]⌈ The RTE generator shall generate symbolic name for application errors with equal `<interface>` name, `<error>` name and `<error value>` only once. ⌋*(RTE00165)*

5.5.2 Rte_Instance

The `Rte_Instance` data type defines the handle used to access instance specific information from the component data structure.

[rte_sws_1148]⌈ The underlying data type for an instance handle shall be a pointer to a *Component Data Structure*. ⌋*(RTE00011, RTE00051)*

The component data structure (see Section 5.4.2) is uniquely defined for a component type and therefore the data type for the instance handle is automatically unique for each component type.

The instance handle type is defined in the application header file rte_sws_1007.

To avoid long and complex type names within SW-C code the following requirement imposes a fixed name on the instance handle data type.

[rte_sws_1150]⌈ The name of the instance handle type shall be defined, using `typedef` as `Rte_Instance`. ⌋*(BSW00305)*

5.5.3 RTE Modes

An `Rte_ModeType` is used to hold the identifiers for the ModeDeclarations of a ModeDeclarationGroup.

[rte_sws_2627]⌈ For each `ModeDeclarationGroupPrototype`, used in the SW-C's ports, the *Application Types Header File* shall contain a type definition

[5] No additional capitalization is applied to the names.

```
#ifndef RTE_MODETYPE_<prefix><ModeDeclarationGroup>
#define RTE_MODETYPE_<prefix><ModeDeclarationGroup>
typedef <type> Rte_ModeType_<prefix><ModeDeclarationGroup>;
#endif
```

where <ModeDeclarationGroup> is the shortName of the ModeDeclarationGroup,

<prefix> is the optional prefix attribute defined by the IncludedModeDeclarationGroupSet referring the ModeDeclarationGroup and

<type> is the shortName of the ImplementationDataType which is mapped to the ModeDeclarationGroup by a ModeRequestTypeMap. ⌋*(RTE00144)*

Note: *ImplementationDataType*s are generated in the *RTE Types Header* file.

Note: The type definition specified in rte_sws_2627 is deprecated to avoid incompatible or duplicate type definitions. It is recommended to not use this type in software components anymore (see rte_sws_2628).

rte_sws_2738 guarantees that for each ModeDeclarationGroup, used in the SW-C's ports, there is a unique mapping to an ImplementationDataType.

For a *ModeDecarationGroup* of category "ALPHABETIC_ORDER", the value <n>U within the Rte_ModeType_<ModeDeclarationGroup> is reserved to express a transition between modes, where <n> is the number of modes declared within the group. For *ModeDecarationGroups* of category "EXPLICIT_ORDER", a transition between modes is represented by the explicitly specified onTransitionValue.

[rte_sws_2659]⌈ For each *ModeDeclarationGroup* of category "ALPHABETIC_ORDER", the *Application Types Header File* shall contain a definition

```
#ifndef RTE_TRANSITION_<prefix><ModeDeclarationGroup>
#define RTE_TRANSITION_<prefix><ModeDeclarationGroup> <n>U
#endif
```

where <ModeDeclarationGroup> is the *shortName* of the *ModeDeclarationGroup*,

<prefix> is the optional prefix attribute defined by the IncludedModeDeclarationGroupSet referring the ModeDeclarationGroup and

<n> is the number of modes declared within the group.[6] ⌋*(RTE00144)*

[rte_sws_3858]⌈ For each *ModeDeclarationGroup* of category "EXPLICIT_ORDER", the *Application Types Header File* shall contain a definition

```
#ifndef RTE_TRANSITION_<prefix><ModeDeclarationGroup>
#define RTE_TRANSITION_<prefix><ModeDeclarationGroup> \
            <onTransitionValue>U
#endif
```

[6]No additional capitalization is applied to the names.

where <ModeDeclarationGroup> is the *shortName* of the *ModeDeclarationGroup*,

<prefix> is the optional prefix attribute defined by the IncludedModeDeclarationGroupSet referring the ModeDeclarationGroup and

<onTransitionValue> is the *onTransitionValue* of the *ModeDeclarationGroup*. ⌋*(RTE00144)*

[rte_sws_7640]⌈ The RTE Generator shall reject configurations where a SW-C uses two ModeDeclarationGroups with the same name but different ModeDeclarations. ⌋*(RTE00144, RTE00018)*

The rational for rte_sws_7640 is to protect against conditions which would lead to rte_sws_2659and rte_sws_2627 to generate conflicting types or macro definitions.

[rte_sws_2568]⌈ For each mode of a *ModeDeclarationGroup* of category "ALPHABETIC_ORDER", the *Application Types Header File* shall contain a definition

```
#ifndef RTE_MODE_<prefix><ModeDeclarationGroup>_<ModeDeclaration>
#define RTE_MODE_<prefix><ModeDeclarationGroup>_<ModeDeclaration> \
            <index>U
#endif
```

where <ModeDeclarationGroup> is the short name of the *ModeDeclarationGroup*,

<prefix> is the optional prefix attribute defined by the IncludedModeDeclarationGroupSet referring the ModeDeclarationGroup

<ModeDeclaration> is the *shortName* of a *ModeDeclaration*, and <index> is the index of the *ModeDeclaration*s in alphabetic ordering (ASCII / ISO 8859-1 code in ascending order) of the *shortNames* within the *ModeDeclarationGroup*[7].
The lowest index shall be '0' and therefore the range of assigned values is 0..<n-1> where <n> is the number of modes declared within the group. ⌋*(RTE00144)*

[rte_sws_3859]⌈ For each mode of a *ModeDeclarationGroup* of category "EXPLICIT_ORDER", the *Application Types Header File* shall contain a definition

```
#ifndef RTE_MODE_<prefix><ModeDeclarationGroup>_<ModeDeclaration>
#define RTE_MODE_<prefix><ModeDeclarationGroup>_<ModeDeclaration> \
            <value>U
#endif
```

where <ModeDeclarationGroup> is the short name of the *ModeDeclarationGroup*,

<prefix> is the optional prefix attribute defined by the IncludedModeDeclarationGroupSet referring the ModeDeclarationGroup

<ModeDeclaration> is the *shortName* of a *ModeDeclaration*, and <value> is the *value* specified at the *ModeDeclaration*. ⌋*(RTE00144)*

[7]No additional capitalization is applied to the names.

5.5.4 Enumeration Data Types

Enumeration is not a plain primitive *ImplementationDataType*. Rather a range of integers can be used as a structural description. The mapping of integers on "labels" in the enumeration is actually modelled in the SwC-T with the semantics class *CompuMethod* of a *SwDataDefProps* [2]. Enumeration data types are modeled as *ImplementationDataType*s having a *SwDataDefProps* referencing a *CompuMethod* that contains only *CompuScales* with point ranges (i. e. lower and upper limit of a *CompuScale* are identical).

[rte_sws_3809]⌈ The *Application Types Header File* shall include the definitions of all enumeration constants of *ImplementationDataType*s and *ApplicationDataType*s for each *ImplementationDataType*/*ApplicationDataType*s used (referenced) by this software component. ⌋*(RTE00167)*

This requirement ensures the availability of *ImplementationDataType* enumeration constants for the internal use in AUTOSAR software components.

[rte_sws_3810]⌈ For each *CompuScale* which has a point range and is located in the *compuInternalToPhys* container of a *CompuMethod* referenced by an *ImplementationDataType* or *ApplicationPrimitiveDataType* according rte_sws_3809 with *category* "`TEXTTABLE`", "`SCALE_LINEAR_AND_TEXTTABLE`", or "`SCALE_RATIONAL_AND_TEXTTABLE`", the *Application Types Header File* file shall contain a definition

```
#ifndef <prefix><EnumLiteral>
#define <prefix><EnumLiteral> <value><suffix>
#endif /* <prefix><EnumLiteral> */
```

where the name of the enumeration literal `<EnumLiteral>` is derived according to the following rule:

if (attribute `symbol` of `CompuScale` is available and not empty) {
  `<EnumLiteral>` **:=** C identifier specified in `symbol` attribute of `CompuScale`
} **else** {
  if (string specified in the `VT` element of the `CompuConst` of the `CompuScale`
    is a valid C identifier) {
    `<EnumLiteral>` **:=**
      string specified in the `VT` element of the `CompuConst` of the `CompuScale`
  } **else** {
    if (attribute `shortLabel` of `CompuScale` is available and not empty) {
      `<EnumLiteral>` **:=**
        string specified in `shortLabel` attribute of `CompuScale`
    }
  }
}

`<prefix>` is the optional *literalPrefix* attribute defined by the *IncludedDataTypeSet* referring the *AutosarDataType* using the *CompuMethod*. `<value>` is the value repre-

senting the *CompuScale*'s point range. <suffix> shall be "U" for unsigned data types and empty for signed data types. ⌋*(RTE00167)*

Please note that the prefix can either be defined that the *IncludedDataTypeSet* with a *literalPrefix* attribute references the *ApplicationDataType* or it references the *ImplementationDataType*.

rte_sws_3810 implies that the RTE does add prefix to the names of the enumeration constants on explicit demand only. This is necessary in order to handle enumeration constants supplied by Basic Software modules which all use their own prefix convention. Such Enumeration constant names have to be unique in the whole AUTOSAR system.

[rte_sws_8401]⌈ In the case that the same ImplementationDataType or ApplicationPrimitiveDataType is referenced via different IncludedDataTypeSets with different literalPrefix attributes, the definition according to rte_sws_3810 has to be provided once for each different literalPrefix. ⌋*(RTE00167)*

[rte_sws_3851]⌈ If the input of the RTE generator contains a CompuMethod with category "TEXTTABLE", "SCALE_LINEAR_AND_TEXTTABLE", or "SCALE_RATIONAL_AND_TEXTTABLE" that contains a CompuScale with a point range, and

- neither the attribute symbol of the CompuScale is available and not empty,
- nor the string specified in the VT element of the CompuConst of the CompuScale is a valid C identifier,
- nor the attribute shortLabel of CompuScale is available and not empty,

the RTE generator shall reject this input as an invalid configuration. ⌋*(RTE00018)*

[rte_sws_3813]⌈ The RTE shall reject configurations where the same software component type uses ImplementationDataTypes and ApplicationPrimitiveDataTypes referencing two or more CompuMethods with category "TEXTTABLE", "SCALE_LINEAR_AND_TEXTTABLE", or "SCALE_RATIONAL_AND_TEXTTABLE" that both contain a CompuScale with a different point range and an identical enumeration literal name as an invalid configuration. The only exception is that the usage of the ImplementationDataTypes are defined with non identical <literalPrefix>es. ⌋*(RTE00018)*

[rte_sws_7175]⌈ The RTE shall reject configurations where an ImplementationDataType or an ApplicationPrimitiveDataTypereferences a CompuMethod which is of category "TEXTTABLE", "SCALE_LINEAR_AND_TEXTTABLE", or "SCALE_RATIONAL_AND_TEXTTABLE" and has CompuScales with identical enumeration literal names. ⌋*(RTE00018)*

Note that there might exist additional CompuScales with non-point ranges inside a CompuMethod of category "TEXTTABLE", "SCALE_LINEAR_AND_TEXTTABLE", or "SCALE_RATIONAL_AND_TEXTTABLE" , but for those no enumeration literals are generated by the RTE generator.

5.5.5 Range Data Types

For the ApplicationPrimitiveDataType a Range might be specified by referencing a data constraint (dataConstr) giving the lowerLimit and the upperLimit. To allow a Software Component the access to these values two definitions for these values shall be generated.

[rte_sws_5051]⌈ The *Application Types Header File* shall include the definitions of all lowerLimit and upperLimit constants of each ApplicationPrimitiveDataType used by this software component once per ApplicationPrimitiveDataType if the ApplicationPrimitiveDataType is not referenced via different IncludedDataTypeSets. ⌋*(RTE00167)*

[rte_sws_8402]⌈ The Application Types Header File shall include the definitions of all lowerLimit and upperLimit constants of each ApplicationPrimitiveDataType used by this software component for each combination of different literalPrefix and ApplicationPrimitiveDataType when the same ImplementationDataType or ApplicationPrimitiveDataType is referenced via different IncludedDataTypeSets. ⌋*(RTE00167)*

[rte_sws_5052]⌈ The lowerLimit and upperLimit constants for *ApplicationPrimitiveDataType* referencing an *dataConstr* shall be generated by RTE generator in the *Application Type Header File* as:

```
1 #define <prefix><DataType>_LowerLimit <lowerValue><suffix>
2 #define <prefix><DataType>_UpperLimit <upperValue><suffix>
```

where <DataType> is the name of the ApplicationPrimitiveDataType used by the software component.

<ImplType> is the name of the ImplementationDataType on which the ApplicationPrimitiveDataType is mapped.

<prefix> is the optional literalPrefix attribute defined by the IncludedDataTypeSet referring the AutosarDataType to which the the *dataConstr* belongs.

<lowerValue> and <upperValue> are the values lowerLimit and upperLimit of the dataConstr referenced by the *ApplicationPrimitiveDataType*. The values in the macro definitions shall always reflect the closed interval, regardless of the interval type specified by the dataConstr.

<suffix> shall be "U" for unsigned data types and empty for signed data types. ⌋*(RTE00167)*

Please note that rte_sws_7196 is not applicable for rte_sws_5052. Further on it's possible that a DataPrototype using an *ApplicationPrimitiveDataType* might reference additional *dataConstr* (see rte_sws_7196). In this case the *upperLimit* and *lowerLimit* definitions according rte_sws_5052 do not reflect the real applicable range of the DataPrototype. No macros are generated for DataPrototype specific data constraints.

Please note that the prefix can either be defined that the IncludedDataTypeSet with a literalPrefix attribute references the ApplicationDataType or it references the ImplementationDataType.

[rte_sws_8403]⌈ For AUTOSAR data types which have an invalidValue specified, the AUTOSAR Types header file shall contain a definition

```
#define InvalidValue_<DataType> <invalidValue><suffix>
```

where <DataType> is the short name of the data type.

<invalidValue> is the value defined as invalidValue for the data type.

<suffix> shall be "U" for unsigned data types and empty for signed data types. ⌋*()*

5.6 API Reference

The functions described in this section are organized by the RTE API mapping name used by C and C++ AUTOSAR software-components to access the API. The API mapping hides from the AUTOSAR software-component programmer any need to be aware of the steps taken by the RTE generator to ensure that the generated API functions have unique names.

The instance handle as the first parameter of the API calls is marked as an optional parameter in this section. If an AUTOSAR software-component supports multiple instantiation, the instance handle shall be passed rte_sws_1013.

Note that rte_sws_3806 requires that the instance handle parameter does not exist if the AUTOSAR software-component does not support multiple instantiation.

5.6.1 Rte_Ports

Purpose: Provide an array of the ports of a given interface type and a given provide / require usage that can be accessed by the indirect API.

Signature: **[rte_sws_2619]**⌈

```
Rte_PortHandle_<i>_<R/P>
Rte_Ports_<i>_<R/P>([IN Rte_Instance])
```

Where here <i> is the port interface name and ‘P’ or ‘R’ are literals to indicate provide or require ports respectively. ⌋*(RTE00051)*

Existence: **[rte_sws_2613]**⌈ An Rte_Ports API shall be created for each interface type and usage by a port when the indirectAPI attribute of that port is set to true. ⌋*(RTE00051)*

Description: The `Rte_Ports` API provides access to an array of ports for the port oriented API.

[rte_sws_3602]⌈ Only those ports for which the indirect API was generated shall be contained in the array of ports. ⌋*(RTE00051)*

Return Value: Array of port data structures of the corresponding interface type and usage.

Notes: None.

5.6.2 Rte_NPorts

Purpose: Provide the number of ports of a given interface type and provide / require usage that can be accessed through the indirect API.

Signature: **[rte_sws_2614]**⌈
```
uint8
Rte_NPorts_<i>_<R/P>([IN Rte_Instance])
```
Where here `<i>` is the port interface name and 'P' or 'R' are literals to indicate provide or require ports respectively. ⌋*(RTE00051)*

Existence: **[rte_sws_2615]**⌈ An `Rte_NPorts` API shall be created for each interface type and usage by a port when the `indirectAPI` attribute of the port is set to true. ⌋*(RTE00051)*

Description: The `Rte_NPorts` API supports access to an array of ports for the port oriented API.

[rte_sws_3603]⌈ The `Rte_NPorts` shall return only the number of ports of a given interface and provide / require usage for which the indirect API was generated. ⌋*(RTE00051)*

Return Value: Number of port data structures of the corresponding interface type and usage.

Notes: None.

5.6.3 Rte_Port

Purpose: Provide access to the port data structure for a single port of a particular software component instance. This allows a software component to extract a sub-group of ports characterized by the same interface in order to iterate over this sub-group.

Signature: **[rte_sws_1354]**⌈
```
Rte_PortHandle_<i>_<R/P>
Rte_Port_<p>([IN Rte_Instance])
```

where <i> is the port interface name and <p> is the name of the port. ⌋*(RTE00051)*

Existence: **[rte_sws_1355]**⌈ An Rte_Port API shall be created for each port of an AUTOSAR SW-C, for which the indirectAPI attribute is set to true. ⌋*(RTE00051)*

Description: The Rte_Port API provides a pointer to a single port data structure, in order to support the indirect API.

Return Value: Pointer to port data structure for the appropriate port.

Notes: None.

5.6.4 Rte_Write

Purpose: Initiate an "explicit" sender-receiver transmission of data elements with "data" semantic (swImplPolicy different from 'queued').

Signature: **[rte_sws_1071]**⌈

```
Std_ReturnType
Rte_Write_<p>_<o>([IN Rte_Instance <instance>],
                  IN <data>)
```

Where <p> is the port name and <o> the VariableDataPrototype within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00098, RTE00028, RTE00131)*

Existence: **[rte_sws_1280]**⌈ The presence of a VariableAccess in the dataSendPoint role for a provided VariableDataPrototype with data semantics shall result in the generation of an Rte_Write API for the provided VariableDataPrototype. ⌋*(RTE00051)*

[rte_sws_ext_7818] The Rte_Write APIs may only be used by the runnable that contains the corresponding VariableAccess in the dataSendPoint role

Description: The Rte_Write API call initiates a sender-receiver communication where the transmission occurs at the point the API call is made (cf. explicit transmission).

The Rte_Write API call includes the IN parameter <data> to pass the data element to write.

The IN parameter <data> is passed by value or reference according to the ImplementationDataType as described in the section 5.2.6.5.

If the IN parameter <data> is passed by reference, the pointer must remain valid until the API call returns.

The RTE generator shall take into account the kind of connected require port which might not be just a variable but also a NV data. The table 4.6 gives an overview of compatibility rules.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the `Rte_Write`.

- **[rte_sws_7820]**⌈ `RTE_E_OK` – data passed to communication service successfully. ⌋*(RTE00094)*
- **[rte_sws_7822]**⌈ `RTE_E_COM_STOPPED` – the RTE could not perform the operation because the COM service is currently not available (inter ECU communication only). RTE shall return `RTE_E_COM_STOPPED` when the corresponding COM service returns `COM_SERVICE_NOT_AVAILABLE`. ⌋*(RTE00094)*
- **[rte_sws_2756]**⌈ `RTE_E_SEG_FAULT` – a segmentation violation is detected in the handed over parameters to the RTE API as required in rte_sws_2752 and rte_sws_2753. No transmission is executed. ⌋*(RTE00210)*

Notes: The `Rte_Write` call is used to transmit “data” (`swImplPolicy` not queued).

[rte_sws_7824]⌈ In case of inter ECU communication, the `Rte_Write` shall cause an immediate transmission request. ⌋*(RTE00028, RTE00131)*

Note that depending on the configuration a transmission request may not result in an actual transmission, for example transmission may be rate limited (time-based filtering) and thus dependent on other factors than API calls.

[rte_sws_7826]⌈ In case of inter ECU communication, the `Rte_Write` API shall return when the signal has been passed to the communication service for transmission. ⌋*(RTE00028, RTE00131)*

Depending on the communication server the transmission may or may not have been acknowledged by the receiver at the point the API call returns.

[rte_sws_2635]⌈ In case of intra ECU communication, the `Rte_Write` API call shall return after copying the data to RTE local memory or using IOC buffers. ⌋*(RTE00028, RTE00131)*

[rte_sws_1080]⌈ If the transmission acknowledgement is enabled, the RTE shall notify component when the transmission is acknowledged or a transmission error occurs. ⌋*(RTE00122)*

[rte_sws_1082]⌈ If a provide port typed by a sender-receiver interface has multiple require ports connected (i.e. it has multiple re-

ceivers), then the RTE shall ensure that writes to all receivers are independent. ⌋*(RTE00028)*

Requirement rte_sws_1082 ensures that an error detected by the RTE when writing to one receiver, e.g. communication is stopped, does not prevent the transmission of this message to other components.

5.6.5 Rte_Send

Purpose: Initiate an "explicit" sender-receiver transmission of data elements with "event" semantic (`swImplPolicy` equal to 'queued').

Signature: **[rte_sws_1072]**⌈

```
Std_ReturnType
Rte_Send_<p>_<o>([IN Rte_Instance <instance>],
                 IN <data>,
                 [IN uint16 <length>])
```

Where `<p>` is the port name and `<o>` the `VariableDataPrototype` within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00141, RTE00028, RTE00131)*

Existence: **[rte_sws_1281]**⌈ The presence of a `VariableAccess` in the `dataSendPoint` role for a provided `VariableDataPrototype` with event semantics shall result in the generation of an `Rte_Send` API for the provided `VariableDataPrototype`. ⌋*(RTE00051)*

[rte_sws_7813]⌈ The optional IN parameter `<length>` of the `Rte_Send` API shall be generated if the `VariableDataPrototype` is of type dynamic. ⌋*(RTE00190)*

[rte_sws_ext_7819] The `Rte_Send` APIs may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataSendPoint` role

Description: The `Rte_Send` API call initiates a sender-receiver communication where the transmission occurs at the point the API call is made (cf. explicit transmission).

The `Rte_Send` API call includes the IN parameter `<data>` to pass the data element to send.

The IN parameter `<data>` is passed by value or reference according to the `ImplementationDataType` as described in the section 5.2.6.5.

If the IN parameter `<data>` is passed by reference, the pointer must remain valid until the API call returns.

If the VariableDataPrototype is of type dynamic, the Rte_Send API call includes the IN parameter <length> to pass the number of elements in the data element to send.

The RTE generator shall take into account the kind of connected require port which might not be just a variable but also a NV data. The table 4.6 gives an overview of compatibility rules.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the Rte_Send.

- **[rte_sws_7821]**⌈ RTE_E_OK – data passed to communication service successfully. ⌋*(RTE00094)*
- **[rte_sws_7823]**⌈ RTE_E_COM_STOPPED – the RTE could not perform the operation because the COM service is currently not available (inter ECU communication only). RTE shall return RTE_E_COM_STOPPED when the corresponding COM service returns COM_SERVICE_NOT_AVAILABLE. ⌋*(RTE00094)*
- **[rte_sws_2634]**⌈ RTE_E_LIMIT – an 'event' has been discarded due to a full queue by one of the ECU local receivers (intra ECU communication only). ⌋*(RTE00143)*
- **[rte_sws_2754]**⌈ RTE_E_SEG_FAULT – a segmentation violation is detected in the handed over parameters to the RTE API as required in rte_sws_2752 and rte_sws_2753. No transmission is executed. ⌋*(RTE00210)*

Notes: The Rte_Send call is used to transmit "events" (swImplPolicy = queued).

[rte_sws_7825]⌈ In case of inter ECU communication, the Rte_Send shall cause an immediate transmission request. ⌋*(RTE00028, RTE00131)*

Note that depending on the configuration a transmission request may not result in an actual transmission, for example transmission may be rate limited (time-based filtering) and thus dependent on other factors than API calls.

[rte_sws_7827]⌈ In case of inter ECU communication, the Rte_Send API shall return when the signal has been passed to the communication service for transmission. ⌋*(RTE00028, RTE00131)*

Depending on the communication server the transmission may or may not have been acknowledged by the receiver at the point the API call returns.

[rte_sws_2633]⌈ In case of intra ECU communication, the Rte_Send API call shall return after attempting to enqueue the data in the IOC or RTE internal queues. ⌋*(RTE00028, RTE00131)*

If the transmission acknowledgement is enabled, the RTE shall notify component when the transmission is acknowledged or a transmission error occurs. rte_sws_1080

If a provide port typed by a sender-receiver interface has multiple require ports connected (i.e. it has multiple receivers), then the RTE shall ensure that writes to all receivers are independent. rte_sws_1082

Requirement rte_sws_1082 ensures that an error detected by the RTE when writing to one receiver, e.g. an overflow in one component's queue, does not prevent the transmission of this message to other components.

5.6.6 Rte_Switch

Purpose: Initiate a mode switch. The `Rte_Switch` API call is used for 'explicit' sending of a `mode switch notification`.

Signature: **[rte_sws_2631]**⌈

```
Std_ReturnType
Rte_Switch_<p>_<o>([IN Rte_Instance <instance>],
                   IN <mode>)
```

Where `<p>` is the port name and `<o>` the *ModeDeclarationGroupPrototype* within the `ModeSwitchInterface` categorizing the port. ⌋*(BSW00310, RTE00143, RTE00028, RTE00131)*

Existence: **[rte_sws_2632]**⌈ The existence of a *ModeSwitchPoint* shall result in the generation of a `Rte_Switch` API. ⌋*(RTE00051)*

[rte_sws_ext_2681] The `Rte_Switch` API may only be used by the runnable that contains the corresponding *ModeSwitchPoint*

Description: The `Rte_Switch` triggers a mode switch for all connected require `ModeDeclarationGroupPrototype`s.

The `Rte_Switch` API call includes exactly one IN parameter for the next mode `<mode>`. The IN parameter `<mode>` is passed by value according to the `ImplementationDataType` on which the *ModeDeclarationGroup* is mapped. The type name shall be equal to the shortName of the `ImplementationDataType`.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the `Rte_Switch` call.

- **[rte_sws_2674]**⌈ `RTE_E_OK` – data passed to service successfully. ⌋*(RTE00094)*

- **[rte_sws_2675]**⌈ `RTE_E_LIMIT` – a mode switch has been discarded by the receiving partition due to a full queue. ⌋*(RTE00143)*

Notes: `Rte_Switch` is restricted to ECU local communication.

If a mode instance is currently involved in a transition then the `Rte_Switch` API will attempt to queue the request and return rte_sws_2667. However if no transition is in progress for the mode instance, the mode disablings and the activations of OnEntry, OnTransition, and OnExit ExecutableEntities for this mode instance are executed before the `Rte_Switch` API returns rte_sws_2665.

Note that the mode switch might be discarded when the queue is full and a mode transition is in progress, see rte_sws_2675.

[rte_sws_2673]⌈ If the mode switched acknowledgment is enabled, the RTE shall notify the mode manager when the latest mode switch is completed in all receiving partitions. ⌋*(RTE00122)*

5.6.7 Rte_Invalidate

Purpose: Invalidate a data element for an “explicit” sender-receiver transmission.

Signature: **[rte_sws_1206]**⌈

```
Std_ReturnType
Rte_Invalidate_<p>_<o>([IN Rte_Instance <instance>])
```

Where `<p>` is the port name and `<o>` the `VariableDataPrototype` within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00078)*

Existence: **[rte_sws_1282]**⌈ An `Rte_Invalidate` API shall be created for any `VariableAccess` in the `dataSendPoint` role that references a provided `VariableDataPrototype` which associated `InvalidationPolicy` is set to `keep` or `replace`. ⌋*(RTE00051, RTE00078)*

[rte_sws_ext_2682] The `Rte_Invalidate` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataSendPoint` role

Description: The `Rte_Invalidate` API takes no parameters other than the instance handle – the return value is used to indicate the success, or otherwise, of the API call to the caller.

[rte_sws_1231]⌈ When COM is used for communication and the `VariableDataPrototype` is primitive the COM API function `Com_`-

InvalidateSignal shall be called for invalidation. ⌋*(RTE00019, RTE00078)*

[rte_sws_5063]⌈ When COM is used for communication and the VariableDataPrototype is composite the COM API function Com_InvalidateSignalGroup shall be called for invalidation. ⌋*(RTE00019, RTE00078)*

The behavior required when COM is not used for communication is described in Section 4.3.1.8.

Return Value: The return value is used to indicate the "OK" status or errors detected by the RTE during execution of the Rte_Invalidate call.

- **[rte_sws_1207]**⌈ RTE_E_OK – No error occurred. ⌋*(RTE00094)*
- **[rte_sws_1339]**⌈ RTE_E_COM_STOPPED – the RTE could not perform the operation because the COM service is currently not available (inter ECU communication only). RTE shall return RTE_E_COM_STOPPED when the corresponding COM service returns COM_SERVICE_NOT_AVAILABLE. ⌋*(RTE00094)*

Notes: The API name includes an identifier <p>_<o> that is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

The communication service configuration determines whether the signal receiver(s) receive an "invalid signal" notification or whether the invalidated signal is silently replaced by the signal's initial value.

5.6.8 Rte_Feedback

Purpose: Provide access to acknowledgement notifications for explicit sender-receiver communication and to pass error notification to senders.

Signature: **[rte_sws_1083]**⌈

```
Std_ReturnType
Rte_Feedback_<p>_<o>([IN Rte_Instance <instance>])
```

Where <p> is the port name and <o> the VariableDataPrototype within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00122)*

Existence: **[rte_sws_1283]**⌈ Acknowledgement is enabled for a provided VariableDataPrototype by the existence of a TransmissionAcknowledgementRequest in the SenderComSpec. ⌋*(RTE00051, RTE00122)*

[rte_sws_1284]⌈ A blocking Rte_Feedback API shall be generated for a provided VariableDataPrototype if acknowledgement

is enabled and a WaitPoint references a DataSendCompletedEvent that in turn references the VariableDataPrototype. ⌋*(RTE00051, RTE00122)*

[rte_sws_1285]⌈ A non-blocking Rte_Feedback API shall be generated for a provided VariableDataPrototype if acknowledgement is enabled and a VariableAccess in the dataSendPoint role references the VariableDataPrototype but no WaitPoint references the DataSendCompletedEvent that references the VariableDataPrototype. ⌋*(RTE00051, RTE00122)*

[rte_sws_1286]⌈ If acknowledgement is enabled for a provided VariableDataPrototype and a DataSendCompletedEvent references a runnable entity as well as the VariableDataPrototype, the runnable entity shall be activated when the transmission acknowledgement occurs or when a timeout was detected by the RTE. rte_sws_1137. ⌋*(RTE00051, RTE00122)*

Requirement rte_sws_1286 merely affects when the runnable is activated – an API call should still be created, according to requirement rte_sws_1285 to actually read the data.

[rte_sws_1287]⌈ A DataSendCompletedEvent that references a RunnableEntity and is referenced by a WaitPoint shall be an invalid configuration which is rejected by the RTE generator. ⌋*(RTE00051, RTE00122, RTE00018)*

[rte_sws_ext_2687] A blocking Rte_Feedback API may only be used by the runnable that contains the corresponding WaitPoint

[rte_sws_7634]⌈ A call to Rte_Feedback shall not change the status returned by Rte_Feedback. ⌋*(RTE00122)*

The Rte_Feedback API return value is only changed when a new transmission is requested (Rte_Send or Rte_Write) or when the notification from COM is received.

[rte_sws_7635]⌈ After a Rte_Send or Rte_Write transmission request, only the first notification from COM shall be taken into account for a given Signal or SignalGroup. ⌋*(RTE00122)*

rte_sws_7635 is needed in case of cyclic transmission which could result in multiple transmissions with different status.

Description: The Rte_Feedback API takes no parameters other than the instance handle – the return value is used to indicate the acknowledgement status to the caller.

The Rte_Feedback API applies only to explicit sender-receiver communication.

Return Value: The return value is used to indicate the status of the transmission and errors detected by the RTE.

- **[rte_sws_1084]**⌈ `RTE_E_NO_DATA` – No acknowledgments or error notifications were received from COM when the `Rte_Feedback` API was called (non-blocking call) or when the `WaitPoint` timeout expired (blocking call). ⌋*(RTE00094, RTE00122)*
- `RTE_E_COM_STOPPED` – returned in one of these cases:
  - **[rte_sws_7636]**⌈ (Inter-ECU communication only) The last transmission was rejected (when the `Rte_Send` or `Rte_Write` API was called), with an `RTE_E_COM_STOPPED` return code. ⌋*(RTE00094, RTE00122)*
  - **[rte_sws_3774]**⌈ (Inter-ECU communication only) An error notification from COM was received before any timeout notification. ⌋*(RTE00094, RTE00122)*
- **[rte_sws_7637]**⌈ `RTE_E_TIMEOUT` – (Inter-ECU and Inter-Partition only) A timeout notification was received from COM or IOC before any error notification. ⌋*(RTE00094, RTE00122)*
- **[rte_sws_1086]**⌈ `RTE_E_TRANSMIT_ACK` – In case of inter-ECU communication, a transmission acknowledgment was received from COM; or in case of intra-ECU communication, even if a queue overflow occurred. ⌋*(RTE00094, RTE00122)*
- **[rte_sws_7658]**⌈ `RTE_E_UNCONNECTED` – Indicates that the sender port is not connected. ⌋*(RTE00094, RTE00122, RTE00139)*
- **[rte_sws_2740]**⌈ `RTE_E_IN_EXCLUSIVE_AREA` – Used only for the blocking API. `RTE_E_IN_EXCLUSIVE_AREA` indicates that the runnable can not enter wait, as one of the `ExecutableEntity`s in the call stack of this task is currently in an exclusive area, see rte_sws_2739. - In a properly configured system, this error should not occur. ⌋*(RTE00092, RTE00046, RTE00032)*

The `RTE_E_TRANSMIT_ACK` and `RTE_E_UNCONNECTED` return values are not considered to be an error but rather indicates correct operation of the API call.

[rte_sws_7652]⌈ The initial return value of the `Rte_Feedback` API, before any attempt to write some data shall be `RTE_E_TRANSMIT_ACK`. ⌋*(RTE00094, RTE00122, RTE00128, RTE00185)*

Notes: If multiple transmissions on the same port/element are outstanding it is not possible to determine which is acknowledged first. If this is important, transmissions should be serialized with the next occurring only when the previous transmission has been acknowledged or has timed out.

A transmission acknowledgment (or error and timeout) notification is not always provided by COM (the bus or PDU Router may not support transmission acknowledgment for this PDU, or COM may not be configured to perform transmission deadline monitoring).

In case of a blocking `Rte_Feedback`, the `WaitPoint` timeout should be compatible with the timeout defined at the COM level.

Note that transmission acknowledgement is not supported in case of 1:n communication (see rte_sws_5506).

5.6.9 Rte_SwitchAck

Purpose: Provide access to mode switch completed acknowledgements and error notifications to `mode managers`.

Signature: **[rte_sws_2725]**⌈
`Std_ReturnType Rte_SwitchAck_<p>_<o>([IN Rte_Instance <instance>])`

Where `<p>` is the port name and `<o>` the `ModeDeclarationGroupPrototype` within the `ModeSwitchInterface` categorizing the port. ⌋*(BSW00310, RTE00122)*

Existence: **[rte_sws_2676]**⌈ Acknowledgement is enabled for a provided `ModeDeclarationGroupPrototype` by the existence of a `ModeSwitchedAckRequest` in the `ModeSwitchSenderComSpec`. ⌋*(RTE00051, RTE00122)*

[rte_sws_2677]⌈ A blocking `Rte_SwitchAck` API shall be generated for a provided `ModeDeclarationGroupPrototype` if acknowledgement is enabled and a `WaitPoint` references a `ModeSwitchedAckEvent` that in turn references the `ModeDeclarationGroupPrototype`. ⌋*(RTE00051, RTE00122)*

[rte_sws_2678]⌈ A non-blocking `Rte_SwitchAck` API shall be generated for a provided `ModeDeclarationGroupPrototype` if acknowledgement is enabled and a `ModeSwitchPoint` references the `ModeDeclarationGroupPrototype` but no `Mod-`

eSwitchedAckEvent references the ModeDeclarationGroup-Prototype. ⌋*(RTE00051, RTE00122)*

[rte_sws_ext_2726] A blocking Rte_SwitchAck API may only be used by the runnable that contains the corresponding WaitPoint

Description: The Rte_SwitchAck API takes no parameters other than the instance handle – the return value is used to indicate the acknowledgement status to the caller.

Return Value: The return value is used to indicate the status of a mode switch and errors detected by the RTE.

- **[rte_sws_2727]**⌈ RTE_E_NO_DATA – (non-blocking read) The mode switch is still in progress. ⌋*(RTE00094, RTE00122)*
- **[rte_sws_2728]**⌈ RTE_E_TIMEOUT – The configured time-out exceeds before the mode transition was completed. ⌋*(RTE00094, RTE00210)*
- **[rte_sws_3853]**⌈ RTE_E_TIMEOUT – The partition of the mode users is stopped or restarting or has been restarted while the mode switch was requested. ⌋*(RTE00094, RTE00210)*
- **[rte_sws_2729]**⌈ RTE_E_TRANSMIT_ACK – The mode switch has been completed (see rte_sws_2587). ⌋*(RTE00094, RTE00122)*
- **[rte_sws_7659]**⌈ RTE_E_UNCONNECTED – Indicates that the mode provider port is not connected. ⌋*(RTE00094, RTE00122, RTE00139)*
- **[rte_sws_2741]**⌈ RTE_E_IN_EXCLUSIVE_AREA – Used only for the blocking API. RTE_E_IN_EXCLUSIVE_AREA indicates that the runnable can not enter wait, as one of the Executable Entitys in the call stack of this task is currently in an exclusive area, see rte_sws_2739. - In a properly configured system, this error should not occur. ⌋*(RTE00092, RTE00046, RTE00032)*

The RTE_E_TRANSMIT_ACK return value is not considered to be an error but rather indicates correct operation of the API call.

When RTE_E_NO_DATA occurs, a component is free to reinvoke Rte_SwitchAck and thus repeat the attempt to read the feedback status.

Notes: If multiple mode switches of the same mode machine instance are outstanding, it is not possible to determine which is acknowledged first. If this is important, switches should be serialized with the next switch occurring only when the previous switch has been acknowledged. The queue length should be 1.

5.6.10 Rte_Read

Purpose: Performs an "explicit" read on a sender-receiver communication data element with "data" semantics (swImplPolicy != queued). By compatibility, the port may also have a `ParameterInterface` or a `NvDataInterface`. The `Rte_Read` API is used for explicit read by argument.

Signature: **[rte_sws_1091]**⌈

```
Std_ReturnType
Rte_Read_<p>_<o>([IN Rte_Instance <instance>],
                 OUT <data>))
```

Where `<p>` is the port name and `<o>` the `VariableDataPrototype` within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00141, RTE00028, RTE00131)*

Existence: **[rte_sws_1289]**⌈ A non-blocking `Rte_Read` API shall be generated if a `VariableAccess` in the `dataReceivePointByArgument` role references a required `VariableDataPrototype` with 'data' semantics. ⌋*(RTE00051)*

[rte_sws_7396]⌈ The RTE shall ensure that direct explicit read accesses will not deliver undefined data item values. In case there may be an explicit read access before the first data reception an initial value has to be provided as the result of this explicit read access. ⌋*(RTE00051, RTE00183)*

A `WaitPoint` cannot reference a `DataReceivedEvent` that in turn references a required `VariableDataPrototype` with 'data' semantics shall be considered an invalid configuration (see rte_sws_3018). Hence there are no blocking `Rte_Read` API.

[rte_sws_ext_2683] The `Rte_Read` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByArgument` role

[rte_sws_1313]⌈ A `DataReceivedEvent` that references a runnable entity and is referenced by a `WaitPoint` shall be an invalid configuration. ⌋*(RTE00051, RTE00018)*

The RTE generator shall take into account the kind of provide port which might not be just a variable but also a Parameter (fixed, const or standard), a standard sender (i.e. a variable) or a NV data. The table 4.6 gives an overview of compatibility rules.

Description: The `Rte_Read` API call includes the OUT parameter `<data>` to pass back the received data.

The pointer to the OUT parameter `<data>` must remain valid until the API call returns.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the `Rte_Read` API call or errors detected by the communication system.

- **[rte_sws_1093]**⌈ `RTE_E_OK` – data read successfully. ⌋*(RTE00094)*
- **[rte_sws_2626]**⌈ `RTE_E_INVALID` – `data element invalid`. ⌋*(RTE00078)*
- **[rte_sws_2703]**⌈ `RTE_E_MAX_AGE_EXCEEDED` – `data element outdated`. This `Overlayed Error` can be combined with any of the above error codes. ⌋*(RTE00147)*
- **[rte_sws_7643]**⌈ `RTE_E_NEVER_RECEIVED` – No data received since system start or partition restart. ⌋*(RTE00184, RTE00224)*

  [rte_sws_7690]⌈ `RTE_E_UNCONNECTED` – Indicates that the receiver port is not connected. ⌋*(RTE00094, RTE00139, RTE00200)*

Notes: The API name includes an identifier `<p>_<o>` that indicates the read access point name and is formed from the port and operation item names. See section 5.2.6.4 for details on the naming convention.

5.6.11 Rte_DRead

Purpose: Performs an "explicit" read on a sender-receiver communication data element with "data" semantics (swImplPolicy != queued). By compatibility, the port may also have a `ParameterInterface` or a `NvDataInterface`. The `Rte_DRead` API is used for explicit read by value.

Signature: **[rte_sws_7394]**⌈

```
<return>
Rte_DRead_<p>_<o>([IN Rte_Instance <instance>])
```

Where `<p>` is the port name and `<o>` the `VariableDataPrototype` within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00141, RTE00028, RTE00131, RTE00183)*

Existence: **[rte_sws_7395]**⌈ A non-blocking `Rte_DRead` API shall be generated if a `VariableAccess` in the `dataReceivePointByValue` role references a required `VariableDataPrototype` with 'data' semantics. This requirement is applicable only for primitive data types. ⌋*(RTE00051, RTE00183)*

The RTE shall ensure that direct explicit read accesses will not deliver undefined data item values. In case there may be an explicit

read access before the first data reception an initial value has to be provided as the result of this explicit read access. rte_sws_7396

A `WaitPoint` cannot reference a `DataReceivedEvent` that in turn references a required `VariableDataPrototype` with 'data' semantics shall be considered an invalid configuration (see rte_sws_3018). Hence there are no blocking `Rte_DRead` API.

[rte_sws_ext_7397] The `Rte_DRead` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByValue` role

A `DataReceivedEvent` that references a runnable entity and is referenced by a `WaitPoint` shall be an invalid configuration. rte_sws_1313

The RTE generator shall take into account the kind of provide port which might not be just a variable but also a Parameter (fixed, const or standard), a standard sender (i.e. a variable) or a NV data. The table 4.6 gives an overview of compatibility rules.

Description: The `Rte_DRead` API returns the received data as a return value.

Return Value: The `Rte_DRead` return value provide access to the data value of the `VariableDataPrototype`.

The return type of `Rte_DRead` is dependent on the `ImplementationDataType` of the `VariableDataPrototype`. Thus the component does not need to use type casting to convert access to the `VariableDataPrototype` data.

For details of the `<return>` value definition see section 5.2.6.6.

Please note that the `Rte_DRead` API only supports `VariableDataPrototype`s typed by a `Primitive Implementation Data Type` or `Redefinition Implementation Data Type` redefining a `Primitive Implementation Data Type`.

Notes: The API name includes an identifier `<p>_<o>` that indicates the read access point name and is formed from the port and operation item names. See section 5.2.6.4 for details on the naming convention.

5.6.12 Rte_Receive

Purpose: Performs an "explicit" read on a sender-receiver communication data element with "event" semantics (`swImplPolicy`=queued).

[rte_sws_1092]⌈

```
Std_ReturnType
Rte_Receive_<p>_<o>([IN Rte_Instance <instance>],
```

```
                                    OUT <data>,
                                    [OUT uint16 <length>])
```

Where <p> is the port name and <o> the data element within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00141, RTE00028, RTE00131)*

Existence:

[rte_sws_1288]⌈ A non-blocking Rte_Receive API shall be generated if a VariableAccess in the dataReceivePointByArgument role references a required VariableDataPrototype with 'event' semantics. ⌋*(RTE00051)*

[rte_sws_7638]⌈ The RTE Generator shall reject configurations were a VariableDataPrototype with 'event' semantics is referenced by a VariableAccess in the dataReceivePointByValue role. ⌋*(RTE00018)*

[rte_sws_7814]⌈ The optional OUT parameter <length> of the Rte_Receive API shall be generated if the VariableDataPrototype is of type dynamic. ⌋*(RTE00190)*

[rte_sws_1290]⌈ A blocking Rte_Receive API shall be generated if a VariableAccess in the dataReceivePointByArgument role references a required VariableDataPrototype with 'event' semantics that is, in turn, referenced by a DataReceivedEvent and the DataReceivedEvent is referenced by a WaitPoint. ⌋*(RTE00051)*

[rte_sws_ext_2684] The Rte_Receive API may only be used by the runnable that contains the corresponding VariableAccess in the dataReceivePointByArgument role

A DataReceivedEvent that references a runnable entity and is referenced by a WaitPoint shall be an invalid configuration. rte_sws_1313

Description:

The Rte_Receive API call includes the OUT parameter <data> to pass back the received data element.

If the VariableDataPrototype is of type dynamic, the Rte_Receive API call include the OUT parameter <length> to pass back the number of elements in the received data element.

The pointers to the OUT parameters must remain valid until the API call returns.

[rte_sws_7673]⌈ In case return value is RTE_E_NO_DATA, RTE_E_TIMEOUT, RTE_E_UNCONNECTED or RTE_E_IN_EXCLUSIVE_AREA, the OUT parameters shall remain unchanged. ⌋*(RTE00094, RTE00141)*

Return Value: The return value is used to indicate errors detected by the RTE during execution of the `Rte_Receive` API call or errors detected by the communication system.

- **[rte_sws_2598]**⌈ `RTE_E_OK` – data read successfully. ⌋*(RTE00094)*
- **[rte_sws_1094]**⌈ `RTE_E_NO_DATA` – (explicit non-blocking read) no events were received and no other error occurred when the read was attempted. ⌋*(RTE00094)*
- **[rte_sws_1095]**⌈ `RTE_E_TIMEOUT` – (explicit blocking read) no events were received and no other error occurred when the read was attempted. ⌋*(RTE00094, RTE00069)*
- **[rte_sws_2572]**⌈ `RTE_E_LOST_DATA` – Indicates that some incoming data has been lost due to an overflow of the receive queue or due to an error of the underlying communication layers. This is not an error of the data returned in the parameters. This `Overlayed Error` can be combined with any of the above. ⌋*(RTE00107, RTE00110, RTE00094)*
- **[rte_sws_7665]**⌈ `RTE_E_UNCONNECTED` – Indicates that the receiver port is not connected. ⌋*(RTE00107, RTE00110, RTE00094, RTE00139, RTE00200)*

  Unlike `RTE_E_NO_DATA`, there is no need to retry receiving an event in this case.
- **[rte_sws_2743]**⌈ `RTE_E_IN_EXCLUSIVE_AREA` – Used only for the blocking API. `RTE_E_IN_EXCLUSIVE_AREA` indicates that the runnable can not enter wait, as one of the `ExecutableEntity`s in the call stack of this task is currently in an exclusive area, see rte_sws_2739. - In a properly configured system, this error should not occur. ⌋*(RTE00092, RTE00046, RTE00032)*

The `RTE_E_NO_DATA`, `RTE_E_TIMEOUT` and `RTE_E_UNCONNECTED` return values are not considered to be errors but rather indicate correct operation of the API call.

Notes: The API name includes an identifier `<p>_<o>` that indicates the read access point name and is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

5.6.13 Rte_Call

Purpose: Initiate a client-server communication.

Signature: **[rte_sws_1102]**⌈
```
Std_ReturnType
Rte_Call_<p>_<o>([IN Rte_Instance <instance>],
                 [IN|IN/OUT|OUT] <data_1>...
                 [IN|IN/OUT|OUT] <data_n>)
```
Where <p> is the port name and <o> the operation within the client-server interface categorizing the port. ⌋*(BSW00310, RTE00029)*

Existence: **[rte_sws_1293]**⌈ A synchronous Rte_Call API shall be generated if a SynchronousServerCallPoint references a required ClientServerOperation. ⌋*(RTE00051, RTE00111)*

[rte_sws_1294]⌈ An asynchronous Rte_Call API shall be generated if an AsynchronousServerCallPoint references a required ClientServerOperation. ⌋*(RTE00051, RTE00111)*

A configuration that includes both synchronous and asynchronous ServerCallPoints for a given ClientServerOperation is invalid (rte_sws_3014).

[rte_sws_ext_2685] The Rte_Call API may only be used by the runnable that contains the corresponding ServerCallPoint

Description: Client function to initiate client-server communication. The Rte_Call API is used for both synchronous and asynchronous calls.

The Rte_Call API includes zero or more IN, IN/OUT and OUT parameters.

[rte_sws_6639]⌈ IN/OUT parameters are passed by value when they are "Primitive Implementation Data Type"s and the call is asynchronous. ⌋*(RTE00051, RTE00111)*

Rational: In case of an asynchronous call, the IN/OUT parameters are only IN parameters.

The IN, IN/OUT and OUT parameters are passed by value or reference according to the ImplementationDataType as described in the section 5.2.6.5.

The pointers to all parameters passed by reference must remain valid until the API call returns.

Return Value: **[rte_sws_1103]**⌈ The return value shall be used to indicate infrastructure errors detected by the RTE during execution of the Rte_Call call and, for synchronous communication, infrastructure and application errors during execution of the server. ⌋*(RTE00094, RTE00123, RTE00124)*

- **[rte_sws_1104]**⌈ RTE_E_OK – The API call completed successfully. ⌋*(RTE00094)*

- **[rte_sws_1105]**⌈ RTE_E_LIMIT – The client has multiple outstanding asynchronous client-server invocations of the same operation in the same port. The server invocation shall be discarded, the buffers of the return parameters shall not be modified (see also rte_sws_2658). ⌋*(RTE00094, RTE00079)*
- **[rte_sws_1106]**⌈ RTE_E_COM_STOPPED – the RTE could not perform the operation because the COM service is currently not available (inter ECU communication only). RTE shall return RTE_E_COM_STOPPED when the corresponding COM service returns COM_SERVICE_NOT_AVAILABLE. The buffers of the return parameters shall not be modified. ⌋*(RTE00094)*
- **[rte_sws_1107]**⌈ RTE_E_TIMEOUT – (synchronous inter-task and inter-ECU only) No reply was received within the configured timeout. The buffers of the return parameters shall not be modified. ⌋*(RTE00094, RTE00069)*
- **[rte_sws_7656]**⌈ RTE_E_UNCONNECTED – Indicates that the client port is not connected. ⌋*(RTE00094, RTE00139, RTE00200)*
- **[rte_sws_2744]**⌈ RTE_E_IN_EXCLUSIVE_AREA – Used only for the synchronous call to a remote server on a remote core or remote ECU. RTE_E_IN_EXCLUSIVE_AREA indicates that the runnable can not enter wait, as one of the ExecutableEntitys in the call stack of this task is currently in an exclusive area, see rte_sws_2739. - In a properly configured system, this error should not occur. ⌋*(RTE00092, RTE00046, RTE00032)*
- **[rte_sws_2755]**⌈ RTE_E_SEG_FAULT – a segmentation violation is detected in the handed over parameters to the RTE API as required in rte_sws_2752 and rte_sws_2753. No transmission is executed. ⌋*(RTE00210)*
- **[rte_sws_2577]**⌈ The application error (synchronous client-server) from a server shall only be returned if none of the above infrastructure errors (other than RTE_E_OK) have occurred. ⌋*(RTE00123)*

Note that the RTE_E_OK return value indicates that the Rte_Call API call completed successfully. In case of a synchronous client server call it also indicates successful processing of the request by the server.

An asynchronous server invocation is considered to be outstanding until either the client retrieved the result successfully, a timeout was detected by the RTE in inter-ECU and inter-partition communication or the server runnable has terminated after a timeout was detected in intra-ECU communication.

When the RTE_E_TIMEOUT error occurs, RTE shall discard any subsequent responses to that request, (see rte_sws_2657).

Notes: **[rte_sws_1109]**⌈ The interface operation's OUT parameters shall be omitted for an *asynchronous* call. ⌋*(RTE00029, RTE00079)*

In case of asynchronous communication:

- the Rte_Call only includes IN and IN/OUT parameters.
- the Rte_Result only includes IN/OUT and OUT parameters to collect the result of the server call.
- the IN/OUT parameters provided during the Rte_Call can be a different addresse than the IN/OUT parameter passed during the Rte_Result.

5.6.14 Rte_Result

Purpose: Get the result of an asynchronous client-server call.

Signature: **[rte_sws_1111]**⌈

```
Std_ReturnType
Rte_Result_<p>_<o>([IN Rte_Instance <instance>],
                   [IN/OUT|OUT <param 1>]...
                   [IN/OUT|OUT <param n>])
```

Where <p> is the port name and <o> the operation within the client-server interface categorizing the port. ⌋*(BSW00310)*

The signature can include zero or more IN/OUT and OUT parameters depending on the signature of the operation in the client-server interface.

Existence: **[rte_sws_1296]**⌈ A non-blocking Rte_Result API shall be generated if an AsynchronousServerCallReturnsEvent references a required ClientServerOperation and no WaitPoint references the AsynchronousServerCallReturnsEvent. ⌋*(RTE00051)*

[rte_sws_1297]⌈ A blocking Rte_Result API shall be generated if an AsynchronousServerCallReturnsEvent references a required ClientServerOperation and a WaitPoint references the AsynchronousServerCallReturnsEvent. ⌋*(RTE00051)*

[rte_sws_ext_2686] The blocking Rte_Result API may only be used by the runnable that contains the corresponding WaitPoint

[rte_sws_1298]⌈ If an AsynchronousServerCallReturnsEvent references a RunnableEntity and a required ClientServerOperation, the RunnableEntity shall be

activated when the operation's result is available or when a timeout was detected by the RTE rte_sws_1133. ⌋*(RTE00051)*

Requirement rte_sws_1298 merely affects when the runnable is activated – an API call should still be created to actually read the reply based on requirement rte_sws_1296.

[rte_sws_1312]⌈ An `AsynchronousServerCallReturnsEvent` that references a runnable entity and is referenced by a `WaitPoint` is invalid. ⌋*(RTE00051)*

Description: The `Rte_Result` API is used by a client to collect the result of an *asynchronous* client-server communication.

The `Rte_Result` API includes zero or more IN/OUT and OUT parameters to pass back results.

The pointers to all parameters passed by reference must remain valid until the API call returns.

Return Value: The return value is used to indicate errors from either the `Rte_Result` call itself or communication errors detected before the API call was made.

- **[rte_sws_1112]**⌈ `RTE_E_OK` – The API call completed successfully. ⌋*(RTE00094)*
- **[rte_sws_1113]**⌈ `RTE_E_NO_DATA` – (non-blocking read) The server's result is not available but no other error occurred within the API call or the server was not called since `Rte_Start` or the restart of the Partition. The buffers for the IN/OUT and OUT parameters shall not be modified. ⌋*(RTE00094)*
- **[rte_sws_8301]**⌈ `RTE_E_NO_DATA` – (non-blocking read) The previous `Rte_Call` returned an `RTE_E_SEG_FAULT`. ⌋*(RTE00094)*
- **[rte_sws_1114]**⌈ `RTE_E_TIMEOUT` – The server's result is not available within the specified timeout but no other error occurred within the API call. The buffers for the IN/OUT and OUT parameters shall not be modified. ⌋*(RTE00094, RTE00069)*
- **[rte_sws_3606]**⌈ `RTE_E_COM_STOPPED` – the RTE could not perform the operation because the COM service is currently not available (inter ECU communication only). RTE shall return `RTE_E_COM_STOPPED` when the corresponding COM service returns `COM_SERVICE_NOT_AVAILABLE`. The server's result has *not* been successfully retrieved from the communication service. The buffers of the return parameters shall not be modified. ⌋*(RTE00094)*

- **[rte_sws_7657]**⌈ `RTE_E_UNCONNECTED` – Indicates that the client port is not connected. ⌋*(RTE00094, RTE00139, RTE00200)*
- **[rte_sws_2745]**⌈ `RTE_E_IN_EXCLUSIVE_AREA` – Used only for the blocking API. `RTE_E_IN_EXCLUSIVE_AREA` indicates that the runnable can not enter wait, as one of the `ExecutableEntity`s in the call stack of this task is currently in an exclusive area, see rte_sws_2739. - In a properly configured system, this error should not occur. ⌋*(RTE00092, RTE00046, RTE00032)*

  [rte_sws_2746]⌈ `Rte_Result` shall not return `RTE_E_IN_EXCLUSIVE_AREA`, if the wait is resolved by a mapping of the server runnable to a task with higher priority on the same core. ⌋*(RTE00092, RTE00046, RTE00032)*
- **[rte_sws_8302]**⌈ `RTE_E_SEG_FAULT` – a segmentation violation is detected in the handed over parameters to the RTE API as required in rte_sws_2752 and rte_sws_2753. No transmission is executed. ⌋*(RTE00094)*
- **[rte_sws_2578]**⌈ Application Errors – The error code of the server shall only be returned, if none of the above infrastructure errors or indications have occurred. ⌋*(RTE00094, RTE00123)*

The `RTE_E_NO_DATA`, `RTE_E_TIMEOUT`, and `RTE_E_UNCONNECTED` return values are not considered to be errors but rather indicate correct operation of the API call.

When the `RTE_E_TIMEOUT` error occurs, RTE shall discard any subsequent responses to that request, (see rte_sws_2657).

When `RTE_E_NO_DATA` occurs, a component is free to invoke `Rte_Result` again and thus repeat the attempt to read the server's result.

Notes: The API name includes an identifier `<p>_<o>` that indicates the read access point name and is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

5.6.15 Rte_Pim

Purpose: Provide access to the defined per-instance memory (section) of a software component.

Signature: **[rte_sws_1118]**⌈

```
<type>/<return reference>
Rte_Pim_<name>([IN Rte_Instance <instance>])
```

Where <name> is the (short) name of the per-instance name. ⌋*(BSW00310, RTE00075)*

Existence: **[rte_sws_1299]**⌈ An Rte_Pim API shall be created for each defined PerInstanceMemory or arTypedPerInstanceMemory within the AUTOSAR software-component (description). ⌋*(RTE00051)*

Description: The Rte_Pim API provides access to the per-instance memory (section) defined in the context of a SwcInternalBehavior of a software-component description.

Return Value: **[rte_sws_1119]**⌈ The API returns a typed reference (in C a typed pointer) to the per-instance memory. ⌋*(RTE00051, RTE00075)*

Notes: For a 'C' typed PerInstanceMemory, the name of the return type <type> shall be defined in the type attribute of the PerInstanceMemory. The type itself is defined using the typeDefinition attribute of the PerInstanceMemory. It is assumed that this attribute contains a string that represents a C type definition (typedef) in valid C syntax (see rte_sws_2304 and rte_sws_7133). For an arTypedPerInstanceMemory the <return reference> is defined by the associated AutosarDataType (see rte_sws_7161). For details of the <return reference> definition see section 5.2.6.7.

5.6.16 Rte_CData

Purpose: Provide access to the calibration parameter an AUTOSAR software-component defined internally. The ParameterDataPrototype in the role perInstanceParameter or sharedParameter is used to define software component internal calibration parameters. Internal because the ParameterDataPrototype cannot be reused outside the software-component. Access is read-only. It can be configured for each calibration parameter individually if it is shared by all instances of an AUTOSAR software-component or if each instance has an own data value associated with it.

Signature: **[rte_sws_1252]**⌈

```
<return>
Rte_CData_<name>([IN Rte_Instance <instance>])
```

Where <name> is the calibration parameter name. ⌋*(BSW00310, RTE00155)*

Existence: **[rte_sws_1300]**⌈ An Rte_CData API shall be generated if a ParameterAccess references a ParameterDataPrototype in the role perInstanceParameter or sharedParameter within the SwcInternalBehavior of an AUTOSAR software-component. ⌋*(RTE00051, RTE00155)*

Description: The Rte_CData API provides access to the defined calibration parameter within a software-component. The actual data values for a software-component instance may be set after component compilation.

Return Value: The Rte_CData return value provide access to the data value of the ParameterDataPrototype in the role perInstanceParameter or sharedParameter.

The return type of Rte_CData is dependent on the ImplementationDataType of the ParameterDataPrototype and can either be a value or a pointer to the location where the value can be accessed. Thus the component does not need to use type casting to convert access to the ParameterDataPrototype data.

For details of the <return> value definition see section 5.2.6.6.

[rte_sws_3927]⌈ If a ParameterElementPrototype is aggregated by an SwcInternalBehavior in the role of sharedParameter, the return value of the corresponding Rte_CData API shall provide access to the calibration parameter value common to all instances of the AtomicSwComponentType. ⌋*(RTE00051, RTE00155)*

[rte_sws_3952]⌈ If a ParameterElementPrototype is aggregated by an SwcInternalBehavior in the role of perInstanceParameter, the return value of the corresponding Rte_CData API shall provide access to the calibration parameter value specific to the instance of the AtomicSwComponentType. ⌋*(RTE00051, RTE00155)*

Notes: None.

5.6.17 Rte_Prm

Purpose: Provide access to the parameters defined by an AUTOSAR ParameterSwComponentType. Access is read-only.

Signature: **[rte_sws_3928]**⌈

```
<return>
Rte_Prm_<p>_<o>([IN Rte_Instance <instance>])
```

Where <p> is the port name and <o> is the name of the ParameterDataPrototype within the ParameterInterface categorizing the port. ⌋*(BSW00310, RTE00155)*

Existence: **[rte_sws_3929]**⌈ A Rte_Prm API shall be generated if a ParameterAccess references a ParameterDataPrototype in a require PortPrototype. ⌋*(BSW00310, RTE00155)*

Description: The Rte_Prm API provides access to the defined parameter within a ParameterSwComponentType.

In the case of a standard parameter (swImplPolicy = standard), i.e. a calibration, the actual data values for a ParameterSwComponentType instance may be set after ParameterSwComponentType compilation.

In the case of fixed parameter or constant parameter, the value is set during compilation time.

Return Value: **[rte_sws_3930]**⌈ For primitive data types, the Rte_Prm API shall return the parameter value. For composite data types, the Rte_Prm API shall return a reference (in C, a pointer) to the parameter, which shall be const. With fixed parameters, only primitive data is possible.

The return type of Rte_Prm is specified by the ImplementationDataType associated to the ParameterDataPrototype. Thus the component does not need to use type casting to access the calibration parameter. ⌋*(RTE00051, RTE00155, RTE00171)* The Rte_Prm return value provide access to the data value of the ParameterDataPrototype.

The return type of Rte_Prm is dependent on the ImplementationDataType of the ParameterDataPrototype and can either be a value or a pointer to the location where the value can be accessed. Thus the component does not need to use type casting to convert access to the ParameterDataPrototype data.

For details of the <return> value definition see section 5.2.6.6.

Notes: The Rte_Prm API shall not be used within a pre-compilation directive, e.g. #if. For such case, the coder shall use the Rte_SysCon definitions which are dedicated to variant handling.

5.6.18 Rte_IRead

Purpose: Provide **read** access to the VariableDataPrototype referenced by VariableAccess in the dataReadAccess role.

Signature: **[rte_sws_3741]**⌈

```
<return>
Rte_IRead_<re>_<p>_<o>([IN Rte_Instance])
```

Where <re> is the runnable entity name, <p> the port name and <o> the VariableDataPrototype name. ⌋*(BSW00310, RTE00128)*

Existence: **[rte_sws_1301]**⌈ An Rte_IRead API shall be created for a required VariableDataPrototype if the RunnableEntity has a Vari-

ableAccess in the dataReadAccess role referring to this VariableDataPrototype. ⌋*(RTE00051)*

Description: The Rte_IRead API provides access to the VariableDataPrototypes declared as accessed by a runnable using VariableAccesses in the dataReadAccess role. The API function is guaranteed to be have constant execution time and therefore can also be used within category 1A runnable entities.

No error information is provided by this API. If required, the error status can be picked up with a separate API, see 5.6.22

The data value can always be read. To provide the required consistency the API provides access to a *copy* of the data data element for which it's guaranteed that it never changes during the actual execution of the runnable entity.

Implicit data read access by a SW-C should always return defined data.

[rte_sws_1268]⌈ The RTE shall ensure that implicit read accesses will not deliver undefined data item values. ⌋*(RTE00108, RTE00051, RTE00128)*

In case where there may be an implicit read access before the first data reception an initial value has to be provided as the result of this implicit read access.

Return Value: The Rte_IRead return value provide access to the data value of the VariableDataPrototype.

The return type of Rte_IRead is dependent on the ImplementationDataType of the VariableDataPrototype and can either be a value or a pointer to the location where the value can be accessed. Thus the component does not need to use type casting to convert access to the VariableDataPrototype data.

For details of the <return> value definition see section 5.2.6.6.

Notes: None.

5.6.19 Rte_IWrite

Purpose: Provide **write** access to the VariableDataPrototypes referenced by VariableAccesses in the dataWriteAccess role.

Signature: **[rte_sws_3744]**⌈

```
void
Rte_IWrite_<re>_<p>_<o>([IN RTE_Instance],
                        IN <data>)
```

Where <re> is the runnable entity name, <p> the port name and <o> the VariableDataPrototype name. ⌋*(BSW00310, RTE00129)*

Existence: **[rte_sws_1302]**⌈ An Rte_IWrite API shall be created for a provided VariableDataPrototype if the RunnableEntity has a VariableAccess in the dataWriteAccess role referring to this VariableDataPrototype. ⌋*(RTE00051)*

Description: The Rte_IWrite API provides write access to the VariableDataPrototypes declared as accessed by a runnable using VariableAccesses in the dataWriteAccess role. The API function is guaranteed to be have constant execution time and therefore can also be used within category 1A runnable entities.

No access error information is required for the user – the value can always be written. To provide the required write-back semantics the RTE only makes written values available to other entities after the writing runnable entity has terminated.

[rte_sws_3746]⌈ The Rte_IWrite API call includes the IN parameter <data> to pass the data element to write. ⌋*(RTE00051, RTE00129)*

The IN parameter <data> is passed by value or reference according to the ImplementationDataType as described in the section 5.2.6.5.

If the IN parameter <data> is passed by reference, the pointer must remain valid until the API call returns.

Return Value: **[rte_sws_3747]**⌈ Rte_IWrite has no return value. ⌋*(RTE00051)*

For C/C++ rte_sws_3747 means using a return type of void.

Notes: None.

5.6.20 Rte_IWriteRef

Purpose: Provide a reference to the VariableDataPrototype referenced by a VariableAccess in the dataWriteAccess role.

Signature: **[rte_sws_5509]**⌈

```
<return reference>
Rte_IWriteRef_<re>_<p>_<o>([IN RTE_Instance])
```

Where <re> is the runnable entity name, <p> the port name and <o> the VariableDataPrototype name. ⌋*(BSW00310, RTE00129)*

Existence: **[rte_sws_5510]**⌈ An Rte_IWriteRef API shall be created for a provided VariableDataPrototype if the RunnableEntity has a

VariableAccess in the dataWriteAccess role referring to this VariableDataPrototype. ⌋*(RTE00051)*

Description: The Rte_IWriteRef API returns a reference to the VariableDataPrototypes declared as accessed by a runnable using VariableAccesses in the dataWriteAccess role. The reference can be used by the runnable to directly update the corresponding data elements. This is especially useful for data elements of Structure Implementation Data Type or Array Implementation Data Type. The API function is guaranteed to be have constant execution time and therefore can also be used within category 1A runnable entities.

No error information is required for the user. To provide the required write-back semantics the RTE only makes written values available to other entities after the writing runnable entity has terminated.

[rte_sws_ext_7679] The reference returned by Rte_IWriteRef shall not be used by the runnables for reading the value previously written.

The rationale for rte_sws_ext_7679 is that Rte_IWriteRef has a write semantic. Also, in case of an unconnected port, the written data shall be discarded (similarly to rte_sws_1347), and implementations may return a reference to the same buffer for all Rte_IWriteRef of unconnected provide ports.

Return Value: The Rte_IWriteRef return value provide access to the data write buffer of the VariableDataPrototype.

[rte_sws_5511]⌈ Rte_IWriteRef returns a reference to the corresponding VariableDataPrototype. ⌋*(RTE00051)*

The return reference type of Rte_IWriteRef is dependent on the ImplementationDataType of the VariableDataPrototype and is a pointer to the location where the value can be accessed. Thus the component does not need to use type casting to convert access to the VariableDataPrototype data.

For details of the <return reference> definition see section 5.2.6.7.

Notes: None.

5.6.21 Rte_IInvalidate

Purpose: Invalidate a VariableDataPrototype referenced by a VariableAccess in the dataWriteAccess role.

Signature: **[rte_sws_3800]**⌈
`void Rte_IInvalidate_<re>_<p>_<o>([IN Rte_Instance <instance>])`

Where `<re>` is the runnable entity name, `<p>` the port name and `<o>` the `VariableDataPrototype` name. ⌋*(BSW00310, RTE00078)*

Existence: **[rte_sws_3801]**⌈ An `Rte_IInvalidate` API shall be created for a provided `VariableDataPrototype` if the `RunnableEntity` has `VariableAccess`es in the `dataWriteAccess` role referring to this `VariableDataPrototype` and the associated `InvalidationPolicy` of the `VariableDataPrototype` is set to `keep` or `replace`. ⌋*(RTE00051, RTE00078)*

Description: The `Rte_IInvalidate` API takes no parameters other than the instance handle – the return value is used to indicate the success, or otherwise, of the API call to the caller.

[rte_sws_3802]⌈ In case of a primitive `VariableDataPrototype` the `Rte_IInvalidate` shall be implemented as a macro that writes the `invalidValue` to the buffer. ⌋*(RTE00078)*

[rte_sws_5064]⌈ In case of a composite `VariableDataPrototype` the `Rte_IInvalidate` shall be implemented as a macro that writes the `invalidValue` of every primitive part of the composition to the buffer. ⌋*(RTE00078)*

[rte_sws_3778]⌈ If `Rte_IInvalidate` is followed by an `Rte_IWrite` call for the same `VariableDataPrototype` or vice versa, the RTE shall use the last value written before the runnable entity terminates (last-is-best semantics). ⌋*(RTE00078)*

rte_sws_3778 states that an `Rte_IWrite` overrules an `Rte_IInvalidate` call if it occurs after the `Rte_IInvalidate`, since `Rte_IWrite` overwrites the contents of the internal buffer for the data element prototype before it is made known to other runnable entities.

Return Value: **[rte_sws_3803]**⌈ `Rte_IInvalidate` has no return value. ⌋*(RTE00094)*

For C/C++ rte_sws_3803 means using a return type of `void`.

Notes: The communication service configuration determines whether the signal receiver(s) receive an “invalid signal” notification or whether the invalidated signal is silently replaced by the signal’s initial value.

5.6.22 Rte_IStatus

Purpose: Provide the error status of a `VariableDataPrototype` referenced by a `VariableAccess` in the `dataReadAccess` role.

Signature: **[rte_sws_2599]**⌈
`Std_ReturnType`
`Rte_IStatus_<re>_<p>_<o>([IN Rte_Instance])`

Where `<re>` is the runnable entity name, `<p>` the port name and `<o>` the `VariableDataPrototype` name. ⌋*(RTE00147, RTE00078)*

Existence: **[rte_sws_2600]**⌈ An `Rte_IStatus` API shall be created for a required `VariableDataPrototype` if a RunnableEntity has a `VariableAccess` in the `dataReadAccess` role referring to this `VariableDataPrototype`, and

- if at the `RPortPrototype` an `NonqueuedReceiverComSpec` with either
  - the attribute `AliveTimeout` set to a value greater than zero **and/or**
  - the attribute `handleNeverReceived` set to `TRUE`

  and/or
- if at the `SenderReceiverInterface` classifying the `RPortPrototype` an `InvalidationPolicy` set to `keep`

is specified for this `VariableDataPrototype`. ⌋*(RTE00147, RTE00078)*

[rte_sws_ext_2601] The `Rte_IStatus` API shall only be used by a RunnableEntity that either has a `VariableAccess` in the `dataReadAccess` role referring to the `VariableDataPrototype` or is triggered by a DataReceiveErrorEvent referring to the `VariableDataPrototype`.

Description: The `Rte_IStatus` API provides access to the current status of the data elements declared as accessed by a runnable using a `VariableAccess` in the `dataReadAccess` role. The API function is guaranteed to be have constant execution time and therefore can also be used within category 1A runnable entities.

To provide the required consistency access by a runnable is to a *copy* of the status together with the data that is guaranteed never to be modified by the RTE during the lifetime of the runnable entity.

Return Value: The return value is used to indicate errors detected by the communication system.

- **[rte_sws_2602]**⌈ `RTE_E_OK` – no errors. ⌋*(RTE00094)*
- **[rte_sws_2603]**⌈ `RTE_E_INVALID` – data element invalid. ⌋*(RTE00078)*

- **[rte_sws_2604]**⌈ `RTE_E_MAX_AGE_EXCEEDED` – `data element outdated`. This `Overlayed Error` can be combined with any of the above error codes. ⌋*(RTE00147)*
- **[rte_sws_7644]**⌈ `RTE_E_NEVER_RECEIVED` – No data received since system start or partition restart. ⌋*(RTE00184, RTE00224)*

  [rte_sws_7691]⌈ `RTE_E_UNCONNECTED` – Indicates that the receiver port is not connected. ⌋*(RTE00094, RTE00139, RTE00200)*

Notes: None.

5.6.23 Rte_IrvIRead

Purpose: Provide **read** access to the *InterRunnableVariables with implicit* behavior of an AUTOSAR SW-C.

Signature: **[rte_sws_3550]**⌈

```
<return>
Rte_IrvIRead_<re>_<o>([IN RTE_Instance <instance>])
```

Where `<re>` is the name of the runnable entity the API might be used in, `<o>` is the name of the `VariableDataPrototype` in role `implicitInterRunnableVariable`. ⌋*(BSW00310, RTE00142)*

Existence: **[rte_sws_1303]**⌈ An `Rte_IrvIRead` API shall be created for each `VariableAccess` in role `readLocalVariable` to an `implicitInterRunnableVariable`. ⌋*(RTE00051, RTE00142)*

Description: The `Rte_IrvIRead` API provides read access to the defined InterRunnableVariables with *implicit* behavior within a component description.

The return value is used to deliver the requested data value. The return value is not required to pass error information to the user because no inter-ECU communication is involved and there will always be a readable value present.

Requirement rte_sws_3581 is valid for InterRunnableVariables with implicit and InterRunnableVariables with explicit behavior:

[rte_sws_3581]⌈ The RTE has to ensure that read accesses to an InterRunnableVariables won't deliver undefined data item values. In case write access before read access cannot be guaranteed by configuration an initial values for the InterRunnableVariable has to be written to it. ⌋*(RTE00142)*

This initial value has to be an input for the RTE generator and might be initially defined in the AUTOSAR SW-C description.

Return Value: The `Rte_IrvIRead` return value provide access to the data value of the InterRunnableVariable.

The return type of `Rte_IrvIRead` is dependent on the `ImplementationDataType` of the InterRunnableVariable and can either be a value or a pointer to the location where the value can be accessed. Thus the component does not need to use type casting to convert access to the InterRunnableVariable data.

For details of the `<return>` value definition see section 5.2.6.6.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.5.6. More details about InterRunnableVariables with *implicit* behavior is explained in section 4.2.5.6.1.

5.6.24 Rte_IrvIWrite

Purpose: Provide **write** access to the *InterRunnableVariables with implicit behavior* of an AUTOSAR SW-C.

Signature: **[rte_sws_3553]**⌈

```
void
Rte_IrvIWrite_<re>_<o>([IN RTE_Instance <instance>],
                       IN <data>)
```

Where `<re>` is the name of the `RunnableEntity` the API might be used in, `<o>` is the name of the `VariableDataPrototype` in the role `implicitInterRunnableVariable` to access and `<data>` is the placeholder for the data the InterRunnableVariable shall be set to. ⌋*(BSW00310, RTE00142)*

Existence: **[rte_sws_1304]**⌈ An `Rte_IrvIWrite` API shall be created for each `VariableAccess` in role `writtenLocalVariable` to an `implicitInterRunnableVariable`. ⌋*(RTE00142, RTE00051)*

Description: The `Rte_IrvIWrite` API provides write access to the InterRunnableVariables with *implicit* behavior within a component description. The runnable entity name in the signature allows runnable context specific optimizations.

The data given by `Rte_IrvIWrite` is dependent on the InterRunnableVariable data type. Thus the component does not need to use type casting to write the InterRunnableVariable.

The return value is unused. The return value is not required to pass error information to the user because no inter-ECU communication is involved and the value can always be written.

The IN parameter `<data>` is passed by value or reference according to the `ImplementationDataType` as described in the section 5.2.6.5.

Return Value: **[rte_sws_3555]**⌈ `Rte_IrvIWrite` shall have no return value. ⌋*(RTE00142, RTE00051)*

For C/C++, requirement rte_sws_3555 means using a return type of `void`.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.5.6. Further details about InterRunnableVariables with *implicit* behavior are explained in Section 4.2.5.6.1.

5.6.25 Rte_IrvRead

Purpose: Provide **read** access to the *InterRunnableVariables with explicit behavior* of an AUTOSAR SW-C.

Signature: **[rte_sws_3560]**⌈
primitive type signature:

```
<return>
Rte_IrvRead_<re>_<o>([IN RTE_Instance <instance>])
```

complex type signature:

```
void
Rte_IrvRead_<re>_<o>([IN RTE_Instance <instance>], OUT <data>)
```

Where `<re>` is the name of the runnable entity the API might be used in, `<o>` is the name of the InterRunnableVariables.

The complex type signature is used, if the `ImplementationDataType` of the `InterRunnableVariable` resolves to `Array Implementation Data Type` or `Structure Implementation Data Type`, otherwise the primitive type signature is used. ⌋*(BSW00310, RTE00142)*

Existence: **[rte_sws_1305]**⌈ An `Rte_IrvRead` API shall be created for each read InterRunnableVariable using explicit access.

⌋*(RTE00142, RTE00051)*

Description: The `Rte_IrvRead` API provides read access to the defined InterRunnableVariables with *explicit* behavior within a component description.

The return value is not required to pass error information to the user because no inter-ECU communication is involved and there will always be a readable value present.

For the primitive type signature, the return value is used to deliver the requested data value. For the complex type signature, the return value is void.

For the complex type signature, the `Rte_IrvRead` API call includes the OUT parameter `<data>` to pass back the received data. The OUT parameter `<data>` is typed as reference (pointer) to the type of the InterRunnableVariable. The pointer to the OUT parameter `<data>` must remain valid until the API call returns.

Return Value: The `Rte_IrvRead` return value provide access to the data value of the InterRunnableVariable.

The return type of `Rte_IrvRead` is dependent on the `ImplementationDataType` of the InterRunnableVariable. Thus the component does not need to use type casting to convert access to the InterRunnableVariable data.

For details of the `<return>` value definition see section 5.2.6.6.

Please note that the `Rte_IrvRead` API Signature only has a return value if the InterRunnableVariable is typed by a `Primitive Implementation Data Type` or `Redefinition Implementation Data Type` redefining a `Primitive Implementation Data Type`.

[rte_sws_3562]⌈ For the primitive type signature, the `Rte_IrvRead` call shall return the value of the accessed InterRunnableVariable. ⌋*(RTE00142, RTE00051)*

For complex type signature, the `Rte_IrvRead` call does not return any value (void).

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.5.6. Further details about InterRunnableVariables with *explicit* behavior are explained in Section 4.2.5.6.2.

5.6.26 Rte_IrvWrite

Purpose: Provide **write** access to the *InterRunnableVariables with explicit behavior* of an AUTOSAR SW-C.

Signature: **[rte_sws_3565]**⌈

```
void
Rte_IrvWrite_<re>_<o>([IN RTE_Instance <instance>],
                      IN <data>)
```

Where `<re>` is the name of the runnable entity the API might be used in, `<o>` is the name of the InterRunnableVariable to access and `<data>` is the placeholder for the data the InterRunnableVariable shall be set to. ⌋*(BSW00310, RTE00142)*

Existence: **[rte_sws_1306]**⌈ An `Rte_IrvWrite` API shall be created for each written InterRunnableVariable using explicit access. ⌋*(RTE00142, RTE00051)*

Description: The `Rte_IrvWrite` API provides write access to the InterRunnableVariables with *explicit* behavior within a component description.

The return value is unused. The return value is not required to pass error information to the user because no inter-ECU communication is involved and the value can always be written.

[rte_sws_3567]⌈ The `Rte_IrvWrite` API call include the IN parameter `<data>` to pass the data element to write. ⌋*(RTE00142, RTE00051)*

The IN parameter `<data>` is passed by value or reference according to the `ImplementationDataType` as described in the section 5.2.6.5.

If the IN parameter `<data>` is passed by reference, the pointer must remain valid until the API call returns.

Return Value: **[rte_sws_3569]**⌈ `Rte_IrvWrite` shall have no return value. ⌋*(RTE00142)*

For C/C++, requirement rte_sws_3569 means using a return type of `void`.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.5.6. Further details about InterRunnableVariables with *explicit* behavior are explained in Section 4.2.5.6.2.

5.6.27 Rte_Enter

Purpose: Enter an exclusive area.

Signature: **[rte_sws_1120]**⌈
```
void
Rte_Enter_<name>([IN Rte_Instance <instance>])
```
Where `<name>` is the exclusive area name. ⌋*(BSW00310, RTE00046, RTE00115)*

Existence: **[rte_sws_1307]**⌈ An `Rte_Enter` API shall be created for each ExclusiveArea that is declared and which has an *canEnterExclusiveArea* association. ⌋*(RTE00115, RTE00051)*

Description: The `Rte_Enter` API call is invoked by an AUTOSAR software-component to define the start of an exclusive area.

Return Value: None.

Notes: The RTE is not required to support nested invocations of `Rte_Enter` for the same exclusive area.

[rte_sws_1122]⌈ The RTE shall permit calls to `Rte_Enter` and `Rte_Exit` to be nested as long as different exclusive areas are exited in the reverse order they were entered. ⌋*(RTE00046, RTE00032, RTE00115)*

[rte_sws_ext_7171] The `Rte_Enter` and `Rte_Exit` API may only be used by *Runnable Entities* that contain a corresponding *canEnterExclusiveArea* association

[rte_sws_ext_7172] The `Rte_Enter` and `Rte_Exit` API may only be called nested if different exclusive areas are invoked; in this case exclusive areas shall exited in the reverse order they were entered.

Within the AUTOSAR OS an attempt to lock a resource cannot fail because the lock is already held. The lock attempt can only fail due to configuration errors (e.g. caller not declared as accessing the resource) or invalid handle. Therefore the return type from this function is `void`.

5.6.28 Rte_Exit

Purpose: Leave an exclusive area.

Signature: **[rte_sws_1123]**⌈
```
void
Rte_Exit_<name>([IN Rte_Instance <instance>])
```

Where <name> is the exclusive area name. ⌋*(BSW00310, RTE00046, RTE00051)*

Existence: **[rte_sws_1308]**⌈ An Rte_Exit API shall be created for each ExclusiveArea that is declared and which has an *canEnterExclusiveArea* association. ⌋*(RTE00115, RTE00051)*

Description: The Rte_Exit API call is invoked by an AUTOSAR software-component to define the end of an exclusive area.

Return Value: None.

Notes: The RTE is not required to support nested invocations of Rte_Exit for the same exclusive area.

Requirement rte_sws_1122 permits calls to Rte_Enter and Rte_Exit to be nested as long as different exclusive areas are exited in the reverse order they were entered.

5.6.29 Rte_Mode

There exist two versions of the Rte_Mode API. Depending on the attribute *enhancedModeApi* in the *software component description* there shall be provided different versions of this API (see also 5.6.30).

Purpose: Provides the currently active mode of a mode switch port.

Signature: **[rte_sws_2628]**⌈

```
<return>
Rte_Mode_<p>_<o>([IN Rte_Instance <instance>])
```

Where <p> is the port name, and <o> the *ModeDeclarationGroupPrototype* name within the ModeSwitchInterface categorizing the port. ⌋*(RTE00144)*

Existence: **[rte_sws_2629]**⌈ If a *ModeAccessPoint* exists and if the attribute *enhancedModeApi* of the ModeSwitchSenderComSpec resp. ModeSwitchReceiverComSpec is set to *false* or does not exist a Rte_Mode API according to rte_sws_2628 shall be generated. ⌋*(RTE00147, RTE00078)*

[rte_sws_ext_7568] The Rte_Mode API may only be used by the runnable that contains the corresponding *ModeAccessPoint*

Description: The Rte_Mode API tells the AUTOSAR software-component which mode of a *ModeDeclarationGroup* of a given port is currently active. This is the information that the RTE uses for the ModeDisablingDependency's. A new mode will not be indicated immediately after the reception of a mode switch notification from a mode manager, see section 4.4.4. During mode transitions, i.e.

during the execution of runnables that are triggered on exiting one mode or on entering the next mode, overlapping mode disablings of two modes are active. In this case, the Rte_Mode will return RTE_TRANSITION_<ModeDeclarationGroup>.

The Rte_Mode will return the same mode for all mode switch ports that are connected to the same mode switch port of the mode manager (see rte_sws_2630).

It is supported to have *ModeAccessPoint*(s) referring the provide mode switch ports of the mode manager to provide access for the mode manager on the information that the RTE uses for the ModeDisablingDependency's.

Return Value: The return type of Rte_Mode is dependent on the ImplementationDataType of the *ModeDeclarationGroup*. It shall return the value of the *ModeDeclarationGroupPrototype*. The type name shall be equal to the shortName of the ImplementationDataType.

The return value of the Rte_Mode is used to inform the caller about the current mode of the mode machine instance. The Rte_Mode API shall return the following values:

- **[rte_sws_2731]**⌈If the mode users of a mode machine instance are in a partition that is stopped or restarting, Rte_Mode shall return RTE_TRANSITION_<ModeDeclarationGroup>, where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup.⌋*(RTE00144, RTE00210)*

  Note, that this can only occur as the return value to the mode manager and only, if the mode manager is in another, running partition.

- **[rte_sws_7666]**⌈During a transition of the mode machine instance, Rte_Mode shall return RTE_TRANSITION_<ModeDeclarationGroup>, where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup.⌋*(RTE00144)*

- **[rte_sws_2660]**⌈When the mode machine instance is in a defined mode, Rte_Mode shall return RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>, where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup and <ModeDeclaration> is the short name of the currently active ModeDeclaration.⌋*(RTE00144)*

In inter partition mode management, RTE on the mode manager sided partition might not have direct access to the state variables of the mode machine instance.

[rte_sws_2732]⌈ In inter partition mode management, the return value of the Rte_Mode API to the mode manager shall be consistent with the start of a transition by the Rte_Switch API and the inter partition communication of the ModeSwitchedAckEvent. ⌋*(RTE00144, RTE00210)*RTE00144

Notes: The Rte_Mode API may already indicate the next ModeDeclaration, before the mode manager has picked up the ModeSwitchedAckEvent with the Rte_SwitchAck. This is not in contradiction to rte_sws_2732.

5.6.30 Enhanced Rte_Mode

Purpose: Provides the currently active mode of a mode switch port. If the mode machine instance is in transition additionally the values of the previous and the next mode are provided.

Signature: **[rte_sws_8500]**⌈

```
<return>
Rte_Mode_<p>_<o>([IN Rte_Instance <instance>,]
                 OUT <previousmode>,
                 OUT <nextmode>)
```

Where <p> is the port name, and <o> the *ModeDeclarationGroupPrototype* name within the ModeSwitchInterface categorizing the port. ⌋*(RTE00144)*

Existence: **[rte_sws_8501]**⌈ The existence of a *ModeAccessPoint* given that the attribute *enhancedModeApi* of the ModeSwitchReceiverComSpec resp. ModeSwitchReceiverComSpec is set to *true* shall result in the generation of a Rte_Mode API according to rte_sws_8500. ⌋*(RTE00147, RTE00078)*

[rte_sws_ext_8502] The Rte_Mode API may only be used by the runnable that contains the corresponding *ModeAccessPoint*

Description: The Rte_Mode API tells the AUTOSAR software-component which mode of a *ModeDeclarationGroup* of a given port is currently active. This is the information that the RTE uses for the ModeDisablingDependency's. A new mode will not be indicated immediately after the reception of a mode switch notification from a mode manager, see section 4.4.4. During mode transitions, i.e. during the execution of runnables that are triggered on exiting one mode or on entering the next mode, overlapping mode disablings

of two modes are active. In this case, the Rte_Mode will return RTE_TRANSITION_<ModeDeclarationGroup>. The parameter <previousmode> than contains the mode currently being left,the parameter <nextmode> the mode being entered.

The Rte_Mode will return the same mode for all mode switch ports that are connected to the same mode switch port of the mode manager (see rte_sws_2630).

It is supported to have *ModeAccessPoint*(s) referring the provided mode switch ports of the mode manager to provide access for the mode manager on the information that the RTE uses for the ModeDisablingDependency's.

Return Value: The return type of Rte_Mode is dependent on the ImplementationDataType of the *ModeDeclarationGroup*. It shall return the value of the *ModeDeclarationGroupPrototype*. The type name shall be equal to the shortName of the ImplementationDataType. The return value of the Rte_Mode and the parameters <previousmode> and <nextmode> are used to inform the caller about the current mode of the mode machine instance.

[rte_sws_8503]⌈If the mode users of a mode machine instance are in a partition that is stopped or restarting, Rte_Mode shall return the following values

- the return value shall be RTE_TRANSITION_<ModeDeclarationGroup>,
- <previousmode> shall contain the value of the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> of the initialMode of the ModeDeclarationGroup
- <nextmode> shall contain the value of the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> of the initialMode of the ModeDeclarationGroup

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup. ⌋*(RTE00144, RTE00210)*

[rte_sws_8504]⌈ During a transition of a mode machine instance Rte_Mode shall return the following values

- the return value shall be RTE_TRANSITION_<ModeDeclarationGroup>,
- <previousmode> shall contain the value of the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> of the mode being left,

- <nextmode> shall contain the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> of the mode being entered,

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup and <ModeDeclaration> is the short name of the ModeDeclaration. ⌋*(RTE00144, RTE00210)*

[rte_sws_8505]⌈When the mode machine instance is in a defined mode, Rte_Mode shall return the follwoing values

- the return value shall contain the value of RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>,
- <previousmode> shall contain the value of the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>
- <nextmode> shall contain the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup and <ModeDeclaration> is the short name of the currently active ModeDeclaration.⌋*(RTE00144)*

In inter partition mode management, RTE on the mode manager sided partition might not have direct access to the state variables of the mode machine instance.

[rte_sws_8506]⌈ In inter partition mode management, the return value and the contents of the parameters <previousmode> and <nextmode> of the Rte_Mode API to the mode manager shall be consistent with the start of a transition by the Rte_Switch API and the inter partition communication of the ModeSwitchedAckEvent. ⌋*(RTE00144, RTE00210)*

Notes: The Rte_Mode API may already indicate the next ModeDeclaration, before the mode manager has picked up the ModeSwitchedAck-Event with the Rte_SwitchAck. This is not in contradiction to rte_sws_2732.

5.6.31 Rte_Trigger

Purpose: Raise a external trigger of a trigger port.

Signature: **[rte_sws_7200]**⌈
signature without queuing support:

```
void
Rte_Trigger_<p>_<o>([IN Rte_Instance <instance>])
```

signature with queuing support:

```
Std_ReturnType
Rte_Trigger_<p>_<o>([IN Rte_Instance <instance>])
```

Where <p> is the port name and <o> the Trigger within the trigger interface categorizing the port.

The signature for queuing support shall be generated by the RTE generator if the swImplPolicy of the associated Trigger is set to queued. ⌋*(RTE00162)*

Existence: **[rte_sws_7201]**⌈ The existence of a ExternalTriggeringPoint shall result in the generation of a Rte_Trigger API. ⌋*(RTE00162)*

[rte_sws_ext_7202] The Rte_Trigger API may only be used by the runnable that contains the corresponding ExternalTriggering-Point.

Description: The Rte_Trigger API triggers an execution for all runnables whose ExternalTriggerOccurredEvent is associated to the Trigger.

Return Value: None in case of signature without queuing support.

[rte_sws_6720]⌈ The Rte_Trigger API shall return the following values:

- RTE_E_OK if the trigger was successfully queued or if no queue is configured
- RTE_E_LIMIT if the trigger was not queued because the maximum queue size is already reached.

in the case of signature with queuing support. ⌋*(RTE00235)*

Notes: Rte_Trigger is restricted to ECU local communication.

5.6.32 Rte_IrTrigger

Purpose: Raise a internal trigger to activate Runnable entities of the same software component instance.

Signature: **[rte_sws_7203]**⌈
signature without queuing support:

```
void
Rte_IrTrigger_<re>_<o>([IN Rte_Instance <instance>])
```

signature with queuing support:

```
Std_ReturnType
Rte_IrTrigger_<re>_<o>([IN Rte_Instance <instance>])
```

Where <re> is the name of the runnable entity the API might be used in and <o> is the name of the InternalTriggeringPoint.

The signature for queuing support shall be generated by the RTE generator if the swImplPolicy of the associated InternalTriggeringPoint is set to queued. ⌋*(RTE00163)*

Existence: **[rte_sws_7204]**⌈ The existence of a InternalTriggeringPoint shall result in the generation of a Rte_IrTrigger API. ⌋*(RTE00163)*

[rte_sws_ext_7205] The Rte_IrTrigger API may only be used by the runnable that contains the corresponding InternalTriggeringPoint.

Description: The Rte_IrTrigger triggers an execution for all runnables whose InternalTriggerOccurredEvent is associated to the InternalTriggeringPoint.

Return Value: None in case of signature without queuing support.

[rte_sws_6721]⌈ The Rte_Trigger API shall return the following values:

- RTE_E_OK if the trigger was successfully queued or if no queue is configured
- RTE_E_LIMIT if the trigger was not queued because the maximum queue size is already reached.

in the case of signature with queuing support. ⌋*(RTE00235)*

Notes: None.

5.6.33 Rte_IFeedback

Purpose: Provide access to acknowledgement notifications for implicit sender receiver communication and to pass error notification to senders.

Signature: **[rte_sws_7367]**⌈

```
Std_ReturnType
Rte_IFeedback_<re>_<p>_<o> ([IN RTE_Instance <instance>])
```

Where <re> is the runnable entity name, <p> the port name and <o> the VariableDataPrototype within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00122, RTE00129, RTE00185)*

Existence: Note: according to rte_sws_1283, acknowledgment is enabled for a provided `VariableDataPrototype` by the existence of a `TransmissionAcknowledgementRequest` in the `SenderComSpec`.

[rte_sws_7646]⌈ An `Rte_IFeedback` API shall be created for a provided `VariableDataPrototype` if acknowledgment is enabled and the `RunnableEntity` has a `VariableAccess` in the `dataWriteAccess` role referring to this `VariableDataPrototype`.⌋*(RTE00122, RTE00129, RTE00185)*

[rte_sws_7647]⌈ An `Rte_IFeedback` API shall be created for a provided `VariableDataPrototype` if acknowledgment is enabled and a `DataWriteCompletedEvent` references the `RunnableEntity` as well as the `VariableDataPrototype`. ⌋*(RTE00122, RTE00129, RTE00185)*

[rte_sws_7648]⌈ If acknowledgment is enabled for a provided `VariableDataPrototype` and a `DataWriteCompletedEvent` references a runnable entity as well as the `VariableDataPrototype`, the runnable entity shall be activated when the transmission acknowledgment occurs or when a timeout was detected by the RTE. See rte_sws_7379. ⌋*(RTE00122, RTE00129, RTE00185)*

[rte_sws_7649]⌈ The `Rte_IFeedback` API shall only be used by by a `RunnableEntity` that either has a `VariableAccess` in the `dataWriteAccess` role referring to the `VariableDataPrototype` or is triggered by a `DataWriteCompletedEvent` referring to the `VariableDataPrototype`. ⌋*(RTE00122, RTE00129, RTE00185)*

Description: The `Rte_IFeedback` API takes no parameters other than the instance handle – the return value is used to indicate the acknowledgment status to the caller.

The `Rte_IFeedback` API applies only to implicit sender-receiver communication.

The `Rte_IFeedback` API provides access to the transmission feedback of the data elements, declared as sent by a runnable using a `VariableAccess` in the `dataWriteAccess` role, and sent after the previous invocation of the runnable. The API function is guaranteed to be have constant execution time and therefore can also be used within category 1A runnable entities.

The required consistency access by a runnable can be provided by copying of the status before the execution of the runnable so that it cannot be modified by the RTE during the lifetime of the runnable entity.

Return Value: The return value is used to indicate the "status" status and errors detected by the RTE during execution of the `Rte_IFeedback` call.

- **[rte_sws_7374]**⌈ `RTE_E_NO_DATA` – No acknowledgments or error notifications were received from COM when the runnable entity was started. ⌋*(RTE00094, RTE00122, RTE00129, RTE00185)*
- **[rte_sws_7375]**⌈ `RTE_E_COM_STOPPED` – (Inter-ECU communication only) The last transmission was rejected (when the local buffer was sent), with an `RTE_E_COM_STOPPED` return code or an error notification was received from COM before any timeout notification. ⌋*(RTE00094, RTE00122, RTE00129, RTE00185)*
- **[rte_sws_7650]**⌈ `RTE_E_TIMEOUT` – (Inter-ECU only) A timeout notification was received from COM before any error notification. ⌋*(RTE00094, RTE00122, RTE00129, RTE00185)*
- **[rte_sws_7376]**⌈ `RTE_E_TRANSMIT_ACK` – A transmission acknowledgment was received. This error code is valid for both inter-ECU and intra-ECU communication. ⌋*(RTE00094, RTE00122, RTE00129, RTE00185)*
- **[rte_sws_7660]**⌈ `RTE_E_UNCONNECTED` – Indicates that the sender port is not connected. ⌋*(RTE00094, RTE00122, RTE00129, RTE00185, RTE00139)*

The `RTE_E_TRANSMIT_ACK` and `RTE_E_UNCONNECTED` return values are not considered to be an error but rather indicates correct operation of the API call.

[rte_sws_7651]⌈ The initial return value of the `Rte_IFeedback` API, when the runnable entity is executed before any attempt to write some data shall be `RTE_E_TRANSMIT_ACK`. ⌋*(RTE00094, RTE00122, RTE00129, RTE00185)*

Notes: See the notes in for the `Rte_Feedback` API in section 5.6.8.

5.6.34 Rte_IsUpdated

Purpose: Provide access to the update flag for an explicit receiver.

Signature: **[rte_sws_7390]**⌈

```
boolean
Rte_IsUpdated_<p>_<o> ([IN RTE_Instance <instance>])
```

Where `<p>` is the port name and `<o>` the `VariableDataPrototype` within the sender-receiver interface categorizing the port. ⌋*(BSW00310, RTE00179)*

Existence: **[rte_sws_7391]**⌈ An `Rte_IsUpdated` API shall be created for a required `VariableDataPrototype` if a `RunnableEntity` has a `VariableAccess` in the `dataReceivePointByArgument` or `dataReceivePointByValue` role referring to the `VariableDataPrototype` and the `enableUpdate` attribute is enabled in the `NonqueuedReceiverComSpec` of the `VariableDataPrototype`. ⌋*(RTE00179)*

[rte_sws_ext_7603] The `Rte_IsUpdated` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByArgument` or `dataReceivePointByValue` role.

Description: The `Rte_IsUpdated` API takes no parameters other than the instance handle – the return value is used to indicate if the `VariableDataPrototype` has been updated or not.

The `Rte_IsUpdated` API applies only to sender-receiver communication.

Return Value: The return value is used to indicate if the `VariableDataPrototype` has been updated or not.

- **[rte_sws_7392]**⌈ `TRUE` – DataElement updated since last read. ⌋*(RTE00094, RTE00179)*
- **[rte_sws_7393]**⌈ `FALSE` – DataElement not updated since last read. ⌋*(RTE00094, RTE00179)*

Notes: None.

5.7 Runnable Entity Reference

An AUTOSAR component defines one or more “runnable entities”. A runnable entity is a piece of code with a single entry point and an associate set of data. A software-component description provides definitions for each runnable entity within the software-component.

For components implemented using C or C++ the entry point of a runnable entity is implemented by a function with global scope defined within a software-component’s source code. The following sections consider the function signature and prototype.

5.7.1 Signature

The definition of all runnable entities, whatever the `RTEEvent` that triggers their execution, follows the same basic form.

[rte_sws_1126]⌈

```
<void|Std_ReturnType> <name>([IN Rte_Instance <instance>],
                             [role parameters])
```

Where <name> [8] is the symbol describing the runnable's entry point (symbol). The definition of the *role parameters* is defined in Section 5.7.3. ⌋*(RTE00031)*

Section 5.2.6.4 contains details on a recommended naming conventions for runnable entities based on the RTEEvent that triggers the runnable entity. The recommended naming convention makes explicit the functions that implement runnable entities as well as clearly associating the runnable entity and the applicable data element or operation.

5.7.2 Entry Point Prototype

The RTE determines the required role parameters, and hence the prototype of the entry point, for a runnable entity based on information in the input information. The entry point defined in the component source *must* be compatible with the parameters passed by the RTE when the runnable entity is triggered by the RTE and therefore the RTE generator is required to emit a prototype for the function.

[rte_sws_1132]⌈ The RTE generator shall emit a prototype for the runnable entity's entry point in the *Application Header File*. ⌋*(RTE00087, RTE00051, RTE00031)*

The prototype for a function implementing the entry point of a runnable entity is emitted for both "RTE Contract" and "RTE Generation" phases. The function name for the prototype is the runnable entity's entry point. The prototype of the entry point function includes the runnable entity's instance handle and its role parameters, see Figure 5.2.

[rte_sws_7194]⌈ The RTE Generator shall wrap each RunnableEntity's *Entry Point Prototype* in the *Application Header File* with the *Memory Mapping* and *Compiler Abstraction* macros.

```
1  #define <c>_START_SEC_<sadm>
2  #include "<c>_MemMap.h"
3
4  FUNC(<void|Std_ReturnType>, <c>_<sadm>)  <prefix><name> (
5                               [IN Rte_Instance <instance>],
6                               [role parameters]);
7
8  #define <c>_STOP_SEC_<sadm>
9  #include "<c>_MemMap.h"
```

where <c> is the shortName of the software component type,

<sadm> is the shortName of the referred swAddrMethod.

[8] Runnable entities have two "names" associated with them in the AUTOSAR Software Component Template; the runnable's identifier and the entry point's symbol. The identifier is used to reference the runnable entity within the input data and the symbol used within code to identify the runnable's implementation. In the context of a prototype for a runnable entity, "name" is the runnable entity's entry point symbol.

<prefix> is the optional SymbolProps.symbol attribute of the AtomicSoftware-ComponentType owning the RunnableEntity.

<name> is the attribute symbol describing the RunnableEntity's entry point.

The definition of the *role parameters* is defined in Section 5.7.3. The Memory Mapping macros could wrap several *Entry Point Prototype* if these are referring to the same swAddrMethod. If RunnableEntity does not refer a swAddrMethod the <sadm> is set to default CODE. ⌋*(RTE00148, RTE00149)*

[rte_sws_6531]⌈ The RTE Generator shall wrap each *Entry Point Prototype* in the *Application Header File* of a variant existent RunnableEntity if the variability shall be implemented. ⌋*(RTE00201)*

```
#if (<condition>)

<Entry Point Prototype>

#endif
```

where condition is the *Condition Value Macro* of the VariationPoint relevant for the variant existence of the RunnableEntity (see table 4.16), Entry Point Prototype is the code according an invariant *Entry Point Prototype* (see also rte_sws_1131, rte_sws_7177, rte_sws_2512, rte_sws_1133, rte_sws_1359, rte_sws_1166, rte_sws_1135, rte_sws_1137, rte_sws_7207, rte_sws_7208, rte_sws_7379).

[rte_sws_1016]⌈ The function implementing the entry point of a runnable entity shall define an instance handle as the first formal parameter if and only if the software component's supportsMultipleInstantiation attribute is set to TRUE. ⌋*(RTE00011, RTE00031)*

The RTE will ensure that when the runnable entity is triggered the instance handle parameter indicates the correct component instance. The remaining parameters passed to the runnable entity depend on the RTEEvent that triggers execution of the runnable entity.

Due to the global name space of a C Linker Locater symbols of RunnableEntitys have to be unique in the ECU. When AtomicSoftwareComponentTypes of several vendors are integrated in the same ECU name clashes might occur if the same symbol is accidentally used twice. To ease the dissolving of name clashes the RTE supports an abstraction of the RunnableEntity symbol in the implementation of the software component.

[rte_sws_6713]⌈ The RTE generator shall emit for each RunnableEntity a define for a symbolic name of the RunnableEntity.

```
#define RTE_RUNNABLE_<name> <prefix><symbol>
```

where <name> is the shortName of the RunnableEntity,

<prefix> is the optional SymbolProps.symbol attribute of the AtomicSoftwareComponentType owning the RunnableEntity.

<symbol> is the attribute symbol describing the RunnableEntity's entry point.

⌋*(RTE00087, RTE00051, RTE00031)*

This symbolic name of the RunnableEntity can be used as follows in the software component implementation.

Example 5.29

For software component "'HugeSwc"' with a runnable "'FOO"' where the SymbolProps.symbol is set to "'TinySwc"' the *Application Header File* contains the definition:

```
1 /* Application Header File of HugeSwc*/
2 #define RTE_RUNNABLE_FOO TinySwcfoo
```

This can be used in the software components c file for the definition of the runnable:

```
1 /* software component c file */
2 RTE_RUNNABLE_FOO()
3 {
4         /* The algorithm of foo */
5         return;
6 }
```

A change of the SymbolProps.symbol valued would have no effect on the c implementation of the software component but it would change the contract and the used labels in a object code delivery.

5.7.3 Role Parameters

The *role parameters* are optional and their presence and types depend on the RTEEvent that triggers the execution of the runnable entity. The role parameters that are necessary for each triggering RTEEvent are defined in Section 5.7.5.

[rte_sws_6703]⌈ The RTE Generator shall name role parameters according to the value of the symbol attribute of RunnableEntityArguments if RunnableEntityArguments are defined for the related RunnableEntity and if no mapping to a BswModuleEntry is defined. ⌋*(RTE00087)*

[rte_sws_6704]⌈ The RTE Generator shall name role parameters according to the shortName of the SwServiceArgs of the mapped BswModuleEntry if a mapping of the RunnableEntity to a BswModuleEntry is defined. ⌋*(RTE00087)*

Please note that RunnableEntityArguments defined for a RunnableEntity which is mapped to a BswModuleEntry are irrelevant.

[rte_sws_6705]⌈ The RTE Generator shall generate nameless role parameters if neither RunnableEntityArguments nor a mapping to a BswModuleEntry is defined for the RunnableEntity. ⌋*(RTE00087)*

Further details about the mapping of RunnableEntitys and BswModuleEntry can be found section "'Synchronization with a Corresponding SWC"' of the document [9]

5.7.4 Return Value

A function in C or C++ is required to have a return type. The RTE only uses the function return value to return application error codes of a server operation.

[rte_sws_1130]⌈ A function implementing a runnable entity entry point shall only have the return type Std_ReturnType, if the runnable entity represents a server operation and the AUTOSAR interface description of that client server communication lists potential application errors. All other functions implementing a runnable entity entry point shall have a return type of void. ⌋*(RTE00124, RTE00031)*

[rte_sws_ext_2704] Only the least significant six bit of the return value of a server runnable shall be used by the application to indicate an error. The upper two bit shall be zero. See also rte_sws_2573.

5.7.5 Triggering Events

The RTE is the *sole* entity that can trigger the execution of a runnable entity. The RTE triggers runnable entities in response to different RTEEvents.

The most basic RTEEvent that can trigger a runnable entity is the TimingEvent that causes a runnable entity to be periodically triggered by the RTE. In contrast, the remaining RTEEvents that can trigger runnable entities all occur as a result of communication activity or as a result of mode switches.

The following subsections describe the conditions that can trigger execution of a runnable entity. For each triggering event the signature of the function (the “entry point”) that implements the runnable entity is defined. The signature definition includes two classes of parameters for each function;

1. The instance handle – the parameter type is always Rte_Instance. (rte_sws_1016)
2. The role parameters – used to pass information required by the runnable entity as a consequence of the triggering condition. The presence (and number) of role parameters depends solely on the triggering condition.

5.7.5.1 TimingEvent

Purpose: Trigger a runnable entity periodically at a rate defined within the software-component description.

Signature: **[rte_sws_1131]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00072)*

5.7.5.2 BackgroundEvent

Purpose: A recurring RTEEvent which is used to perform background activities. It is similar to a TimingEvent but has no fixed time period and is activated only with low priority.

Signature: **[rte_sws_7177]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00072)*

5.7.5.3 SwcModeSwitchEvent

Purpose: Trigger of a runnable entity as a result of a mode switch. See also sections 4.4.4 and 4.4.7 for reference.

Signature: **[rte_sws_2512]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00072, RTE00143)*

5.7.5.4 AsynchronousServerCallReturnsEvent

Purpose: Triggers a runnable entity used to "collect" the result and status information of an asynchronous client-server operation.

Signature: **[rte_sws_1133]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00072, RTE00029, RTE00079)*

Notes: The runnable entity triggered by an AsynchronousServerCallReturnsEvent `RTEEvent` should use the `Rte_Result` API to actually receive the result and the status of the server operation.

5.7.5.5 DataReceiveErrorEvent

Purpose: Triggers a runnable entity used to "collect" the error status of a `data element` with "data" semantics on the receiver side.

Signature: **[rte_sws_1359]**⌈
```
void <name>([IN Rte_Instance <instance>])
```
⌋*(RTE00072, RTE00029, RTE00079)*

Notes: The runnable entity triggered by a DataReceiveErrorEvent `RTEEvent` should use the `Rte_IStatus` API to actually read the status.

5.7.5.6 OperationInvokedEvent

Purpose: An `RTEEvent` that causes the RTE to trigger a runnable entity whose entry point provides an implementation for a client-server operation. This event occurs in response to a received request from a client to execute the operation.

Signature: **[rte_sws_1166]**⌈
```
<void|Std_ReturnType> <name>
                        ([IN Rte_Instance <instance>],
                         [IN            <portDefArg 1>, ...
                          IN            <portDefArg n>],
                         [IN|INOUT|OUT] <param 1>, ...
                         [IN|INOUT|OUT] <param n>)
```

Where `<portDefArg 1>, ..., <portDefArg n>` represent the port-defined argument values (see Section 4.3.2.4) and `<param 1>, ... <param n>` indicates the operation IN, INOUT and OUT parameters. ⌋*(RTE00029, RTE00079, RTE00072, RTE00152)*

The data type of each port defined argument is taken from the software component template, as defined in `valueType`.

Note that the port-defined argument values are optional, depending upon the server's internal behavior.

[rte_sws_7023]⌈ The operation parameters `<param 1>, ... <param n>` are the specified `ArgumentDataPrototype`s of the `ClientServerOperation` that is associated with the `OperationInvokedEvent`. The operation parameters shall be ordered according to the `ClientServerOperation`'s

ordered list of the ArgumentDataPrototypes. ⌋*(RTE00029, RTE00079, RTE00072)*

[rte_sws_7024]⌈ If the ServerArgumentImplPolicy is set to useArgumentType the data type of the <param> is derived from the ArgumentDataPrototype's ImplementationDataType. ⌋*(RTE00029, RTE00079, RTE00072)*

In case of rte_sws_7024 the RunnableEntitys parameter are equally typed as the parameter for the Rte_Call API described in section 5.2.6.5

[rte_sws_7025]⌈ If the ServerArgumentImplPolicy is set to useArrayBaseType the data type of the <param> is derived from the ArgumentDataPrototype's ImplementationDataTypeElement specifying the base type of the array. ⌋*(RTE00029, RTE00079, RTE00072)*

ServerArgumentImplPolicy is set to useArrayBaseType is only applicable in case of ArgumentDataPrototype's which data type is of category ARRAY.

[rte_sws_7026]⌈ The RTE generator shall reject configuration where the ServerArgumentImplPolicy is set to useArrayBaseType for ArgumentDataPrototype's which data type is not of category ARRAY. ⌋*(RTE00029, RTE00079, RTE00072, RTE00018)*

[rte_sws_7027]⌈ If the ServerArgumentImplPolicy is set to useVoid the data type of the <param> is set to void in case of arguments typed by primitive data types and set to void * in case of arguments typed by composite data types. ⌋*(RTE00029, RTE00079, RTE00072)*

[rte_sws_5193]⌈ If the serverArgumentImplPolicy is set to useArrayBaseType or useVoid the RTE shall cast the arguments passed by Rte_Call() and Rte_Result() to the data types defined by the runnable entity prototype. ⌋*(RTE00029, RTE00079, RTE00072)*

Return Value: If the AUTOSAR interface description of the client server communication lists possible error codes, these are returned by the function using the return type Std_ReturnType. If no error codes are defined for this interface, the return type shall be void (see rte_sws_1130).

This means that even if a runnable entity implementing a server "only" returns E_OK, application errors have to be defined. Else the return types do not match.

5.7.5.7 DataReceivedEvent

Purpose: A runnable entity triggered by the RTE to receive and process a signal received on a sender-receiver interface.

Signature: **[rte_sws_1135]**⌈
```
void <name>([IN Rte_Instance <instance>])
```
⌋*(RTE00072, RTE00028, RTE00131, RTE00107)*

Notes: The data or event is not passed as an additional parameter. Instead, the previously described reception API should be used to access the data/event. This approach permits the same signature for runnables that are triggered by time (TimingEvent) or data reception.

Caution: For intra-ECU communication, the DataReceivedEvent is fired after each completed write operation to the shared data. In case of implicit access, write operation is considered to be completed when the runnable ends. While for inter-ECU communication, the DataReceivedEvent is fired by the RTE after a callback from COM due to data reception. Over a physical network, 'data' is commonly transmitted periodically and hence not only will the latency and jitter of DataReceivedEvents vary depending on whether a configuration uses intra or inter-ECU communication, but also the number and frequency of these RTEEvents may change significantly. This means that a TimingEvent should be used to periodically activation of a runnable rather than relying on the periodic transmission of data.

5.7.5.8 DataSendCompletedEvent

Purpose: A runnable entity triggered by the RTE to receive and process transmit acknowledgment notifications.

Signature: **[rte_sws_1137]**⌈
```
void <name>([IN Rte_Instance <instance>])
```
⌋*(RTE00072, RTE00122, RTE00107)*

Notes: The runnable entity triggered by a DataSendCompletedEvent `RTEEvent` should use the `Rte_Feedback` API to actually receive the status of the acknowledgement.

5.7.5.9 ModeSwitchedAckEvent

Purpose: A runnable entity triggered by the RTE to receive and process mode switched acknowledgement notifications.

Signature: **[rte_sws_2758]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00072, RTE00122, RTE00107)*

Notes: The runnable entity triggered by an `ModeSwitchedAckEvent` should use the `Rte_ModeSwitchAck` API to actually receive the status of the acknowledgement.

5.7.5.10 ExternalTriggerOccurredEvent

Purpose: A runnable entity triggered by the RTE at the occurrence of an external event.

Signature: **[rte_sws_7207]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00162, RTE00072)*

5.7.5.11 InternalTriggerOccurredEvent

Purpose: A runnable entity triggered by the RTE by an inter runnable trigger.

Signature: **[rte_sws_7208]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00163, RTE00072)*

5.7.5.12 DataWriteCompletedEvent

Purpose: A runnable entity triggered by the RTE to receive and process transmit acknowledgment notifications for implicit communication.

Signature: **[rte_sws_7379]**⌈
`void <name>([IN Rte_Instance <instance>])`

⌋*(RTE00072, RTE00122, RTE00185)*

Notes: The runnable entity triggered by a DataWriteCompletedEvent `RTEEvent` should use the `Rte_IFeedback` API to actually receive the status of the acknowledgement.

5.7.6 Reentrancy

A runnable entity is declared within a software-component type. The RTE ensures that concurrent activation of same instance of a runnable entity is only allowed if the runnables attribute "canBeInvokedConcurrently" is set to TRUE (see Section 4.2.6).

When a software-component is multiple instantiated each separate instance has its own instance of the runnable entities in the software-component. Whilst instances of a software-component are independent, the runnable entities instances share the same code (rte_sws_3015).

Example 5.30

Consider a component `c1` with runnable entity `re1` and entry point `ep` that is instantiated twice on the same ECU.

The two instances of `c1` each has a separate *instance* of `re1`. Software-component instances are scheduled independently and therefore each instance of `re1` could be concurrently executing `ep`.

The potential for concurrent execution of runnable entities when multiple instances of a software-component are created means that each entry point should be reentrant.

5.8 RTE Lifecycle API Reference

This section documents the API functions used to start and stop the RTE. RTE Lifecycle API functions are not invoked from AUTOSAR software-components – instead they are invoked from other basic software module(s).

5.8.1 Rte_Start

Purpose: Initialize the RTE itself.

Signature: **[rte_sws_2569]**⌈
`Std_ReturnType Rte_Start(void)`
⌋*(BSW00310, RTE00116)*

Existence: **[rte_sws_1309]**⌈ The `Rte_Start` API is always created. ⌋*(RTE00051)*

Description: `Rte_Start` is intended to allocate and initialise system resources and communication resources used by the RTE.

[rte_sws_ext_2582] `Rte_Start` shall be called only once by the EcuStateManager from trusted OS context on a core after the basic software modules required by RTE are initialized.

These modules include:

- OS
- COM
- memory services

The `Rte_Start` API shall not be invoked from AUTOSAR software components.

[rte_sws_ext_7577] The `Rte_Start` API may only be used after the *Basic Software Scheduler* is initialized (after termination of the `SchM_Init`).

[rte_sws_ext_2714] The `Rte_Start` API shall be called on every core that hosts AUTOSAR software-components of the ECU.

[rte_sws_2585]⌈ `Rte_Start` shall return within finite execution time – it must not enter an infinite loop. ⌋*(RTE00116)*

`Rte_Start` may be implemented as a function or a macro.

Return Value: If the allocation of a resource fails, `Rte_Start` shall return with an error.

- **[rte_sws_1261]**⌈ `RTE_E_OK` – No error occurred. ⌋*(RTE00094)*
- **[rte_sws_1262]**⌈ `RTE_E_LIMIT` – An internal limit has been exceeded. The allocation of a required resource has failed. ⌋*(RTE00094)*

Notes: `Rte_Start` is declared in the lifecycle header file `Rte_Main.h`. The initialization of AUTOSAR software-components takes place after the termination of `Rte_Start` and is triggered by a mode change event on entering run state.

5.8.2 Rte_Stop

Purpose: finalize the RTE itself

Signature: **[rte_sws_2570]**⌈

```
Std_ReturnType Rte_Stop(void)
```

⌋*(RTE00116)*

Existence: **[rte_sws_1310]**⌈ The `Rte_Stop` API is always created. ⌋*(RTE00051)*

Description: `Rte_Stop` is used to finalize the RTE on the core it is called. This service releases all system and communication resources allocated by the RTE on that core.

[rte_sws_ext_2583] `Rte_Stop` shall be called by the EcuStateManager before the basic software modules required by RTE are shut down. These modules include:

- OS
- COM
- memory services

`Rte_Stop` shall be called from trusted context and not by an AUTOSAR software component.

[rte_sws_2584]⌈ `Rte_Stop` shall return within finite execution time. ⌋*(RTE00116)*

`Rte_Stop` may be implemented as a function or a macro.

Return Value:

- **[rte_sws_1259]**⌈ `RTE_E_OK` – No error occurred. ⌋*(RTE00094)*
- **[rte_sws_1260]**⌈ `RTE_E_LIMIT` – a resource could not be released. ⌋*(RTE00094)*

Notes: `Rte_Stop` is declared in the lifecycle header file `Rte_Main.h`.

5.8.3 Rte_PartitionTerminated

Purpose: Indicate to the RTE that a partition is going to be terminated, and the communication with the Partition shall be ignored.

Signature: **[rte_sws_7330]**⌈
`void Rte_PartitionTerminated_<PID>(void)`

⌋*(RTE00223)*

Where `<PID>` is the name of the `EcucPartition` according to the ECU Configuration Description [15].

Existence: **[rte_sws_7331]**⌈ An `Rte_PartitionTerminated` API shall be created for every `Partition`. ⌋*(RTE00223)*

Description: `Rte_PartitionTerminated` is intended to notify the RTE that a given partition is terminated or is being restarted.

[rte_sws_ext_7332] `Rte_PartitionTerminated` shall be called only once by the ProtectionHook.

Rte_PartitionTerminated may be implemented as a function or a macro.

[rte_sws_7334]⌈ The treatments in Rte_PartitionTerminated shall be restricted to the ones allowed in the context of a ProtectionHook. ⌋*(RTE00223)*

Since Rte_PartitionTerminated is called from the ProtectionHook context, it should be as fast as possible. It should be limited to setting a flag. Actual cleanup should be deferred to another task.

The notification provided by Rte_PartitionTerminated can be used later by the RTE to immediately return an error status when SW-Cs of other partitions tries to communicate with the stopped partition. See rte_sws_2710 and rte_sws_2709.

[rte_sws_7335]⌈ Terminating an already terminated Partition shall be ignored. ⌋*(RTE00223)*

Return Value: None.

Notes: Rte_PartitionTerminated is declared in the lifecycle header file Rte_Main.h.

5.8.4 Rte_PartitionRestarting

Purpose: Indicate to the RTE that a Partition is going to be restarted and that the communication with the Partition shall be ignored.

Signature: **[rte_sws_7620]**⌈
`void Rte_PartitionRestarting_<PID>(void)`

Where <PID> is the name of the EcucPartition according to the ECU Configuration Description [15]. ⌋*(RTE00223)*

Existence: **[rte_sws_7619]**⌈ An Rte_PartitionRestarting API shall be created for any Partition which can be restarted (i.e. a Partition whose PartitionCanBeRestarted parameter is enabled). ⌋*(RTE00223)*

Description: Rte_PartitionRestarting is intended to notify the RTE that a given partition is being restarted. As Rte_PartitionTerminated, Rte_PartitionRestarting indicates that the communication with the partition shall be ignored, but in case of Rte_PartitionRestarting, the partition may be restarted later in the ECU lifecycle.

[rte_sws_ext_7618] Rte_PartitionRestarting shall be called only once by the ProtectionHook.

Rte_PartitionRestarting may be implemented as a function or a macro.

[rte_sws_7617]⌈ The treatments in Rte_PartitionRestarting shall be restricted to the ones allowed in the context of a ProtectionHook. ⌋*(RTE00223)*

Since Rte_PartitionRestarting is called from the ProtectionHook context, it should be as fast as possible. It should be limited to setting a flag. Actual cleanup should be deferred to another task.

[rte_sws_7622]⌈ Restarting an already terminated Partition or restarting a Partition during an ongoing restart shall be ignored. ⌋*(RTE00223)*

Return Value: None.

Notes: Rte_PartitionRestarting is declared in the lifecycle header file Rte_Main.h.

5.8.5 Rte_RestartPartition

Purpose: Initialize the RTE resources allocated for a partition.

Signature: **[rte_sws_7188]**⌈
Std_ReturnType Rte_RestartPartition_<PID>(void)

Where <PID> is the name of the EcucPartition according to the ECU Configuration Description [15]. ⌋*(RTE00224)*

Existence: **[rte_sws_7336]**⌈ An Rte_RestartPartition API shall be created for any Partition which can be restarted (i.e. a Partition whose PartitionCanBeRestarted parameter is enabled). ⌋*(RTE00224)*

Description: Rte_RestartPartition is intended to notify the RTE that a given partition will be restarted.

[rte_sws_ext_7337] Rte_RestartPartition shall be called only in the context of the RestartTask of the given partition.

[rte_sws_7338]⌈ Rte_RestartPartition shall return within finite execution time – it must not enter an infinite loop. ⌋*(RTE00224)*

Rte_RestartPartition may be implemented as a function or a macro.

[rte_sws_7339]⌈ The Rte_RestartPartition shall restore an initial RTE environment for the partition and re-activate communication with this partition. ⌋*(RTE00224)*

This includes:

- signal inital values,
- modes,
- queued events,
- sequence counters.

[rte_sws_7340]⌈ `Rte_RestartPartition` shall be ignored if the given partition was not stopped before (with `Rte_PartitionTerminated` or `Rte_PartitionRestarting`). ⌋*(RTE00224)*

Return Value: If the allocation of a resource fails, `Rte_RestartPartition` shall return with an error.

- **[rte_sws_7341]**⌈ `RTE_E_OK` – No error occurred. ⌋*(RTE00224)*
- **[rte_sws_7342]**⌈ `RTE_E_LIMIT` – An internal limit has been exceeded. The allocation of a required resource has failed. ⌋*(RTE00224)*

Notes: `Rte_RestartPartition` is declared in the lifecycle header file `Rte_Main.h`.

5.9 RTE Call-backs Reference

This section documents the call-backs that are generated by the RTE that must be invoked by other components, such as the communication service, and therefore must have a well-defined name and semantics.

[rte_sws_1165]⌈ A call-back implementation created by the RTE generator is not permitted to block. ⌋*(RTE00022)*

Requirement rte_sws_1165 serves to constrain RTE implementations so that all implementations can work with all basic software.

5.9.1 RTE-COM Message Naming Conventions

The COM signals used for communication are defined in the input information provided by Com.

[rte_sws_3007]⌈ The RTE shall initiate an inter-ECU transmission using the COM API with the handle id of the corresponding COM signal for primitive data element `SenderReceiverToSignalMapping`. ⌋*(RTE00019)*

[rte_sws_3008]⌈ The RTE shall initiate an inter-ECU transmission using the COM API with the handle id of the corresponding COM signal group for composite data elements or operation arguments `SenderReceiverToSignalGroupMapping`. ⌋*(RTE00019)*

5.9.2 Communication Service Call-backs

Purpose: Implement the call-back functions that AutoSAR COM invokes as a result of inter-ECU communication, where:

- A data item/event is ready for reception by a receiver.
- A transmission acknowledgment shall be routed to a sender.
- An operation shall be invoked by a server.
- The result of an operation is ready for reading by a client.

Signature: **[rte_sws_3000]**⌈

```
void <CallbackRoutineName> (void);
```

⌋*(RTE00019)*

Where `<CallbackRoutineName>` is the name of the call-back function (refer to Section 5.9.1 for details on the naming convention).

Description: Prototypes for the call-back `<CallbackRoutineName>` provided by AutoSAR COM.

Return Value: No return value : `void`

5.9.3 Naming convention of Communication Callbacks

In the following table, the naming convention of `<CallBackRoutineName>` are defined:

Calling Situation	callbackRoutineName	Comments
A primitive data item/event is ready for reception by a receiver.	**[rte_sws_3001]** `Rte_COMCbk_<sn>`	`<sn>` is the name of the COM signal. This callback function indicates that the signal of the primitive data item/event is ready for reception. Configured in Com: `ComNotification` [COM498_Conf] as part of `ComSignal`

Calling Situation	callbackRoutineName	Comments
A transmission acknowledgment of a primitive data item/event shall be routed to a sender.	**[rte_sws_3002]** `Rte_COMCbkTAck_<sn>`	"TAck" is literal text indicating transmission acknowledgment. This callback function indicates that the signal of the primitive data item/event is already handed over by COM to the PDU router. Configured in Com: `ComNotification` [COM498_Conf] as part of `ComSignal`
A transmission error notificatoin of a primitive data item/event shall be routed to a sender.	**[rte_sws_3775]** `Rte_COMCbkTErr_<sn>`	"TErr" is literal text indicating transmission error. This callback function indicates that an error occurred when the signal of the primitive data item/event was handed over by COM to the PDU router. Configured in Com: `ComErrorNotification` [COM499_Conf] as part of `ComSignal`
A signal invalidation of a primitive data item shall be routed to a receiver.	**[rte_sws_2612]** `Rte_COMCbkInv_<sn>`	"Inv" is literal text indicating signal invalidation. This callback function indicates that COM has received a signal and parsed it as "invalid". Configured in Com: `ComInvalidNotification` [COM315_Conf] as part of `ComSignal`
A signal of a primitive data item is outdated. No new data is available.	**[rte_sws_2610]** `Rte_COMCbkRxTOut_<sn>`	"RxTOut" is literal text indicating reception signal time out. This callback function indicates that the `aliveTimeout` after the last successful reception of the signal of the primitive data item/event has expired (`data element outdated`). Configured in Com: `ComTimeoutNotification` [COM552_Conf] as part of `ComSignal`

Calling Situation	callbackRoutineName	Comments
Transmission has failed and timed out for a primitive data item.	**[rte_sws_5084]** `Rte_COMCbkTxTOut_<sn>`	"TxTOut" is literal text indicating transmission failure and time out. This callback function indicates that the `timeout` of `TransmissionAcknowledgementRequest` for sending the signal of the primitive data item/event has expired. Configured in Com: `ComTimeoutNotification` [COM552_Conf] as part of `ComSignal`
A composite data item/event or the arguments of an operation is ready for reception by a receiver.	**[rte_sws_3004]** `Rte_COMCbk_<sg>`	`<sg>` is the name of the COM signal group, which contains all the signals of the composite data item/event or an operation. This callback function indicates that the signals of the composite data item/event or the arguments of an operation are ready for reception. Configured in Com: `ComNotification` [COM498_Conf] as part of `ComSignalGroup`
A transmission acknowledgment of a composite data item/event shall be routed to a sender.	**[rte_sws_3005]** `Rte_COMCbkTAck_<sg>`	"TAck" is literal text indicating transmission acknowledgment. This callback function indicates that the signals of the composite data item/event is already handed over by COM to the PDU router. Configured in Com: `ComNotification` [COM498_Conf] as part of `ComSignalGroup`

Calling Situation	callbackRoutineName	Comments
A transmission error notificatoin of a composite data item/event shall be routed to a sender.	**[rte_sws_3776]** `Rte_COMCbkTErr_<sg>`	"TErr" is literal text indicating transmission error. This callback function indicates that an error occurred when the signal of the composite data item/event was handed over by COM to the PDU router. Configured in Com: `ComErrorNotification` [COM499_Conf] as part of `ComSignalGroup`
A signal group invalidation of a composite data item shall be routed to a receiver.	**[rte_sws_5065]** `Rte_COMCbkInv_<sg>`	"Inv" is literal text indicating signal group invalidation. This callback function indicates that COM has received a signal group and parsed it as "invalid". Configured in Com: `ComInvalidNotification` [COM315_Conf] as part of `ComSignalGroup`
A signal group of a composite data item is outdated. No new data is available.	**[rte_sws_2611]** `Rte_COMCbkRxTOut_<sg>`	"RxTOut" is literal text indicating reception signal time out. This callback function indicates that the `aliveTimeout` after the last successful reception of the signal group carrying the composite data item has expired (`data element outdated`). Configured in Com: `ComTimeoutNotification` [COM552_Conf] as part of `ComSignalGroup`

Calling Situation	callbackRoutineName	Comments
Transmission has failed and timed out for a composite data item.	**[rte_sws_5085]** `Rte_COMCbkTxTOut_<sg>`	"TxTOut" is literal text indicating transmission failure and time out. This callback function indicates that the `timeout` of `TransmissionAcknowledgementRequest` for sending the signal group of the composite data item/event has expired. Configured in Com: `ComTimeoutNotification` [COM552_Conf] as part of `ComSignalGroup`

Table 5.5: RTE COM Callback Function Naming Conventions

Where:

- `<sn>` is a COM signal name.
- `<sg>` is a COM signal group name.

5.9.4 NVM Service Call-backs

5.9.4.1 Rte_SetMirror

Purpose: Warranty the consistency of the `VariableDataPrototype`s contained in a `NvBlockSwComponentType`, when the associated NVM block is read and copied to the `VariableDataPrototype`s storage locations.

Signature: **[rte_sws_7310]**⌈

```
Std_ReturnType
Rte_SetMirror_<b>_<d> (const void *NVMBuffer)
```

⌋*(RTE00178)*

Where `<b>` is the `SwComponentPrototype`'s name of the `NvBlockSwComponentType` and `<d>` is the `NvBlockDescriptor` name.

Existence: **[rte_sws_7311]**⌈ An `Rte_SetMirror` API shall be created for each instance of a `NvBlockDescriptor`. ⌋*(RTE00178)*

Description: The Rte_SetMirror API copies the values of the VariableDataPrototypes contained in a NvBlockDescriptor from a NVM internal buffer to their locations in the RTE.

[rte_sws_7312]⌈ The Rte_SetMirror API shall copy the specified buffer to the NvBlockDescriptor's ramBlock, according to the NvBlockDescriptor's NvBlockDataMapping. ⌋*(RTE00177)*

The RTE is responsible for ensuring the data consistency, see section 4.2.5 In particular for the NvBlockDescriptor, the Sender-Receiver ports, the Rte_SetMirror, and Rte_GetMirror may access concurrently the same VariableDataPrototypes.

[rte_sws_7319]⌈ The Rte_SetMirror API shall be callable before the Rte is started (with Rte_Start), and can rely on a running OS. ⌋*(RTE00178)*

Return Value: The NVM module uses the return value of the Rte_SetMirror API to check if the copy was successful. In case of failure, the NVM may retry later.

[rte_sws_7602]⌈ The Rte_SetMirror API shall return E_OK if the copy is successful. ⌋*(RTE00178)*

[rte_sws_7613]⌈ The Rte_SetMirror API shall return E_NOT_OK if the copy could not be performed. ⌋*(RTE00178)*

Notes: The NVM shall be configured to use this function when ReadBlock requests are processed (see NvmWriteRamBlockFromNvm in [23]).

5.9.4.2 Rte_GetMirror

Purpose: Warranty the consistency of the VariableDataPrototypes contained in a NvBlockSwComponentType, when their values are written to the NVRAM device by the NVM.

Signature: **[rte_sws_7315]**⌈

```
Std_ReturnType
Rte_GetMirror_<b>_<d> (void *NVMBuffer)
```

⌋*(RTE00178)*

Where <b> is the SwComponentPrototype's name of the NvBlockSwComponentType and <d> is the NvBlockDescriptor name.

Existence: **[rte_sws_7316]**⌈ An Rte_GetMirror API shall be created for each instance of a NvBlockDescriptor. ⌋*(RTE00178)*

Description: The Rte_GetMirror API copies the values of the VariableDataPrototypes contained in a NvBlockDescriptor to a specified NVM internal buffer.

[rte_sws_7317]⌈ The Rte_GetMirror API shall copy the NvBlockDescriptor's ramBlock to the specified buffer, according to the NvBlockDescriptor's NvBlockDataMapping. ⌋*(RTE00177)*

The RTE is responsible for ensuring the data consistency, see section 4.2.5 In particular for the NvBlockDescriptor, the SenderReceiver ports, the Rte_SetMirror, and Rte_GetMirror may access concurrently the same VariableDataPrototypes.

[rte_sws_7350]⌈ The Rte_GetMirror API shall be callable after the Rte is stopped (with Rte_Stop), and can rely on a running OS. ⌋*(RTE00178)*

Return Value: The NVM module uses the return value of the Rte_GetMirror API to check if the copy was successful. In case of failure, the NVM may retry later.

[rte_sws_7601]⌈ The Rte_GetMirror API shall return E_OK if the copy is successful. ⌋*(RTE00178)*

[rte_sws_7614]⌈ The Rte_GetMirror API shall return E_NOT_OK if the copy could not be performed. ⌋*(RTE00178)*

Notes: The NVM shall be configured to use this function when WriteBlock requests are processed (see NvmWriteRamBlockToNvm in [23]).

5.9.4.3 Rte_NvMNotifyJobFinished

Purpose: Forward notifications back to the SW-Cs.

Signature: **[rte_sws_7623]**⌈

```
Std_ReturnType
Rte_NvMNotifyJobFinished_<b>_<d> (
    uint8 ServiceId,
    NvM_RequestResultType JobResult)
```

⌋*(RTE00228)*

Where <b> is the SwComponentPrototype's name of the NvBlockSwComponentType and <d> is the NvBlockDescriptor name.

Existence: **[rte_sws_7624]**⌈ An Rte_NvMNotifyJobFinished API shall be created for each instance of a NvBlockDescriptor. ⌋*(RTE00228)*

Description: The `Rte_NvMNotifyJobFinished` receives the notification from the NvM when a job is finished and forward it to the SW-C.

[rte_sws_7625]⌈ The `Rte_NvMNotifyJobFinished` API shall call the servers referenced by `RoleBasedPortAssignment` with a `NvMNotifyJobFinished` role which are aggregated to the `NvBlockDescriptor`.⌋*(RTE00228)*

[rte_sws_7671]⌈ The `Rte_NvMNotifyJobFinished` API shall return without any action when the RTE is not started, when the RTE is stopped, or when the partition containing the `NvBlockSwComponentType` is terminated or restarting.⌋*(RTE00228)*

Return Value: **[rte_sws_7626]**⌈ The `Rte_NvMNotifyJobFinished` API shall return `E_OK`.⌋*(RTE00228)*

Notes: The NVM shall be configured to use this function (see `NvmSingleBlockCallback` in [23]).

5.9.4.4 Rte_NvMNotifyInitBlock

Purpose: Indicate to the SW-Cs that initialization of the Mirror is requested by the NvM.

Signature: **[rte_sws_7627]**⌈

```
Std_ReturnType
Rte_NvMNotifyInitBlock_<b>_<d> (void)
```

⌋*(RTE00228)*

Where `<b>` is the `SwComponentPrototype`'s name of the `NvBlockSwComponentType` and `<d>` is the `NvBlockDescriptor` name.

Existence: **[rte_sws_7628]**⌈ An `Rte_NvMNotifyInitBlock` API shall be created for each instance of a `NvBlockDescriptor`.⌋*(RTE00228)*

Description: The `Rte_NvMNotifyInitBlock` API receives the notification from the NvM when initialization of the mirror is requested.

[rte_sws_7629]⌈ If the `NvBlockDescriptor` is configured with a `romBlock initValue`, this `initValue` shall be copied into the `NvBlockDescriptor`'s mirror before calling any SW-C server.⌋*(RTE00228)*

[rte_sws_7630]⌈ The `Rte_NvMNotifyInitBlock` API shall call the servers referenced by `RoleBasedPortAssignment` with a

NvMNotifyInitBlock role which are aggregated to the NvBlock-Descriptor. ⌋*(RTE00228)*

[rte_sws_7672]⌈ The Rte_NvMNotifyInitBlock API shall return without any action when the RTE is not started, when the RTE is stopped, or when the partition containing the NvBlockSwCompo-nentType is terminated or restarting. ⌋*(RTE00228)*

Due to rte_sws_7672, a block selected in the NVRAM Manager [23] as read during NvM_ReadAll should not be configured with its NvmInitBlockCallback set to a Rte_NvMNotifyInitBlock API.

Return Value: **[rte_sws_7631]**⌈ The Rte_NvMNotifyInitBlock API shall return E_OK. ⌋*(RTE00228)*

Notes: The NVM shall be configured to use this function (see NvmInitBlockCallback in [23]).

5.10 Expected interfaces

5.10.1 Expected Interfaces from Com

The specification of the RTE requires the usage of the following COM API functions.

Com API function	Context
Com_SendSignal	to transmit a data element of primitive type using COM.
Com_SendDynSignal	to transmit a data element of primitive dynamic type uint8[n] using COM.
Com_ReceiveSignal	to retrieve the new value of a data element of primitive type from COM.
Com_ReceiveDynSignal	to retrieve the new value of a data element of primitive dynamic type uint[8] from COM.
Com_UpdateShadowSignal (deprecated)	to update a primitive element of a data element of composite type in preparation for sending the composite type using COM.
Com_SendSignalGroup	to initiate sending of a data element of composite type using COM.
Com_ReceiveSignalGroup	to retrieve the new value of a data element of composite type from COM.
Com_ReceiveShadowSignal (deprecated)	to retrieve the new value of a primitive element of a data element of composite type from COM.
Com_InvalidateSignal	to invalidate a data element of primitive type using COM.
Com_InvalidateSignalGroup	to invalidate a whole signal group using COM.

Com API function	Context

Table 5.6: COM API functions used by the RTE

Please note that rte_sws_2761 may require to access COM through the use of call trusted function in a partitioned system.

5.10.2 Expected Interfaces from Os

The usage of APIs provided by the Os module [4] is up to the implementation of a specific RTE Generator, System description and Ecu configuration. In general a RTE may utilize any standardized API. Therefore no dedicated list of expected APIs is specified here.

In case of multi-core the RTE may utilize the *IOC*-Module [12] to implement the inter-core communication. The *IOC*-Module is specified to be part of the Os. Therefore no specific APIs are listed here.

5.11 VFB Tracing Reference

The RTE's "VFB Tracing" functionality permits the monitoring of AUTOSAR signals as they are sent and received across the VFB.

The RTE operates in at least two builds (some implementations may provide more than two builds). The first, production, does not enable VFB tracing whereas the second, debug, can be configured to trace some or all "interesting events".

[rte_sws_1327]⌈ The RTE generator shall support a build where no VFB events are traced. ⌋*(RTE00005)*

[rte_sws_1328]⌈ The RTE generator shall support a build that traces (configured) VFB events. ⌋*(RTE00005)*

The RTE generator's 'trace' build is enabled or disabled through definitions in the RTE Configuration Header File rte_sws_1322 and rte_sws_1323. Note that this 'trace' build is intended to enable debugging of software components and not the RTE itself.

5.11.1 Principle of Operation

The "VFB Tracing" mechanism is designed to offer a lightweight means to monitor the interactions of AUTOSAR software-components with the VFB.

The VFB tracing in 'debug' build is implemented by a series of "hook" functions that are invoked automatically by the generated RTE when "interesting events" occur. Each hook function corresponds to a single event.

The supported trace events are defined in Section 5.11.4. A mechanism is described in Section 5.11.5 for configuring which of the many potential trace events are of interest.

5.11.2 Support for multiple clients

The "VFB Tracing" mechanism is designed to support multiple clients for each trace event.

[rte_sws_5093]⌈ For each `RteVfbTraceClientPrefix` configured in the RTE Configuration input each Trace Event shall be generated using that *client prefix* in the optional `<client>` position of the API function name. ⌋*(RTE00005, RTE00008, RTE00192)*

[rte_sws_5091]⌈ The RTE Generator shall provide each Trace Event without a *client prefix*. ⌋*(RTE00005, RTE00008, RTE00192)*

The generation of Trace Events without a *client prefix* ensures compatibility of the trace events with previous RTE releases.

[rte_sws_5092]⌈ In case of multiple clients for one Trace Event the individual trace functions shall be called in the following order:

1. The trace function without *client prefix*.
2. The trace functions with *client prefix* in alphabetically ascending order of the `RteVfbTraceClientPrefix` (ASCII / ISO 8859-1).

⌋*(RTE00005, RTE00008, RTE00192)*

The calling order specification ensures a deterministic execution of the multiple clients.

One example of the usage of *client prefix* is the parallel usage of Debugging [27] and Diagnostic Log and Trace [30]. In this example two `RteVfbTraceClientPrefix` would be specified:

- `Dbg`
- `Dlt`

This shall result in the declaration of three trace functions for the one Trace Event `Rte_[<client>_]Task_Activate(TaskType task)`:

- `Rte_Task_Activate(TaskType task)`
- `Rte_Dbg_Task_Activate(TaskType task)`
- `Rte_Dlt_Task_Activate(TaskType task)`

These trace functions (if all used in one project) will be called in the following order:

1. `Rte_Task_Activate(TaskType task)`
2. `Rte_Dbg_Task_Activate(TaskType task)`

3. `Rte_Dlt_Task_Activate(TaskType task)`

5.11.3 Contribution to the Basic Software Module Description

The RTE Generator in Generation Phase shall also update its Basic Software Module Description (rte_sws_5086) in order to document the possibly traceable functions and their signatures.

[rte_sws_5106]⌈ For each generated hook function - including multiple trace clients (rte_sws_5093) - an entry in the Basic Software Module Description shall be entered describing the hook function and its signature. The `outgoingCallback` element of `BswModuleDescription` shall be used to capture the information. ⌋*(RTE00005, RTE00192)*

5.11.4 Trace Events

5.11.4.1 RTE API Trace Events

RTE API trace events occur when an AUTOSAR software-component interacts with the generated RTE API. For implicit S/R communication, however, tracing is not supported.

5.11.4.1.1 RTE API Start

Description: RTE API Start is invoked by the RTE when an API call is made by a component.

Signature: **[rte_sws_1238]**⌈

```
void Rte_[<client>_]<api>Hook_<cts>_<ap>_Start
     ([const Rte_CDS_<cts>*, ]<param>)
```

Where `<api>` is the RTE API Name (Write, Call, etc.),

`<cts>` is the `component type symbol` of the `AtomicSwComponentType` and

`<ap>` the access point name (e.g. port and data element or operation name, exclusive area name, etc.).

The parameters of the API are the same as the corresponding RTE API. As with the API itself, the instance handle is included if and only if the software component's `supportsMultipleInstantiation` attribute is set to true. Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous. ⌋*(RTE00045, RTE00003, RTE00004)*

5.11.4.1.2 RTE API Return

Description: RTE API Return is a trace event that is invoked by the RTE just before an API call returns control to a component.

Signature: **[rte_sws_1239]**⌈

```
void Rte_[<client>_]<api>Hook_<cts>_<ap>_Return
     ([const Rte_CDS_<cts>*, ]<param>)
```

Where <api> is the RTE API Name (Write, Call, etc.),

<cts> is the component type symbol of the AtomicSwComponentType and

<ap> the access point name (e.g. port and data element or operation name, exclusive area name, etc.).

The parameters of the API are the same as the corresponding RTE API and contain the values of OUT and INOUT parameters on exit from the function. ⌋*(RTE00045)*

As with the API itself, the instance handle is included if and only if the software component's supportsMultipleInstantiation attribute is set to true. Note that Rte_Instance cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and Rte_Instance would therefore be ambiguous.

5.11.4.2 COM Trace Events

COM trace events occur when the generated RTE interacts with the AUTOSAR communication service.

5.11.4.2.1 Signal Transmission

Description: A trace event indicating a transmission request of an Inter-ECU signal (or signal in a signal group) by the RTE. Invoked by the RTE just before Com_SendSignal, Com_SendDynSignal, or Com_UpdateShadowSignal (deprecated) is invoked.

Signature: **[rte_sws_1240]**⌈

```
void Rte_[<client>_]ComHook_<signalName>_SigTx
    (<data>[, <length>])
```

Where <signalName> is the COM signal name, <data> is a pointer to the signal data to be transmitted, and <length> is the length of the signal in case of a dynamic signal. ⌋*(RTE00045, RTE00003, RTE00004)*

5.11.4.2.2 Signal Reception

Description: A trace event indicating a successful attempt to read an Inter-ECU signal (or signal in a signal group) by the RTE. Invoked by the RTE after return from `Com_ReceiveSignal`, `Com_ReceiveDynSignal`, or `Com_ReceiveShadowSignal` (deprecated).

Signature: **[rte_sws_1241]**⌈

```
void Rte_[<client>_]ComHook_<signalName>_SigRx
    (<data>[, <length>])
```

Where `<signalName>` is the COM signal name, `<data>` is a pointer to the signal data received, and `<length>` is a pointer where the length of the dynamic signal is copied in case of a dynamic signal. ⌋*(RTE00045, RTE00003, RTE00004)*

5.11.4.2.3 Signal Invalidation

Description: A trace event indicating a signal invalidation request of an Inter-ECU signal (or of a signal in a signal group) by the RTE. Invoked by the RTE just before `Com_InvalidateSignal` (if parameter `RteUseComShadowSignalApi` is `FALSE`), or `Com_InvalidateShadowSignal` (if parameter `RteUseComShadowSignalApi` is `TRUE`) is invoked.

Signature: **[rte_sws_3814]**⌈

```
void Rte_[<client>_]ComHook_<signalName>_SigIv
    (void)
```

Where `<signalName>` is the COM signal or a signal group name. ⌋*(RTE00045, RTE00003, RTE00004)*

5.11.4.2.4 Signal Group Invalidation

Description: A trace event indicating a signal group invalidation request of an Inter-ECU signal group by the RTE. Invoked by the RTE just before `Com_InvalidateSignalGroup` is invoked.

Signature: **[rte_sws_7639]**⌈

```
void Rte_[<client>_]ComHook_<signalGroupName>_SigGroupIv
    (void)
```

Where `<signalGroupName>` is the name of the signal group. ⌋*(RTE00045, RTE00003, RTE00004)*

5.11.4.2.5 COM Callback

Description: A trace event indicating the start of a COM call-back. Invoked by generated RTE code on entry to the COM call-back.

Signature: **[rte_sws_1242]**⌈

```
void Rte_[<client>_]ComHook<Event>_<signalName>
    (void)
```

Where `<signalName>` is the name of the COM signal or signal group and `<Event>` indicates the callback type and can take the values

- "Rx" for a reception indication callback
- "Inv" for an invalidation callback
- "RxTOut" for a reception timeout callback
- "TxTOut" for a transmission timeout callback
- "TAck" for a transmission acknowledgement callback
- "TErr" for a transmission error callback

⌋*(RTE00045, RTE00003, RTE00004)*

5.11.4.3 OS Trace Events

OS trace events occur when the generated RTE interacts with the AUTOSAR operating system.

5.11.4.3.1 Task Activate

Description: A trace event that is invoked by the RTE immediately prior to the activation of a task containing runnable entities.

Signature: **[rte_sws_1243]**⌈

```
void Rte_[<client>_]Task_Activate(TaskType task)
```

Where `task` is the OS's handle for the task. ⌋*(RTE00045)*

5.11.4.3.2 Task Dispatch

Description: A trace event that is invoked immediately an RTE generated task (containing runnable entities) has commenced execution.

Signature: **[rte_sws_1244]**⌈

```
void Rte_[<client>_]Task_Dispatch(TaskType task)
```

Where `task` is the OS's handle for the task. ⌋*(RTE00045)*

5.11.4.3.3 Set OS Event

Description: A trace event invoked immediately before generated RTE code attempts to set an OS Event.

Signature: **[rte_sws_1245]**⌈

```
void Rte_[<client>_]Task_SetEvent(TaskType task,
                                  EventMaskType ev)
```

Where `task` is the OS's handle for the task for which the event is being set and `ev` the OS event mask. ⌋*(RTE00045)*

5.11.4.3.4 Wait OS Event

Description: Invoked immediately before generated RTE code attempts to wait on an OS Event. This trace event does *not* indicate that the caller has suspended execution since the OS call may immediately return if the event was already set.

Signature: **[rte_sws_1246]**⌈

```
void Rte_[<client>_]Task_WaitEvent(TaskType task,
                                   EventMaskType ev)
```

Where `task` is the OS's handle for the task (that is waiting for the event) and `ev` the OS event mask. ⌋*(RTE00045)*

5.11.4.3.5 Received OS Event

Description: Invoked immediately after generated RTE code returns from waiting on an event.

Signature: **[rte_sws_1247]**⌈

```
void Rte_[<client>_]Task_WaitEventRet(TaskType task,
                                      EventMaskType ev)
```

Where `task` is the OS's handle for the task (that was waiting for an event) and `ev` the event mask indicating the received event. ⌋*(RTE00045)*

Note that not all of the trace events listed above may be available for a given input configuration. For example if a task is activated by a schedule table, it is activated by

the OS rather than by the RTE, hence no trace hook function for task activation can be invoked by the RTE.

5.11.4.4 Runnable Entity Trace Events

Runnable entity trace events occur when a runnable entity is started.

5.11.4.4.1 Runnable Entity Invocation

Description: Event invoked by the RTE just before execution of runnable entry starts via its entry point. This trace event occurs after any copies of data elements are made to support the `Rte_IRead` API Call.

Signature: **[rte_sws_1248]**⌈

```
void Rte_[<client>_]Runnable_<cts>_<reName>_Start
     ([const RTE_CDS_<cts>*])
```

Where `<cts>` is the `component type symbol` of the `AtomicSwComponentType`

and `reName` the runnable entity name.

The instance handle is included if and only if the software component's `supportsMultipleInstantiation` attribute is set to true. Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous. ⌋*(RTE00045)*

5.11.4.4.2 Runnable Entity Termination

purpose: Event invoked by the RTE immediately execution returns to RTE code from a runnable entity. This trace event occurs before any write-back of data elements are made to support the `Rte_IWrite` API Call.

Signature: **[rte_sws_1249]**⌈

```
void Rte_[<client>_]Runnable_<cts>_<reName>_Return
     ([const Rte_CDS_<cts>*])
```

Where `<cts>` is the `component type symbol` of the `AtomicSwComponentType`

and `reName` the runnable entity name.

The instance handle is included if and only if the software component's `supportsMultipleInstantiation` attribute is set to true.

Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous. ⌋*(RTE00045)*

5.11.5 Configuration

The VFB tracing mechanism works by the RTE invoking the tracepoint *hook* function whenever the tracing event occurs.

The support trace events and their hook function name and signature are defined in Section 5.11.4. There are many potential trace events and it is likely that only a few will be of interest at any one time. Therefore The RTE generator supports a mechanism to configure which trace events are of interest.

In order to minimize RTE Overheads, trace events that are not enabled should have no run-time effect on the generated system. This is achieved through generated code within the VFB Tracing Header File (see Section 5.3.7) and the user supplied definitions from the RTE Configuration Header file (see Section 5.3.8).

The definition of trace event hook functions is contained within user code. If a definition is encapsulated within a `#if` block, as follows, the definition will automatically be omitted when the trace event is disabled.

```
#if !defined(<trace event>)
void <trace event>(<params>)
{
  /* Function definition */
}
#endif
```

The configuration of which individual trace events are enabled is entirely under the control of the user via the definitions included in the RTE Configuration header file.

[rte_sws_8000]⌈ When RteVfbTrace is set to "true", a user shall be able to enable any hook function in the RTE Configuration header file, regardless of whether it was not enabled in the RTE configuration with a `RteVfbTraceFunction` parameter. ⌋*(RTE00005, RTE00008)*

5.11.6 Interaction with Object-code Software-Components

VFB tracing is only available during the "RTE Generation" phase rte_sws_1319 and therefore hook functions never appear in an application header file created during "RTE Contract" phase. However, object-code software-components are compiled against the "RTE Contract" phase header and can therefore only trace events that are inserted into the generated RTE. In particular they cannot trace events that require invocation of hook functions to be inserted into the API mapping such as the `Rte_Pim` API. However,

many trace events are applicable to object-code software-components including trace events related to the explicit communication API, to task activity and for runnable entity start and stop.

This approach means that the external interactions of the object-code software-component can be monitored without requiring modification of the delivered object-code and without revealing the internal activity of the software-component. The approach is therefore considered to be consistent with the desire for IP protection that prompts delivery of a software-component as object-code. Finally, tracing can easily be disabled for a production build without invalidating tests of the object-code software-component.

6 Basic Software Scheduler Reference

6.1 Scope

This chapter presents the *Basic Software Scheduler* API from the perspective of *AUTOSAR Basic Software Module* – these API is not applicable for *AUTOSAR software-components*.

Section 6.2 presents basic principles of the API including naming conventions and supported programming languages. Section 6.3 describes the header files used by the *Basic Software Scheduler* and the files created by an RTE generator. The data types used by the API are described in Section 6.4 and Sections 6.5 and 6.6 provide a reference to the *Basic Software Scheduler* API itself including the definition of *Basic Software Module Entities*.

6.2 API Principles

6.2.1 Basic Software Scheduler Namespace

The *Basic Software Scheduler* is interleaved with the scheduling part of the *RTE*. Further on it is generated by the *RTE Generator* together with the *RTE* so *Basic Software Scheduler* and *RTE* can not be separated if both are generated. Therefore the *Basic Software Scheduler* uses the namespace of the *RTE* for internal symbols, variables and functions, see rte_sws_1171.

The only exceptions are defines, data types and functions belonging to the interface of the *Basic Software Scheduler*. These are explicitly mentioned in the specification.

[rte_sws_7284]⌈ All Basic Software Scheduler symbols (e.g. function names, data types, etc.) belonging to the *Basic Software Scheduler*s interfaces are required to use the `SchM_` prefix. ⌋*(BSW00307, BSW00300, RTE00055)*

In case of *Basic Software Module*s supporting multiple instances of the same *Basic Software Module* the name space of the `BswSchedulableEntity`s and the *Basic Software Scheduler* API related to one instance of a *Basic Software Module* is extended by the `vendorId` and the `vendorApiInfix`. See document [31] [BSW00347]. In the following chapters this optional part is denoted by usage of squared brackets `[_<vi>_<ai>]`.

[rte_sws_7528]⌈ If the attribute `vendorApiInfix` exists for a *Basic Software Module*, the RTE generator shall insert the `vendorId` (`<vi>`) and the `vendorApiInfix` (`<ai>`) with leading underscores where it is denoted by `[_<vi>_<ai>]`. ⌋*(BSW00347)*

6.2.2 BSW Scheduler Name Prefix and Section Name Prefix

Since the Basic Software Module Description supports the description of BSW Module Clusters one *Basic Software Module Description* can contain the content of several BSW Modules. In order to fulfill the Standardized Interfaces with the cluster interface different ICC3 *Module abbreviation*s [9] inside one cluster can occur. For the Basic Software Scheduler the *Module abbreviation* is used as *BSW Scheduler Name Prefix* in the SchM API. Nevertheless the `shortName` of the `BswModuleDescription` can as well describe the *BSW Scheduler Name Prefix* and *Section Name Prefix* in order to provide one common prefix in case of ICC3 modules.

In the Meta Model *Module abbreviation*s relevant for the Schedule Manager API are explicitly expressed with the meta class `BswSchedulerNamePrefix`. Further information can be found in document [9].

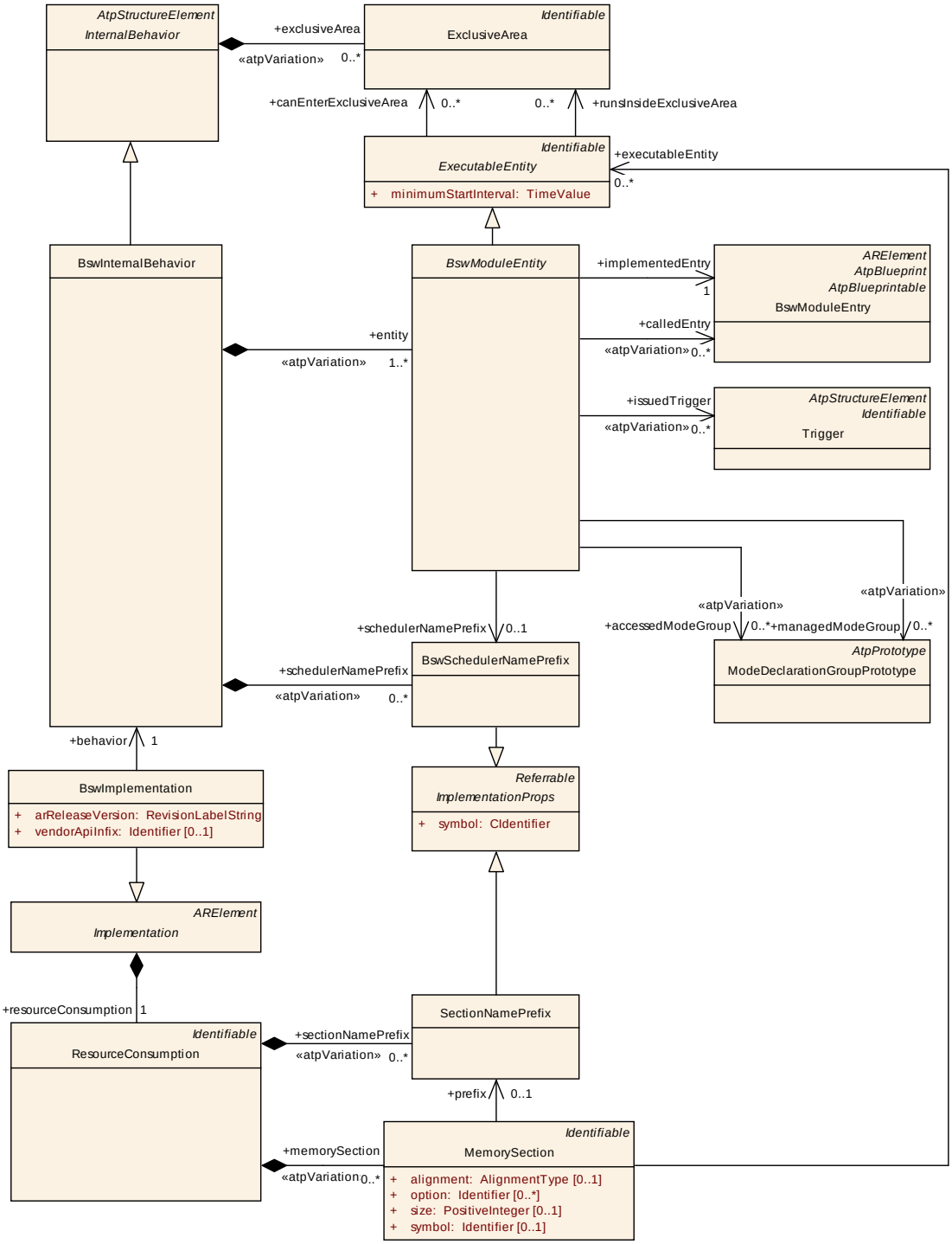


Figure 6.1: `BswSchedulerNamePrefix` and `SectionNamePrefix`

In several requirements of this specification the *Module Prefix* is required and determined as follows:

[rte_sws_7593]⌈ The *BSW Scheduler Name Prefix* <bsnp> of the calling BSW module shall be derived from the `BswModuleDescription` `shortName` if no `BswSched-`

ulerNamePrefix is defined for the BswModuleEntity using the related Basic Software Scheduler API. ⌋*(RTE00148, RTE00149)*

[rte_sws_7594]⌈ The *BSW Scheduler Name Prefix* <bsnp> shall be the value of the symbol attribute of the BswSchedulerNamePrefix of the BswModuleEntity if a BswSchedulerNamePrefix is defined for the BswModuleEntity using the related Basic Software Scheduler API. ⌋*(RTE00148, RTE00149)*

Further on the *Memory Mapping* inside one cluster can either keep or abolish the ICC3 borders. For some cases (e.g. *Entry Point Prototype*) the RTE has to know the used prefixes for the *Memory Allocation Keyword*s as well.

In the Meta Model these prefixes are expressed with the meta class SectionNamePrefix. Further information can be found in document [9].

[rte_sws_7595]⌈ The *Section Name Prefix* <snp> shall be the module abbreviation (in uppercase letters) of the BSW module derived from the BswModuleDescription's shortName if no SectionNamePrefix is defined for the BswModuleEntity implementing the related BswModuleEntry. ⌋*(RTE00148, RTE00149)*

[rte_sws_7596]⌈ The *Section Name Prefix* <snp> shall be the symbol of the SectionNamePrefix of the MemorySection associated to the BswModuleEntity implementing the related BswModuleEntry if a SectionNamePrefix is defined for the BswModuleEntity implementing the related BswModuleEntry. ⌋*(RTE00148, RTE00149)*

For instance the following input configuration

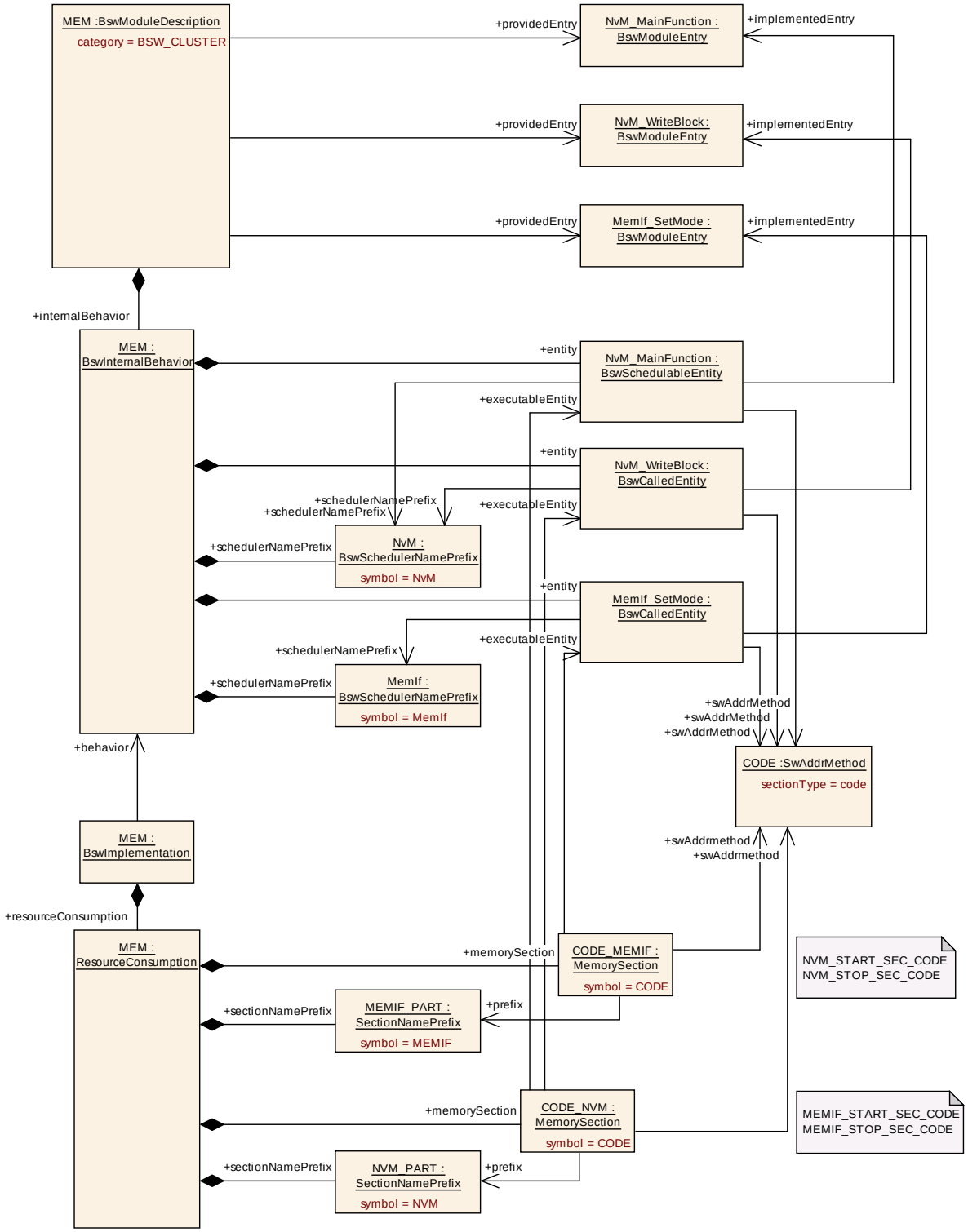


Figure 6.2: Example of ICC2 cluster

would result in the generation of the *Entry Point Prototype* according rte_sws_7195 as:

```
1  #define NVM_START_SEC_CODE
2  #include "MemMap.h"
3
```

```
4 FUNC(void, NVM_CODE) NvM_MainFunction (void);
5
6 #define NVM_STOP_SEC_CODE
7 #include "MemMap.h"
```

6.3 Basic Software Scheduler modules

[rte_sws_7288]⌈ Every file of the *Basic Software Scheduler* shall be named with the prefix `SchM_`. ⌋*(BSW00300)*

6.3.1 Module Interlink Types Header

The *Module Interlink Types Header* defines specific types related to this basic software module derived either from the input configuration or from the RTE / Basic Software Scheduler implementation.

[rte_sws_7503]⌈ The RTE generator shall create a *Module Interlink Types Header File* for each `BswSchedulerNamePrefix` in the `BswInternalBehavior` of each `BswImplementation` referencing such `BswInternalBehavior` defined in the input. ⌋*(BSW00415)*

For instance a input configuration with two `BswImplementation`s (typical with different API infix) referencing a `BswInternalBehavior` with three `BswScheduler-NamePrefixe`s would result in the generation of six Module Interlink Types Header Files.

6.3.1.1 File Name

[rte_sws_7295]⌈ The name of the *Module Interlink Types Header File* shall be formed in the following way:

`SchM_<bsnp>_[<vi>_<ai>]Type.h`

Where here

`<bsnp>` is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

`<vi>` is the `vendorId` of the BSW module and

`<ai>` is the `vendorApiInfix` of the BSW module.

The sub part in squared brackets [<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(BSW00415, BSW00300, BSW00347)*

Example 6.1

The following declaration in the input XML:

```
<AR-PACKAGE>
  <SHORT-NAME>CanDriver</SHORT-NAME>
  <ELEMENTS>
    <BSW-MODULE-DESCRIPTION>
      <SHORT-NAME>Can</SHORT-NAME>
      <INTERNAL-BEHAVIORS>
        <BSW-INTERNAL-BEHAVIOR>
          <SHORT-NAME>YesWeCan</SHORT-NAME>
        </BSW-INTERNAL-BEHAVIOR>
      </INTERNAL-BEHAVIORS>
    </BSW-MODULE-DESCRIPTION>
    <BSW-IMPLEMENTATION>
      <SHORT-NAME>MyCanDrv</SHORT-NAME>
      <VENDOR-ID>25</VENDOR-ID>
      <BEHAVIOR-REF DEST="BSW-INTERNAL-BEHAVIOR">/CanDriver/Can/
          YesWeCan</BEHAVIOR-REF>
      <VENDOR-API-INFIX>Dev0815</VENDOR-API-INFIX>
    </BSW-IMPLEMENTATION>
  </ELEMENTS>
</AR-PACKAGE>
```

should result in the *Module Interlink Types Header* SchM_Can_25_Dev0815Type.h being generated.

The concatenation of the basic software module prefix (which has to be equally with the short name of the basic software module description) and the vendor API infix is required to support the separation of several basic software module instances. In difference to the multiple instantiation concept of software components, where the same component code is used for all component instances, basic software modules are multiple instantiated by creation of own code per instance in a different name space.

6.3.1.2 Scope

[rte_sws_7296]⌈ The *Module Interlink Types Header* for a module shall contain only data types relevant for that instance of a basic software module. ⌋*(BSW00415)*

Requirement rte_sws_7296 means that compile time checks ensure that a *Module Interlink Header File* that uses the *Module Interlink Types Header File* only accesses

the generated data types to which it has been configured. The use of data types which are not used by the basic software module, will fail with a compiler error [RTE00017].

[rte_sws_7297]⌈ The *Module Interlink Types Header* shall be valid for both C and C++ source. ⌋*(RTE00126, RTE00138)*

Requirement rte_sws_7297 is met by ensuring that all definitions within the *Application Types Header File* are defined using C linkage if a C++ compiler is used.

[rte_sws_7298]⌈ All definitions within in the *Module Interlink Types Header File* shall be preceded by the following fragment:

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

[rte_sws_7299]⌈ All definitions within the *Module Interlink Types Header* shall be suffixed by the following fragment:

```
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

6.3.1.3 File Contents

[rte_sws_7500]⌈ The *Module Interlink Types Header* shall include the *RTE Types Header File*. ⌋*(BSW00415)*

The name of the *RTE Types Header File* is defined in Section 5.3.4.

6.3.1.4 Basic Software Scheduler Modes

The *Module Interlink Types Header File* shall contain identifiers for the `ModeDeclarations` and type definitions for `ModeDeclarationGroups` as defined in Chapter 6.4.2

6.3.2 Module Interlink Header

The *Module Interlink Header* defines the *Basic Software Scheduler* API and any associated data structures that are required by the *Basic Software Scheduler* implementation. But the *Module Interlink Header* file is not allowed to create objects in memory.

[rte_sws_7501]⌈ The RTE generator shall create a *Module Interlink Header File* for each `BswSchedulerNamePrefix` in the `BswInternalBehavior` of each

BswImplementation referencing such BswInternalBehavior defined in the input.⌋*(BSW00415)*

[rte_sws_ext_7512] Each BSW module implementation shall include its *Module Interlink Header File* if it uses *Basic Software Scheduler* API or if it implements BswSchedulableEntitys.

[rte_sws_7502]⌈ The *Module Interlink Header File* shall not contain code that creates objects in memory. ⌋*(BSW00308)*

6.3.2.1 File Name

[rte_sws_7504]⌈

The name of the *Module Interlink Header File* shall be formed in the following way:

```
SchM_<bsnp>[_<vi>_<ai>].h
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the BSW module and

<ai> is the vendorApiInfix of the BSW module.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. ⌋*(BSW00415, BSW00300, BSW00347)*

Example 6.2

The following declaration in the input XML:

```
<AR-PACKAGE>
  <SHORT-NAME>CanDriver</SHORT-NAME>
  <ELEMENTS>
    <BSW-MODULE-DESCRIPTION>
      <SHORT-NAME>Can</SHORT-NAME>
      <INTERNAL-BEHAVIORS>
        <BSW-INTERNAL-BEHAVIOR>
          <SHORT-NAME>YesWeCan</SHORT-NAME>
        </BSW-INTERNAL-BEHAVIOR>
      </INTERNAL-BEHAVIORS>
    </BSW-MODULE-DESCRIPTION>
    <BSW-IMPLEMENTATION>
      <SHORT-NAME>MyCanDrv</SHORT-NAME>
      <VENDOR-ID>25</VENDOR-ID>
      <BEHAVIOR-REF DEST="BSW-INTERNAL-BEHAVIOR">/CanDriver/Can/
          YesWeCan</BEHAVIOR-REF>
      <VENDOR-API-INFIX>Dev0815</VENDOR-API-INFIX>
    </BSW-IMPLEMENTATION>
  </ELEMENTS>
</AR-PACKAGE>
```

should result in the *Module Interlink Header* `SchM_Can_25_Dev0815.h` being generated.

The concatenation of the basic software module prefix (which has to be equally with the short name of the basic software module description) and the `vendorApiInfix` is required to support the separation of several basic software module instances. In difference to the multiple instantiation concept of software components, where the same component code is used for all component instances, basic software modules are multiple instantiated by creation of own code per instance in a different name space.

6.3.2.2 Scope

[rte_sws_7505]⌈ The *Module Interlink Header* for a component shall contain declarations relevant for that instance of a basic software module. ⌋*(BSW00415)*

Requirement rte_sws_7505 means that compile time checks ensure that a *Module Interlink Header File* that uses the *Module Interlink Header File* only accesses the generated data types to which it has been configured. The use of data types which are not used by the basic software module, will fail with a compiler error [RTE00017].

6.3.2.3 File Contents

[rte_sws_7506]⌈ The *Module Interlink Header File* shall include the *Module Interlink Types Header File*. ⌋*(BSW00415)*

The name of the *Module Interlink Types Header File* is defined in Section 6.3.1.

[rte_sws_7507]⌈ The *Module Interlink Header* shall be valid for both C and C++ source. ⌋*(RTE00126, RTE00138)*

Requirement rte_sws_7507 is met by ensuring that all definitions within the *Application Types Header File* are defined using C linkage if a C++ compiler is used.

[rte_sws_7508]⌈ All definitions within in the *Module Interlink Header File* shall be preceded by the following fragment:

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

[rte_sws_7509]⌈ All definitions within the *Module Interlink Header File* shall be suffixed by the following fragment:

```
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
```

⌋*(RTE00126, RTE00138)*

6.3.2.3.1 Entry Point Prototype

The *Module Interlink Header File* also includes a prototype for each `BswSchedulableEntity`s entry point (rte_sws_7283).

6.3.2.3.2 Basic Software Scheduler - Basic Software Module Interface

The *Module Interlink Header File* defines the "interface" between a *Basic Software Module* and the *Basic Software Scheduler*. The interface consists of the *Basic Software Scheduler* API for the *Basic Software Module* and the prototypes for `BswSchedulableEntity`s entry point. The definition of the *Basic Software Scheduler* API requires in case of macro implementation that both relevant data structures and API calls are defined. In case of interfaces implemented as functions, the prototypes for the *Basic Software Scheduler* API of the particular *Basic Software Module* instance is sufficient. The data structures are dependent from the implementation and configuration of the *Basic Software Scheduler* and are not standardized. If data structures are required these shall be accessible via the *Module Interlink Header File* as well.

The RTE generator is required rte_sws_7505 to limit the contents of the *Module Interlink Header* file to only that information that is relevant to that instance of a basic software module. This requirement includes the definition of the API.

[rte_sws_7510]⌈ Only *Basic Software Scheduler* API calls that are valid for the particular instance of a basic software module shall be defined within the modules *Module Interlink Header File*. ⌋*(BSW00415, RTE00017)*

Requirement rte_sws_7510 ensures that attempts to invoke invalid API calls will be rejected as a compile-time error [RTE00017].

[rte_sws_6534]⌈ The RTE Generator shall wrap each *Basic Software Scheduler* API definition of a variant existent API according table 4.22 if the variability shall be implemented.

```
#if (<condition> [||<condition>])

<Basic Software Scheduler API Definition>

#endif
```

where `condition` are the condition value macro(s) of the `VariationPoint`s relevant for the conditional existence of the RTE API (see table 4.22), `Basic Software Scheduler API Definition` is the code according an invariant *Basic Software Scheduler* API definition (see also rte_sws_7510, rte_sws_7250, rte_sws_7253, rte_sws_7255, rte_sws_7260, rte_sws_7556, rte_sws_7263, rte_sws_7266) ⌋*(RTE00229)*

The Basic Software Scheduler API for basic software modules is defined in 6.5

[rte_sws_7511]⌈ The *Basic Software Scheduler* API of the particular *Basic Software Module* instance shall be implemented as functions if the basic software module is delivered as object code. ⌋*(BSW00342)*

In case of basic software modules delivered as source code the definitions of the *Basic Software Scheduler* API contained in the *Module Interlink Header File* can be optimized during the “RTE Generation” phase when the mapping of the `BswSchedulableEntity`s to OS Tasks is known.

6.4 API Data Types

Besides the API functions for accessing *Basic Software Scheduler* services, the API also contains *Basic Software Scheduler* specific data types.

6.4.1 Predefined Error Codes for Std_ReturnType

The specification in [29] specifies a standard API return type `Std_ReturnType`. The `Std_ReturnType` defines the "‘status"’ and "‘error values"’ returned by API functions. It is defined as a `uint8` type. The value “0” is reserved for “No error occurred”.

Symbolic name	Value	Comments
[rte_sws_7289] `SCHM_E_OK`	0	No error occurred.
[rte_sws_7290] `SCHM_E_LIMIT`	130	A internal *Basic Software Scheduler* limit has been exceeded. Request could not be handled. OUT buffers are not modified. Note: The value has to be identically with rte_sws_1317
[rte_sws_7562] `SCHM_E_NO_DATA`	131	An explicit read API call returned no data. (This is no error.) Note: The value has to be identically with rte_sws_1061
[rte_sws_7563] `SCHM_E_TRANSMIT_ACK`	132	Transmission acknowledgement received. Note: The value has to be identically with rte_sws_1065

Symbolic name	Value	Comments
[rte_sws_2747] SCHM_E_IN_EXCLUSIVE_AREA	135	The error is returned by a blocking API and indicates that the schedulable entity could not enter a wait state, because one ExecutableEntity of the current task's call stack has entered or is running in an ExclusiveArea. Note: The value has to be identically with rte_sws_2739
[rte_sws_7054] SCHM_E_TIMEOUT	129	The configured timeout exceeds before the intended result was ready. Note: The value has to be identically with rte_sws_1064

Table 6.1: Basic Software Scheduler Error and Status values

The underlying type for Std_ReturnType is defined as a uint8 for reasons of compatibility. Consequently, #define is used to declare the error values:

```
typedef uint8 Std_ReturnType; /* defined in Std_Types.h */

#define SCHM_E_OK 0U
```

[rte_sws_7291]⌈ The errors as defined in table 6.1 shall be defined in the *RTE Header File*. ⌋*(RTE00051)*

An Std_ReturnType value can be directly compared (for equality) with the above pre-defined error identifiers.

6.4.2 Basic Software Modes

An Rte_ModeType is used to hold the identifiers for the ModeDeclarations of a ModeDeclarationGroup.

[rte_sws_7292]⌈ For each ModeDeclarationGroup, the *Module Interlink Types Header File* shall contain a type definition

```
#ifndef RTE_MODETYPE_<ModeDeclarationGroup>
#define RTE_MODETYPE_<ModeDeclarationGroup>
typedef <type> Rte_ModeType_<ModeDeclarationGroup>;
#endif
```

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup and <type> is the shortName of the mapped ImplementationDataType.

⌋*(RTE00213)*

Note: This requirement is deprecated to avoid incompatible or duplicate type definitions (see rte_sws_7260).

Within the Rte_ModeType_<ModeDeclarationGroup>, the (Rte_ModeType_<ModeDeclarationGroup>)<n> value, where <n> is the number of modes declared within the group, is reserved to express a transition between modes.

[rte_sws_7293]⌈ For each ModeDeclarationGroup, the *Module Interlink Types Header File* shall contain a definition

```
#ifndef RTE_TRANSITION_<ModeDeclarationGroup>
#define RTE_TRANSITION_<ModeDeclarationGroup> \
      <n>U
#endif
```

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup[1] and <n> is the number of modes declared within the group. ⌋*(RTE00213)*

[rte_sws_7294]⌈ For each mode of a mode declaration, the *Module Interlink Types Header File* shall contain a definition

```
#ifndef RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>
#define RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> \
      <index>U
#endif
```

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup, <ModeDeclaration> is the short name of a ModeDeclaration[2], and <index> is the index of the ModeDeclarations in alphabetic ordering (ASCII / ISO 8859-1 code in ascending order) of the short names within the ModeDeclarationGroup. The lowest index shall be '0' and therefore the range of assigned values is 0..<n> where <n> is the number of modes declared within the group ⌋*(RTE00213)*

6.5 API Reference

This chapter defines the "interface" between a particular instance of a *Basic Software Module* and the *Basic Software Scheduler*. The wild-card <bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594.

6.5.1 SchM_Enter

Purpose: SchM_Enter function enters an exclusive area of an *Basic Software Module*.

Signature: **[rte_sws_7250]**⌈
void SchM_Enter_<bsnp>[_<vi>_<ai>]_<name>()

[1]No additional capitalization is applied to the names.
[2]No additional capitalization is applied to the names.

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the calling BSW module,

<ai> vendorApiInfix of the calling BSW module and

<name> name is the exclusive area name.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00222, BSW00347, RTE00046)*

Existence: **[rte_sws_7251]**⌈ A SchM_Enter API shall be created for each ExclusiveArea that is declared in the BswBehavior and which has an canEnterExclusiveArea association. ⌋*(RTE00222, RTE00046)*

Description: The SchM_Enter API call is invoked by an AUTOSAR BSW module to define the start of an exclusive area.

Return Value: None.

Notes: The *Basic Software Scheduler* is not required to support nested invocations of SchM_Enter for the same exclusive area.

[rte_sws_7252]⌈ The *Basic Software Scheduler* shall permit calls to SchM_Enter and SchM_Exit to be nested as long as different exclusive areas are exited in the reverse order they were entered. ⌋*(RTE00222, RTE00046)*

[rte_sws_ext_7285] The SchM_Enter and SchM_Exit API may only be used by BswModuleEntitys that contain a corresponding canEnterExclusiveArea association

[rte_sws_ext_7529] The SchM_Enter and SchM_Exit API may only be called nested if different exclusive areas are invoked; in this case exclusive areas shall exited in the reverse order they were entered.

[rte_sws_7578]⌈ The *Basic Software Scheduler* shall support calls of SchM_Enter and SchM_Exit after initialization of the OS but before the *Basic Software Scheduler* is initialized. ⌋*(RTE00222, RTE00046)*

[rte_sws_7579]⌈ The *Basic Software Scheduler* shall support calls of SchM_Enter and SchM_Exit in the context of os tasks, category 1 and category 2 interrupts. ⌋*(RTE00222, RTE00046)*

Note: the possible implementation mechanism for such an exclusive area is limited in this case to mechanism available for the related kind of context. For instance SuspendAllInterrupts and ResumeAllInterrupts service of the OS are available for all kind of context but GetRe-

source and ReleaseResource is only available for tasks and category 2 interrupts.

Within the *AUTOSAR OS* an attempt to lock a resource cannot fail because the lock is already held. The lock attempt can only fail due to configuration errors (e.g. caller not declared as accessing the resource) or invalid handle. Therefore the return type from this function is `void`.

6.5.2 SchM_Exit

Purpose: `SchM_Exit` function leaves an exclusive area of an *Basic Software Module*.

Signature: **[rte_sws_7253]**⌈
`void`
`SchM_Exit_<bsnp>[_<vi>_<ai>]_<name>()`

Where here

`<bsnp>` is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

`<vi>` is the `vendorId` of the calling BSW module,

`<ai>` `vendorApiInfix` of the calling BSW module and

`<name>` name is the exclusive area name.

The sub part in squared brackets `[_<vi>_<ai>]` is omitted if no `vendorApiInfix` is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00222, BSW00347, RTE00046)*

Existence: **[rte_sws_7254]**⌈ A `SchM_Exit` API shall be created for each `ExclusiveArea` that is declared in the `BswBehavior` and which has an `canEnterExclusiveArea` association.. ⌋*(RTE00222, RTE00046)*

Description: The `SchM_Exit` API call is invoked by an AUTOSAR BSW module to define the end of an exclusive area.

Return Value: None.

Notes: The *Basic Software Scheduler* is not required to support nested invocations of `SchM_Exit` for the same exclusive area.

Requirement rte_sws_7252 permits calls to SchM_Exit and SchM_Exit to be nested as long as different exclusive areas are exited in the reverse order they were entered.

[rte_sws_ext_7189] The SchM_Exit API may only be used by BswModuleEntitys that contain a corresponding canEnterExclusiveArea association

6.5.3 SchM_Switch

Purpose: Initiate a mode switch. The SchM_Switch API call is used for sending of a mode switch notification by a *Basic Software Module*.

Signature: **[rte_sws_7255]**⌈

```
Std_ReturnType
SchM_Switch_<bsnp>[_<vi>_<ai>]_<name>(
                  IN <mode>)
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the calling BSW module,

<ai> vendorApiInfix of the calling BSW module and

<name> is the provided (providedModeGroup) ModeDeclarationGroupPrototype name.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00215, BSW00347)*

Existence: **[rte_sws_7256]**⌈ The existence of a managedModeGroup association to a providedModeGroup ModeDeclarationGroupPrototype shall result in the generation of a SchM_Switch API. ⌋*(RTE00215)*

[rte_sws_ext_7257] The SchM_Switch API may only be used by BswModuleEntitys that contain a corresponding managedModeGroup association

Description: The SchM_Switch triggers a mode switch for all connected required (requiredModeGroup) ModeDeclarationGroupPrototypes.

The SchM_Switch API call includes exactly one IN parameter for the next mode <mode> of type Rte_ModeType_<M> where <M> is the *ModeDeclarationGroup* short name.

Return Value: The return value is used to indicate errors detected by the *Basic Software Scheduler* during execution of the `SchM_Switch` call.

- **[rte_sws_7258]**⌈ `SCHM_E_OK` – data passed to service successfully. ⌋*(RTE00213, RTE00214, RTE00094)*
- **[rte_sws_7259]**⌈ `SCHM_E_LIMIT` – a mode switch has been discarded due to a full queue. ⌋*(RTE00213, RTE00214, RTE00143)*

Notes: `SchM_Switch` is restricted to ECU local communication.

If a mode instance is currently involved in a transition then the `SchM_Switch` API will attempt to queue the request and return rte_sws_2667. However if no transition is in progress for the mode instance, the mode disablings and the activations of *OnEntry*, *OnTransition*, and *OnExit* runnables for this mode instance are executed before the `SchM_Switch` API returns rte_sws_2665.

Note that the mode switch might be discarded when the queue is full and a mode transition is in progress, see rte_sws_2675.

[rte_sws_7286]⌈ If the mode switched acknowledgment is enabled, the RTE shall notify the mode manager when the mode switch is completed. ⌋*(RTE00213, RTE00214, RTE00122)*

6.5.4 SchM_Mode

There exist two versions of the `SchM_Mode` APIs. Depending on the attribute *enhancedModeApi* in the *basic ssoftware module description* there shall be provided different versions of this API (see also 6.5.5).

Purpose: Provides the currently active mode of a required (`requiredModeGroup`) `ModeDeclarationGroupPrototype`.

Signature: **[rte_sws_7260]**⌈

```
<return>
SchM_Mode_<bsnp>[_<vi>_<ai>]_<name>()
```

Where here

`<bsnp>` is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

`<vi>` is the `vendorId` of the calling BSW module,

`<ai>` `vendorApiInfix` of the calling BSW module and

`<name>` is the required (`requiredModeGroup`) `ModeDeclarationGroupPrototype` name.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00213, BSW00347)*

Existence: **[rte_sws_7261]**⌈ If a accessedModeGroup association to a providedModeGroup or requiredModeGroup ModeDeclarationGroupPrototype exists and if the attribute *enhancedModeApi* of the BswModeSenderPolicy resp. BswModeReceiverPolicy is set to *false* a SchM_Mode API according to rte_sws_7261. ⌋*(RTE00215)*

Note: This ensures the availability of the SchM_Mode API for the mode manager and mode user

[rte_sws_ext_7587] The SchM_Mode API may only be used by BswModuleEntitys that contain a corresponding *managedModeGroup* association or accessedModeGroup association

Description: The SchM_Mode API tells the *Basic Software Module* which mode of a required or provided ModeDeclarationGroupPrototype is currently active. This is the information that the *RTE* uses for the ModeDisablingDependencys. A new mode will not be indicated immediately after the reception of a mode switch notification from a mode manager, see section 4.4.4.During mode transitions, i.e. during the execution of runnables that are triggered on exiting one mode or on entering the next mode, overlapping mode disablings of two modes are active. In this case, the SchM_Mode API will return RTE_TRANSITION_<ModeDeclarationGroup>.

The SchM_Mode will return the same mode for all required or provided ModeDeclarationGroupPrototypes that are connected. (see rte_sws_2630).

Return Value: The type is Rte_ModeType_<M> where <M> is the *ModeDeclarationGroup* short name.

[rte_sws_7262]⌈ The SchM_Mode API shall return the following values:

- during mode transitions:
  RTE_TRANSITION_<ModeDeclarationGroup>,

  where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup.
- else:
  RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>,

  where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup and <ModeDeclaration> is the short name of the currently active ModeDeclaration

⌋*(RTE00144)*

Notes: None.

6.5.5 Enhanced SchM_Mode

Purpose: Provides the currently active mode of a required (requiredModeGroup) ModeDeclarationGroupPrototype. If the corresponding mode machine instance is in transition additionally the values of the previous and the next mode are provided.

Signature: **[rte_sws_7694]**⌈

```
<return>
SchM_Mode_<bsnp>[_<vi>_<ai>]_<name>(
                OUT <previousmode>,
                OUT <nextmode>)
)
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the calling BSW module,

<ai> vendorApiInfix of the calling BSW module and

<name> is the required (requiredModeGroup) ModeDeclarationGroupPrototype name.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(RTE00213, BSW00347)*

Existence: **[rte_sws_8507]**⌈ The existence of a accessedModeGroup association to a providedModeGroup or requiredModeGroup ModeDeclarationGroupPrototype given that the attribute *enhancedModeApi* of the BswModeSenderPolicy resp. BswModeReceiverPolicy is set to *true* shall result in the generation of a SchM_Mode API according to rte_sws_8506. ⌋*(RTE00215)*

Note: This ensures the availability of the SchM_Mode API for the mode manager and mode user

[rte_sws_ext_8508] The SchM_Mode API may only be used by BswModuleEntitys that contain a corresponding *managedModeGroup* association or accessedModeGroup association

Description: The SchM_Mode API tells the *Basic Software Module* which mode of a required or provided ModeDeclarationGroupPrototype is currently active. This is the information that the *RTE* uses for the ModeDisablingDependencys. A new mode will not be indicated immediately after the reception of a mode switch notification from a mode manager, see section 4.4.4.During mode transitions, i.e. during the execution of runnables that are triggered on exiting one mode or on entering the next mode, overlapping mode disablings of two modes are active. In this case, the SchM_Mode API will return RTE_TRANSITION_<ModeDeclarationGroup>. The parameter <previousmode> then contains the mode currently being left. The parameter <nextmode> contains the mode being entered.

The SchM_Mode will return the same mode for all required or provided ModeDeclarationGroupPrototypes that are connected. (see rte_sws_2630).

Return Value: The type is Rte_ModeType_<M> where <M> is the *ModeDeclarationGroup* short name.

[rte_sws_8509]⌈ During transitions SchM_Mode API shall return the following values:

- the return value shall be RTE_TRANSITION_<ModeDeclarationGroup>
- <previousmode> shall contain the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> of the mode being left,
- <nextmode> shall contain the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration> of the mode being entered,

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup.

⌋*(RTE00144)*

[rte_sws_8510]⌈ If the mode machine instance is in a defined mode SchM_Mode shall return the follwing values:

- the return value shall contain the value of the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>,
- <previousmode> shall contain the value of the RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>,
- <nextmode> shall contain the the value of RTE_MODE_<ModeDeclarationGroup>_<ModeDeclaration>,

where <ModeDeclarationGroup> is the short name of the ModeDeclarationGroup and <ModeDeclaration> is the short name of the currently active ModeDeclaration.

⌋*(RTE00144)*

Notes: None.

6.5.6 SchM_SwitchAck

Purpose: Provide access to acknowledgment notifications for mode communication.

Signature: **[rte_sws_7556]**⌈
```
Std_ReturnType
SchM_SwitchAck_<bsnp>[_<vi>_<ai>]_<name>()
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the calling BSW module,

<ai> vendorApiInfix of the calling BSW module and

<name> is the required (requiredModeGroup) ModeDeclarationGroupPrototype name.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(BSW00310, RTE00213)*

Existence: **[rte_sws_7557]**⌈ Acknowledgement is enabled for a provided (providedModeGroup) ModeDeclarationGroupPrototype by the presence of an ackRequest attribute of the BswModeSenderPolicy. ⌋*(RTE00213, RTE00122)*

[rte_sws_7558]⌈ A non-blocking SchM_SwitchAck API shall be generated for a provided (providedModeGroup) ModeDeclarationGroupPrototype if acknowledgement is enabled and a managedModeGroup association references the providedModeGroup ModeDeclarationGroupPrototype. ⌋*(RTE00213, RTE00122)*

[rte_sws_ext_7567] The SchM_SwitchAck API may only be used by BswModuleEntitys that contain a corresponding managedModeGroup association

Description: The SchM_SwitchAck API takes no parameters – the return value is used to indicate the acknowledgement status to the caller.

Return Value: The return value is used to indicate the "status" status and errors detected by the *Basic Software Scheduler* during execution of the `Rte_SwitchAck` call.

- **[rte_sws_7560]**⌈ `SCHM_E_NO_DATA` – (non-blocking read) no error is occurred when the `SchM_SwitchAck` read was attempted. ⌋*(RTE00213, RTE00122)*
- **[rte_sws_7561]**⌈ `SCHM_E_TRANSMIT_ACK` – For communication of mode switches, this indicates, that the `BswSchedulableEntity`s on the transition have been executed and the mode disablings have been switched to the new mode (see rte_sws_2587). ⌋*(RTE00213, RTE00122)*
- **[rte_sws_7055]**⌈ `SCHM_E_TIMEOUT` The configured timeout exceeds before the mode transition was completed.
  OR:
  The partition of the `mode users` is stopped or restarting or has been restarted while the mode switch was requested. ⌋*(RTE00213, RTE00122)*

The `SCHM_E_TRANSMIT_ACK` return value is not considered to be an error but rather indicates correct operation of the API call.

When `SCHM_E_NO_DATA` occurs, a *Basic Software Module* is free to reinvoke `SchM_SwitchAck` and thus repeat the attempt to read the mode switch acknowledgment status.

The `SCHM_E_TIMEOUT` return value can denote a stopped or restarting partition even for the `SchM_SwitchAck` API in case of a `common mode machine instance`.

Notes: If multiple transmissions on the same provided (`providedModeGroup`) `ModeDeclarationGroupPrototype` are outstanding it is not possible to determine which is acknowledged first. If this is important, transmissions should be serialized with the next occurring only when the previous transmission has been acknowledged or has timed out.

6.5.7 SchM_Trigger

Purpose: Triggers the activation of connected `BswSchedulableEntity`s of the same or other *Basic Software Modules*.

Signature: **[rte_sws_7263]**⌈
signature without queuing support:

```
void
SchM_Trigger_<bsnp>[_<vi>_<ai>]_<name>()
```

signature with queuing support:

```
Std_ReturnType
SchM_Trigger_<bsnp>[_<vi>_<ai>]_<name>()
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the calling BSW module,

<ai> vendorApiInfix of the calling BSW module and

<name> is the released (releasedTrigger) Trigger name.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528.

The signature for queuing support shall be generated by the RTE generator if the swImplPolicy of the Trigger is set to queued. ⌋*(RTE00218, BSW00347)*

Existence:

[rte_sws_7264]⌈ The existence of a issuedTrigger association to the released (releasedTrigger) Trigger shall result in the generation of a SchM_Trigger API. ⌋*(RTE00218)*

[rte_sws_ext_7265] The SchM_Trigger API may only be used by the BswModuleEntity that contains the corresponding issuedTrigger association.

Description:

The SchM_Trigger triggers an execution for all BswSchedulableEntitys whose BswExternalTriggerOccurredEvent is associated to connected required Trigger.

Return Value:

None in case of signature without queuing support.

[rte_sws_6722]⌈ The SchM_Trigger API shall return the following values:

- SCHM_E_OK if the trigger was successfully queued or if no queue is configured
- SCHM_E_LIMIT if the trigger was not queued because the maximum queue size is already reached.

in the case of signature with queuing support. ⌋*(RTE00235)*

Notes:

SchM_Trigger is restricted to ECU local communication.

6.5.8 SchM_ActMainFunction

Purpose: Triggers the activation of the `BswSchedulableEntity` which is associated with an `activationPoint` of the same or *Basic Software Module*.

Signature: **[rte_sws_7266]**⌈
signature without queuing support:

```
void
SchM_ActMainFunction_<bsnp>[_<vi>_<ai>]_<name>()
```

signature with queuing support:

```
Std_ReturnType
SchM_ActMainFunction_<bsnp>[_<vi>_<ai>]_<name>()
```

Where here

`<bsnp>` is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

`<vi>` is the `vendorId` of the calling BSW module,

`<ai>` `vendorApiInfix` of the calling BSW module and

`<name>` is the associated `BswInternalTriggeringPoint` short name.

The sub part in squared brackets `[_<vi>_<ai>]` is omitted if no `vendorApiInfix` is defined for the *Basic Software Module*. See rte_sws_7528.

The signature for queuing support shall be generated by the RTE generator if the `swImplPolicy` of the `BswInternalTriggeringPoint` is set to `queued`. ⌋*(RTE00218, BSW00347)*

Existence: **[rte_sws_7267]**⌈ The existence of an `activationPoint` shall result in the generation of a `SchM_ActMainFunction` API. ⌋*(RTE00218)*

[rte_sws_ext_7268] The `SchM_ActMainFunction` API may only be used by the `BswModuleEntity` that contains the corresponding `activationPoint` association.

Description: The `SchM_ActMainFunction` triggers an execution for all `BswSchedulableEntity`s whose `BswInternalTriggerOccurredEvent` is associated by `activationPoint`.

Return Value: None in case of signature without queuing support.

[rte_sws_6723]⌈ The `SchM_ActMainFunction` API shall return the following values:

- SCHM_E_OK if the trigger was successfully queued or if no queue is configured
- SCHM_E_LIMIT if the trigger was not queued because the maximum queue size is already reached.

in the case of signature with queuing support. ⌋*(RTE00235)*

Notes: SchM_ActMainFunction is restricted to ECU local communication.

6.5.9 SchM_CData

Purpose: Provide access to the calibration parameter of a *Basic Software Module* defined internally. The ParameterDataPrototype in the role perInstanceParameter is used to define *Basic Software Module* internal calibration parameters. Internal because the ParameterDataPrototype cannot be reused outside the *Basic Software Module*. Access is read-only. Each instance has an own data value associated with it.

Signature: **[rte_sws_7093]**⌈
```
void
SchM_CData_<bsnp>[_<vi>_<ai>]_<name>()
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the calling BSW module,

<ai> vendorApiInfix of the calling BSW module and

<name> is the shortName of the ParameterDataPrototype.

The sub part in squared brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(BSW00347, RTE00155)*

Existence: **[rte_sws_7094]**⌈ An SchM_CData API shall be created for each defined ParameterDataPrototype in the role perInstanceParameter ⌋*(RTE00155)*

Description: The SchM_CData API provides access to the defined calibration parameter within a *Basic Software Module*. The actual data values for a *Basic Software Module* instance may be set after component compilation.

Return Value: The SchM_CData return value provide access to the data value of the ParameterDataPrototype in the role perInstanceParameter.

The return type of SchM_CData is dependent on the ImplementationDataType of the ParameterDataPrototype and can either be a value or a pointer to the location where the value can be accessed. Thus the component does not need to use type casting to convert access to the ParameterDataPrototype data.

For details of the <return> value definition see section 5.2.6.6.

[rte_sws_7095]⌈ The return value of the corresponding SchM_CData API shall provide access to the calibration parameter value specific to the instance of the *Basic Software Module*. ⌋*(RTE00155)*

Notes: None.

6.6 Bsw Module Entity Reference

An *AUTOSAR Basic Software Module* defines one or more *"BSW module entities"*. A BSW Module entity is a piece of code with a single entry point and an associate set of attributes. In contrast to runnable entities which are exclusively scheduled by the RTE only a subset of the BSW module entities, the BswSchedulableEntitys are called by the *Basic Software Scheduler*. Others might implement 'C' function interfaces which are directly called by other BSW modules or interrupts which are called by OS / interrupt controller.

A *Basic Software Module Description* provides definitions for each BswModuleEntity within the BSW Module. The *Basic Software Scheduler* triggers the execution of BswSchedulableEntitys in response to different BswEvents.

For BSW modules implemented using C or C++ the entry point of a BswSchedulableEntity is implemented by a function with global scope defined within a BSW Modules source code. The following sections consider the function signature and prototype.

6.6.1 Signature

The definition of all BswSchedulableEntitys, whatever the BswEvent that triggers their execution, follows the same basic form.

Purpose: Trigger a BswSchedulableEntity if the related BswEvent defined within the BswModuleDescription is raised.

Signature: **[rte_sws_7282]**⌈
`void <bsnp>[_<vi>_<ai>]_<name>(void)`

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<vi> is the vendorId of the BSW module,

<ai> is the vendorApiInfix of the BSW module, and

<name> is the second part behind the <bsnp>_ of the BswModuleEntry shortName referred as implementedEntry.

The sub part in square brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528. ⌋*(BSW00347, RTE00211, RTE00213, RTE00216)*

[rte_sws_ext_7287] The *Basic Software Scheduler* requires that the BswModuleEntry has no service arguments and no return value.

6.6.2 Entry Point Prototype

The entry point defined in the Basic Software Modules source *must* be compatible with the called function when the BswSchedulableEntity is triggered by the *Basic Software Scheduler* and therefore the RTE generator is required to emit a prototype for the function.

[rte_sws_7283]⌈ The RTE generator shall emit a *Entry Point Prototype* for each BswSchedulableEntitys implementedEntry in the *Module Interlink Header* file See (6.3.2) according rte_sws_7282. ⌋*(RTE00211, RTE00213, RTE00216)*

[rte_sws_7195]⌈ The RTE Generator shall wrap each BswSchedulableEntity's *Entry Point Prototype* in the *Module Interlink Header* with the *Memory Mapping* and *Compiler Abstraction* macros.

```
#define <snp>[_<vi>_<ai>]_START_SEC_<sadm>
#include "MemMap.h"

FUNC(void, <snp>[_<vi>_<ai>]_<sadm>)
                        <bsnp>[_<vi>_<ai>]_<name> (void);

#define <snp>[_<vi>_<ai>]_STOP_SEC_<sadm>
#include "MemMap.h"
```

Where here

<bsnp> is the *BSW Scheduler Name Prefix* according rte_sws_7593 and rte_sws_7594,

<snp> is the *Section Name Prefix* according rte_sws_7595 and rte_sws_7596,

<vi> is the vendorId of the BSW module,

<ai> is the vendorApiInfix of the BSW module,

<name> is the second part behind the first underscore of the BswModuleEntry shortName referred as implementedEntry and

<sadm> is the shortName of the referred swAddrMethod.

The sub part in square brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *Basic Software Module*. See rte_sws_7528.

The Memory Mapping macros could wrap several *Entry Point Prototype* if these referring the same swAddrMethod. If RunnableEntity does not refer a swAddrMethod the <sadm> is set to CODE. ⌋*(RTE00148, RTE00149)*

Please note the example 6.2 of *Entry Point Prototype*.

[rte_sws_6533]⌈ The RTE Generator shall wrap each *Entry Point Prototype* in the *Module Interlink Header* file of a variant existent BswSchedulableEntity if the variability shall be implemented.

```
#if (<condition>)

<Entry Point Prototype>

#endif
```

where condition is the *Condition Value Macro* of the VariationPoint relevant for the variant existence of the BswSchedulableEntity (see table 4.24), Entry Point Prototype is the code according an invariant *Entry Point Prototype* (see also rte_sws_7282, rte_sws_7283). ⌋*(RTE00229)*

6.6.3 Reentrancy

A BswSchedulableEntity is declared within a BSW Module. The *Basic Software Module Scheduler* ensures that concurrent activation of same BswSchedulableEntity is only allowed if the implemented entry points attribute "isReentrant" is set to "true" (see Section 4.2.6).

Consistency rule:

[rte_sws_7588]⌈ The RTE Generator shall reject configurations where a BswSchedulableEntity whose referenced BswModuleEntry in the role implementedEntry has its isReentrant attribute set to false, and this BswSchedulableEntity is mapped to different tasks which can pre-empt each other. ⌋*(RTE00018)*

6.7 Basic Software Scheduler Lifecycle API Reference

6.7.1 SchM_Init

Purpose: Initialize the *Basic Software Scheduler* part of the RTE.

Signature: **[rte_sws_7270]**⌈

```
void SchM_Init([SchM_ConfigType * ConfigPtr])
```

⌋*(BSW101, RTE00116)*

Existence: **[rte_sws_7271]**⌈ The `SchM_Init` API is always created. ⌋*(BSW101)*

Description: `SchM_Init` is intended to allocate and initialize system resources used by the *Basic Software Scheduler* part of the RTE for the core on which it is called. After initialization the scheduling of `BswSchedulableEntity`s is enabled.

[rte_sws_ext_7272] `SchM_Init` shall be called only once by the *EcuStateManager* on each core after the basic software modules required by the *Basic Software Scheduler* part of the RTE are initialized. These modules include:

- OS

[rte_sws_6544]⌈ The optional parameter `configPtr` shall be a pointer to a post build data set which is used to resolve the `PostBuild Variability` of the *Basic Software Scheduler* and RTE. ⌋*(BSW00405, RTE00229, RTE00204, RTE00206, RTE00207)*

[rte_sws_6545]⌈ The parameter `configPtr` shall only be provided if the input configuration of the RTE and *Basic Software Scheduler* contains `PostBuild Variability` which has to be implemented by the RTE Generator. ⌋*(BSW00405, RTE00229, RTE00204, RTE00206, RTE00207)*

[rte_sws_ext_7576] The `SchM_Deinit` API may only be used after the RTE finalized (after termination of the `Rte_Stop`)

[rte_sws_7273]⌈ `SchM_Init` shall return within finite execution time – it must not enter an infinite loop. ⌋*(BSW101)*

`SchM_Init` may be implemented as a function or a macro.

Return Value: None

Notes: `SchM_Init` is declared in the lifecycle header file `Rte_Main.h`.

6.7.2 SchM_Deinit

Purpose: Finalize the *Basic Software Scheduler* part of the RTE on the core it is called.

Signature: **[rte_sws_7274]**⌈

```
void SchM_Deinit(void)
```

⌋*(BSW00336)*

Existence: **[rte_sws_7275]**⌈ The `SchM_Deinit` API is always created. ⌋*(BSW00336)*

Description: `SchM_Deinit` is used to finalize *Basic Software Scheduler* part of the RTE of the core on which it is called. This service releases all system resources allocated by the *Basic Software Scheduler* part on that core.

[rte_sws_ext_7276] `SchM_Deinit` shall be called by the *EcuStateManager* before the basic software modules required by *Basic Software Scheduler* part are shut down. These modules include:

- OS

[rte_sws_7277]⌈ `SchM_Deinit` shall return within finite execution time. ⌋*(BSW00336)*

`SchM_Deinit` may be implemented as a function or a macro.

Return Value: None

Notes: `SchM_Deinit` is declared in the lifecycle header file `Rte_Main.h`.

6.7.3 SchM_GetVersionInfo

Purpose: Returns the version information of the *Basic Software Scheduler*.

Signature: **[rte_sws_7278]**⌈
```
void SchM_GetVersionInfo(Std_VersionInfoType * versioninfo)
```
⌋*(BSW00407)*

Existence: **[rte_sws_7279]**⌈ The `SchM_GetVersionInfo` API is only created if `RteSchMVersionInfoApi` is set to `true`. ⌋*(BSW00407)*

Description: **[rte_sws_7280]**⌈ `SchM_GetVersionInfo` shall return the version information of the RTE module which includes the *Basic Software Scheduler*. The version information includes:

- Module Id
- Vendor Id
- Vendor specific version numbers

⌋*(BSW00407)*

[rte_sws_7281]⌈ The parameter `versioninfo` of the `SchM_GetVersionInfo` shall point to the memory location holding the version information of the *Basic Software Scheduler*. ⌋*(BSW00407)*

`SchM_GetVersionInfo` may be implemented as a function or a macro.

Return Value: None

Notes: `SchM_GetVersionInfo` is declared in the lifecycle header file `Rte_Main.h`.

The existence of the API `SchM_GetVersionInfo` depends on the parameter `RteSchMVersionInfoApi`.

Vendor specific version numbers shall represent build version which depends from the RTE generator version and the input configuration. It is not in the scope if this specification to standardize the way how the version numbers are created in detail because these are the vendor specific version numbers.

7 RTE ECU Configuration

The RTE provides the glue layer between the AUTOSAR software-components and the Basic Software thus enabling several AUTOSAR software-components to be integrated on one ECU. The RTE layer is shown in figure 7.1.

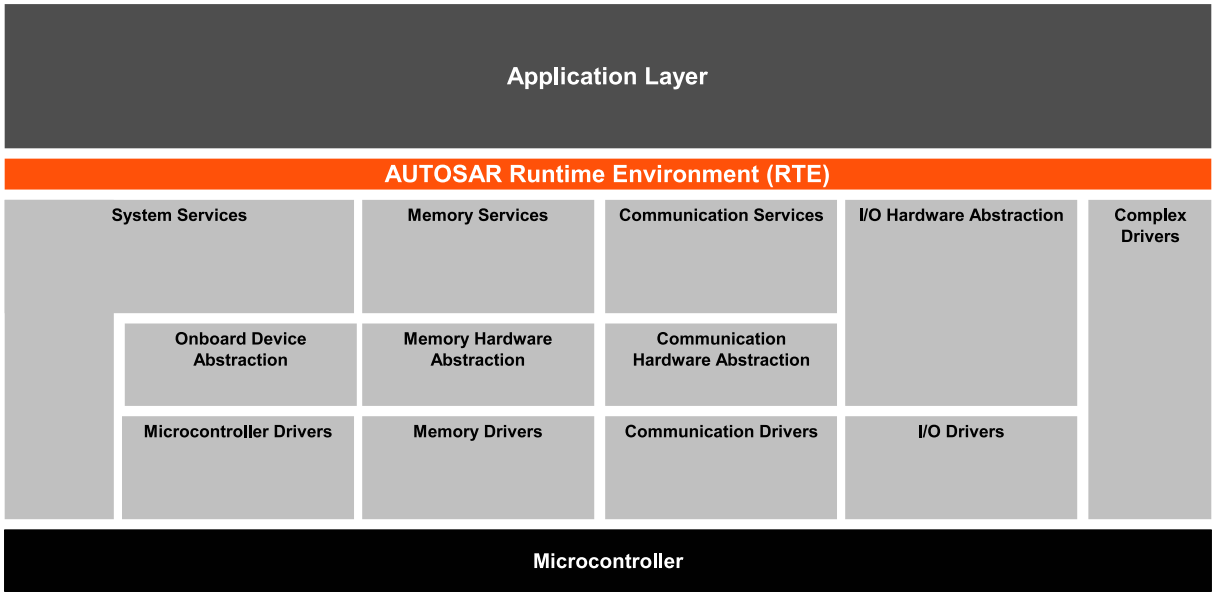


Figure 7.1: ECU Architecture RTE

The overall structure of the RTE configuration parameters is shown in figure 7.2. It has to be distinguished between the configuration parameters for the RTE generator and the configuration parameters for the generated RTE itself.

Most of the information needed to generate an RTE is already available in the ECU Extract of the System Description [8]. From this extract also the links to the AUTOSAR software-component descriptions and ECU Resource description are available. So only additional information not covered by the three aforementioned formats needs to be provided by the ECU Configuration description.

To additionally allow the most flexibility and freedom in the implementations of the RTE, only configuration parameters which are common to all implementations are standardized in the ECU Configuration Parameter definition. Any additional configuration parameters which might be needed to configure a full functional RTE have to be specified using the vendor specific parameter definition mechanism described in the ECU Configuration specification document [15].

7.1 Ecu Configuration Variants

The RTE shall supports two Ecu Configuration Variants:

[rte_sws_5103]⌈ VARIANT-PRE-COMPILE Only parameters with "Pre-compile time" configuration are allowed in this variant. ⌋*(BSW00345, BSW00397)*

[rte_sws_5104]⌈ VARIANT-POST-BUILD Parameters with "Pre-compile time", "Link time" and "Post-build time" are allowed in this variant. ⌋*(BSW00399, BSW00400, RTE00201, RTE00204, RTE00206, RTE00207, RTE00229)*

For details on the ECU Configuration approach please refer to the *Specification of ECU Configuration* [15].

7.2 RTE Module Configuration

Figure 7.2 shows the module configuration of the Rte and its sub-containers.

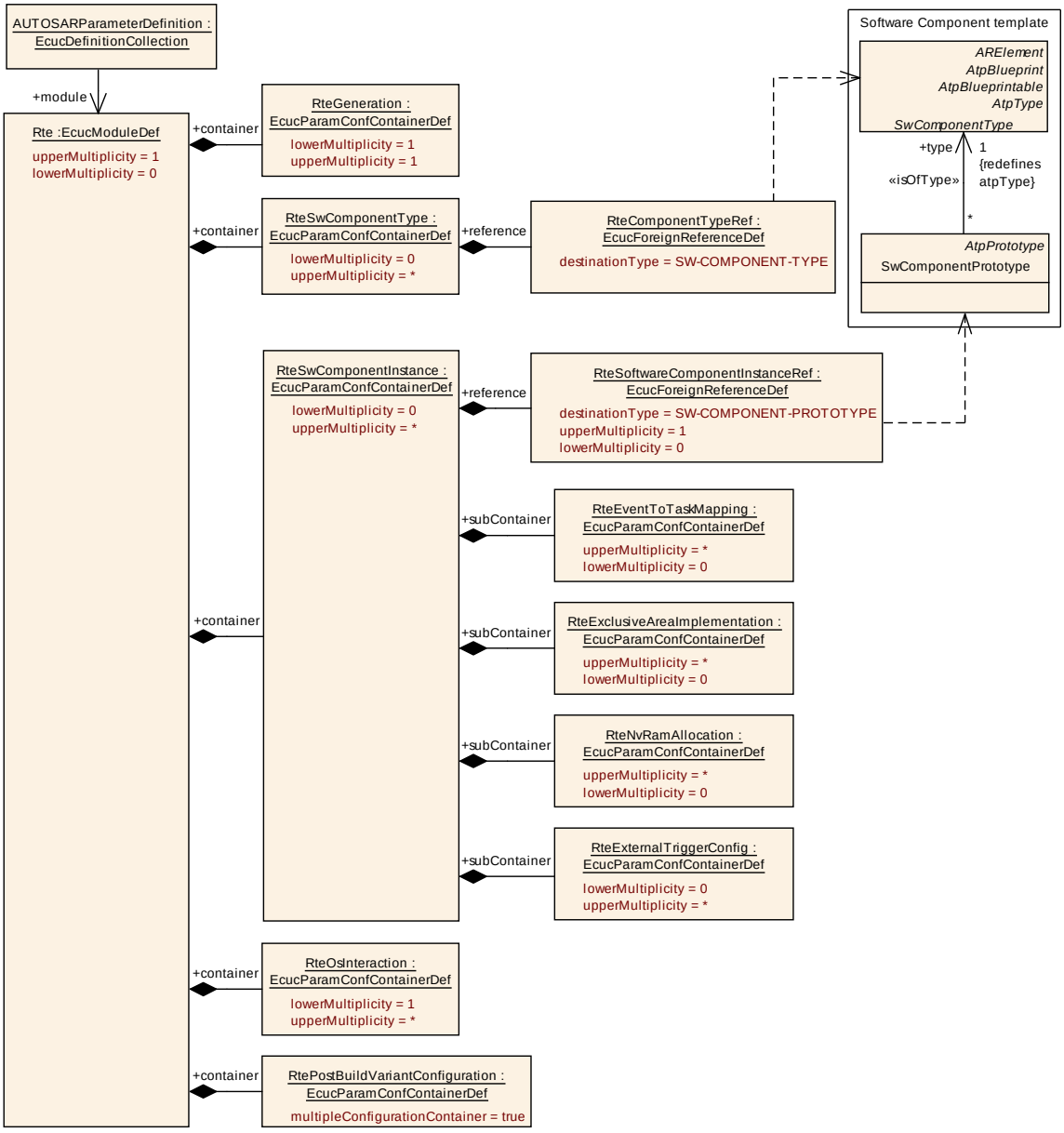


Figure 7.2: RTE configuration overview

Module Name	Rte	
Module Description	Configuration of the Rte (Runtime Environment) module.	
Included Containers		
Container Name	**Multiplicity**	**Scope / Dependency**
RteBswGeneral	1	General configuration parameters of the Bsw Scheduler section.
RteBswModuleInstance	0..*	Represents one instance of a Bsw-Module configured on one ECU.
RteGeneration	1	This container holds the parameters for the configuration of the RTE Generation.
RteImplicitCommunication	0..*	Configuration of the Implicit Communication behavior to be generated.

Container Name	Multiplicity	Scope / Dependency
RteInitializationBehavior	1..*	Specifies the initialization strategy for variables allocated by RTE with the purpose to implement VariableDataPrototypes. The container defines a set of RteSectionInitializationPolicys and one RteInitializationStrategy which is applicable for this set.
RteOsInteraction	1..*	Interaction of the Rte with the Os.
RtePostBuildVariant Configuration	1	Specifies the PostbuildVariantSets for each of the PostBuild configurations of the RTE. The shortName of this container defines the name of the RtePostBuildVariant.
RteSwComponentInstance	0..*	Representation of one SwComponentPrototype located on the to be configured ECU. All subcontainer configuration aspects are in relation to this SwComponentPrototype. The RteSwComponentInstance can be associated with either a AtomicSwComponentType or ParameterSwComponentType.
RteSwComponentType	0..*	Representation of one SwComponentType for the base of all configuration parameter which are affecting the whole type and not a specific instance.

7.2.1 RTE Configuration Version Information

In order to identify the RTE Configuration version a dedicated RTE code has been generated from the RTE Configuration information may contain one or more DOC-REVISION elements in the ECUC-MODULE-CONFIGURATION-VALUES element of the RTE Configuration (see example 7.1).

[rte_sws_5184]⌈ The REVISION-LABEL shall be parsed according to the rules defined in the Generic Structure Template [10] for RevisionLabelString allowing to parse the three version informations for AUTOSAR:

- major version: first part of the REVISION-LABEL
- minor version: second part of the REVISION-LABEL
- patch version: third part of the REVISION-LABEL
- optional fourth part shall be used for documentation purposes in the Basic Software Module Description (see section 3.4.3)

If the parsing fails all three version numbers shall be set to zero. ⌋*(RTE00233)*

[rte_sws_5185]⌈ If there are several DOC-REVISION elements in the input ECUC-MODULE-CONFIGURATION-VALUES the newest according to the DATE shall be taken into account.

If the search for the newest DOC-REVISION fails three version numbers shall be set to zero. ⌋*(RTE00233)*

Example 7.1

```
<AUTOSAR xmlns="http://autosar.org/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://autosar.org/4.0.0␣
    AUTOSAR.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>Rte_Example</SHORT-NAME>
      <ELEMENTS>
        <ECUC-MODULE-CONFIGURATION-VALUES>
          <SHORT-NAME>Rte_Configuration</SHORT-NAME>
          <ADMIN-DATA>
            <DOC-REVISIONS>
              <DOC-REVISION>
                <REVISION-LABEL>2.1.34</REVISION-LABEL>
                <DATE>2009-05-09T00:00:00.0Z</DATE>
              </DOC-REVISION>
              <DOC-REVISION>
                <REVISION-LABEL>2.1.35</REVISION-LABEL>
                <DATE>2009-06-21T09:30:00.0Z</DATE>
              </DOC-REVISION>
            </DOC-REVISIONS>
          </ADMIN-DATA>
          <DEFINITION-REF DEST="ECUC-MODULE-DEF">/AUTOSAR/Rte</DEFINITION-
              REF>
          <CONTAINERS>
            <!-- ... -->
          </CONTAINERS>
        </ECUC-MODULE-CONFIGURATION-VALUES>
      </ELEMENTS>
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

7.3 RTE Generation Parameters

The parameters in the container RteGeneration are used to configure the RTE generator. They all need to be defined during pre-compile time.

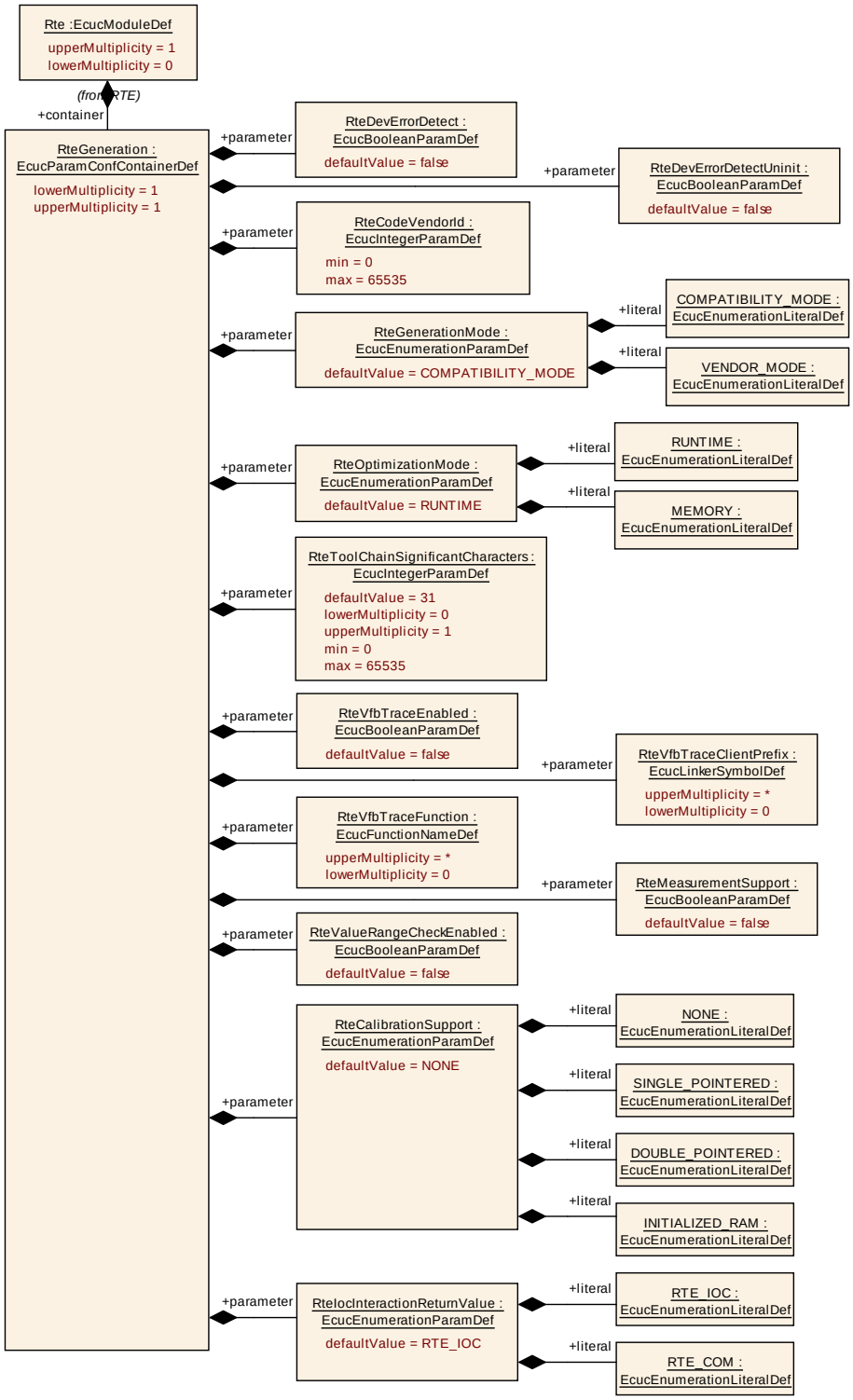


Figure 7.3: RTE generation parameters

RteGeneration

SWS Item	[rte_sws_9009_Conf]		
Container Name	RteGeneration		
Description	This container holds the parameters for the configuration of the RTE Generation.		
Configuration Parameters			

Name	RteCalibrationSupport {RTE_CALIBRATION_SUPPORT} [rte_sws_9007_Conf]		
Description	The RTE generator shall have the option to switch off support for calibration for generated RTE code. This option shall influence complete RTE code at once.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	DOUBLE_POINTERED		
	INITIALIZED_RAM		
	NONE	(default)	
	SINGLE_POINTERED		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteCodeVendorId {RTE_CODE_VENDOR_ID} [rte_sws_9086_Conf]		
Description	Holds the vendor ID of the generated Rte code.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteDevErrorDetect {RTE_DEV_ERROR_DETECT} [rte_sws_9008_Conf]		
Description	The Rte shall log development errors to the Det module.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteDevErrorDetectUninit {RTE_DEV_ERROR_DETECT_UNINIT} [rte_sws_9085_Conf]		
Description	The Rte shall detect if it is started when its APIs are called, and the BSW Scheduler shall check if it is initialized when its APIs are called.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	dependency: Shall only be used when RteDevErrorDetect equals true.		

Name	RteGenerationMode {RTE_GENERATION_MODE} [rte_sws_9010_Conf]		
Description	Switch between the two available generation modes of the RTE generator.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	COMPATIBILITY_MODE	(default)	
	VENDOR_MODE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteIocInteractionReturnValue {RTE_IOC_INTERACTION_RETURN_VALUE} [rte_sws_9094_Conf]		
Description	Defines whether the return value of RTE APIs is based on RTE-IOC interaction or RTE-COM interaction.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	RTE_COM		
	RTE_IOC	(default)	
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteMeasurementSupport {RTE_MEASUREMENT_SUPPORT} [rte_sws_9011_Conf]		
Description	The RTE generator shall have the option to switch off support for measurement for generated RTE code. This option shall influence complete RTE code at once.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteOptimizationMode {RTE_OPTIMIZATION_MODE} [rte_sws_9012_Conf]		
Description	Switch between the two available optimization modes of the RTE generator.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	MEMORY		
	RUNTIME	(default)	
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteToolChainSignificantCharacters {RTE_TOOL_CHAIN_SIGNIFICANT_CHARACTERS} [rte_sws_9013_Conf]		
Description	If present, the RTE generator shall provide the list of C RTE identifiers whose name is not unique when only the first RteToolChainSignificantCharacters characters are considered.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default Value	31		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteValueRangeCheckEnabled {RTE_VALUE_RANGE_CHECK_ENABLED} [rte_sws_9014_Conf]		
Description	If set to true the RTE generator shall enable the value range checking for the specified VariableDataPrototypes.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteVfbTraceClientPrefix [rte_sws_9016_Conf]		
Description	Defines an additional prefix for all VFB trace functions to be generated. With this approach it is possible to have debugging and DLT trace functions at the same time.		
Multiplicity	0..*		
Type	EcucLinkerSymbolDef		
Default Value			
Regular Expression			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteVfbTraceEnabled {RTE_VFB_TRACE_ENABLED} [rte_sws_9015_Conf]		
Description	The RTE generator shall globally enable VFB tracing when RteVfbTrace is set to "true".		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteVfbTraceFunction [rte_sws_9017_Conf]		
Description	The RTE generator shall enable VFB tracing for a given hook function when there is a #define in the RTE configuration header file for the hook function name and tracing is globally enabled. Example: #define Rte_WriteHook_i1_p1_a_Start This also applies to VFB trace functions with a RteVfbTraceClientPrefix, e.g. Rte_Dbg_WriteHook_I1_P1_a_Start.		
Multiplicity	0..*		
Type	EcucFunctionNameDef		
Default Value			
Regular Expression			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.4 RTE PreBuild configuration

In order to support PreBuild configuration variation of the Rte input (see also section 4.7) the container `EcucVariationResolver` is providing a set of references to `PredefinedVariant`. These define values for `SwSystemconst`.

Note that the information for the `EcucVariationResolver` is provided in the `EcuC` part of the ECU Configuration, since it does not only influence the Rte but also many other BSW Modules.

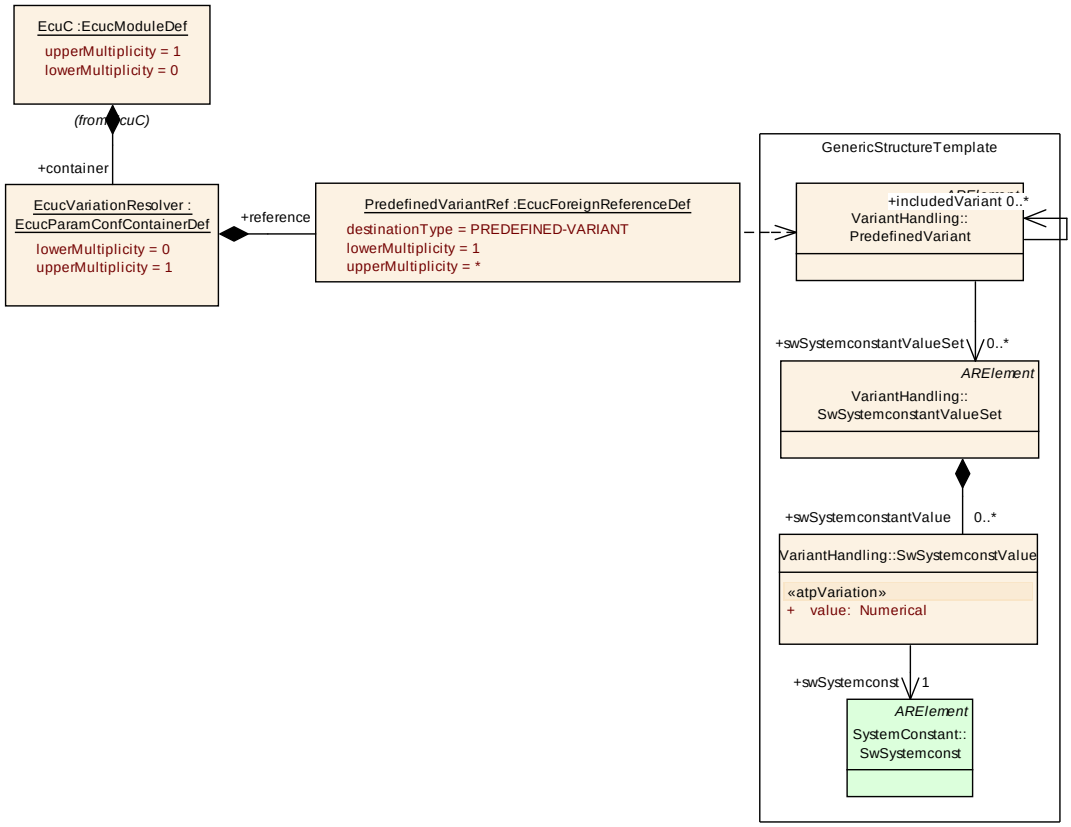


Figure 7.4: RTE PreBuild configuration

EcucVariationResolver

SWS Item	[EcuC009_Conf]
Container Name	EcucVariationResolver
Description	Collection of PredefinedVariant elements containing definition of values for SwSystemconst which shall be applied when resolving the variability during ECU Configuration.
Configuration Parameters	

Name	PredefinedVariantRef [EcuC010_Conf]		
Description			
Multiplicity	1..*		
Type	Foreign reference to PREDEFINED-VARIANT		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.5 RTE PostBuild configuration

In order to support PostBuild configuration variation of the generated Rte (see also section 4.7) the container `RtePostBuildVariantConfiguration` is used. Each instance of this container specifies *one* PostBuild variant of the generated Rte. The `shortName` of the container `RtePostBuildVariantConfiguration` specifies the variant name.

The actual values for the `PostBuildVariantCriterion` are defined in a two step approach:

1. The reference `RtePostBuildUsedPredefinedVariant` collects the `PredefinedVariant` elements.
2. Each `PredefinedVariant` element collects a set of `PostBuildVariantSet`.
3. Each `PostBuildVariantSet` defines the `PostBuildVariantCriterionValues` for a set of `PostBuildVariantCriterion`.

The basic idea is that

- the `PostBuildVariantSet` can be provided by sub-system engineer,
- the `PredefinedVariant` can be designed by the Ecu integrator.

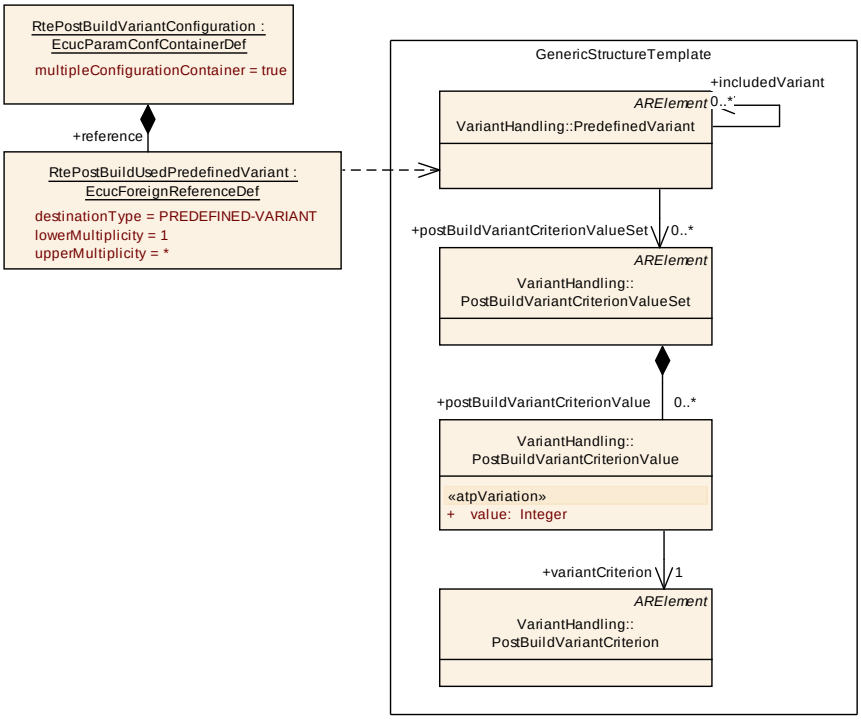


Figure 7.5: RTE PostBuild configuration

RtePostBuildVariantConfiguration

SWS Item	[rte_sws_9084_Conf]
Container Name	RtePostBuildVariantConfiguration[Multi Config Container]
Description	Specifies the PostbuildVariantSets for each of the PostBuild configurations of the RTE. The shortName of this container defines the name of the RtePostBuildVariant.
Configuration Parameters	

Name	RtePostBuildUsedPredefinedVariant [rte_sws_9083_Conf]		
Description	Reference to the PredefinedVariant element which defines the values for PostBuildVariationCriterion elements.		
Multiplicity	1..*		
Type	Foreign reference to PREDEFINED-VARIANT		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.6 Handling of Software Component instances

When entities of Software-Components are to be configured there is the need to actually address the instances of the `AtomicSwComponentType`. Since the Ecu Extract of System Description contains a flat view on the Ecu's Software-Components [8] the `SwComponentPrototype`s in the Ecu Extract already represent the instances of the Software Components.

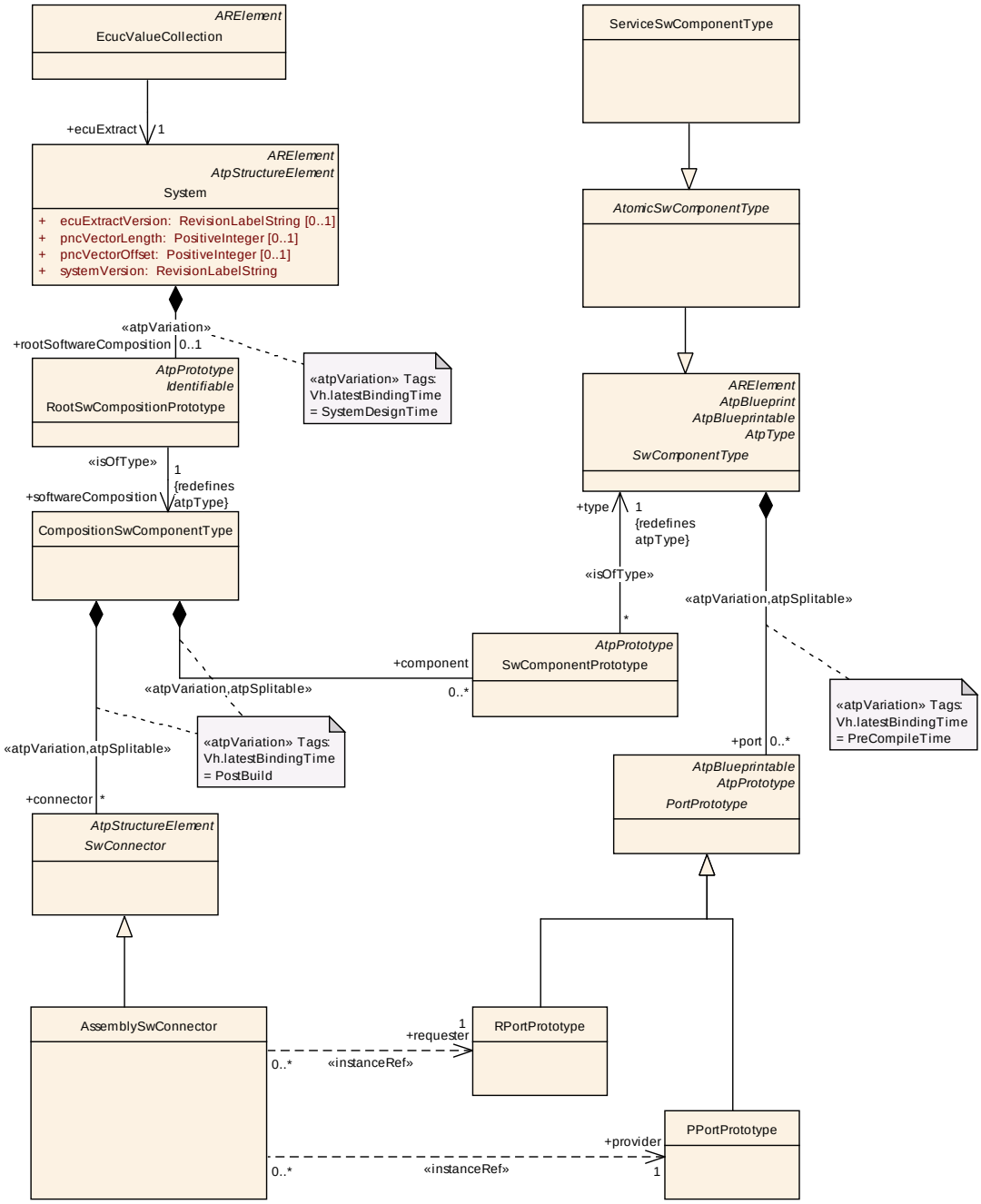


Figure 7.6: Services in the ECU Configuration

RteSwComponentInstance

SWS Item	[rte_sws_9005_Conf]
Container Name	RteSwComponentInstance
Description	Representation of one SwComponentPrototype located on the to be configured ECU. All subcontainer configuration aspects are in relation to this SwComponentPrototype. The RteSwComponentInstance can be associated with either a AtomicSwComponentType or ParameterSwComponentType.
Configuration Parameters	

Name	RteSoftwareComponentInstanceRef [rte_sws_9004_Conf]		
Description	Reference to a SwComponentPrototype.		
Multiplicity	0..1		
Type	Foreign reference to SW-COMPONENT-PROTOTYPE		
Configuration Class	Pre-compile time	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Included Containers		
Container Name	Multiplicity	Scope / Depedency
RteEventToTaskMapping	0..*	Maps a RunnableEntity onto one OsTask based on the activating RTEEvent. Even if a RunnableEntity shall be executed via a direct function call this RteEventToTaskMapping shall be specified, but no RteMappedToTask and RtePositionInTask elements given.
RteExclusiveArea Implementation	0..*	Specifies the implementation to be used for the data consistency of this ExclusiveArea.
RteExternalTriggerConfig	0..*	Defines the configuration of External Trigger Event Communication for Software Components
RteInternalTriggerConfig	0..*	Defines the configuration of Inter Runnable Triggering for Software Components
RteNvRamAllocation	0..*	Specifies the relationship between the AtomicSwComponentType's NVRAMMapping / NVRAM needs and the NvM module configuration.

The container `SwComponentInstance` collects all the configuration information related to one specific instance of a `AtomicSwComponentType`. The individual aspects will be described in the next sections.

7.6.1 RTE Event to task mapping

One of the major fragments of the RTE configuration is the mapping of AUTOSAR Software-Components' `RunnableEntity`s to OS Tasks. The parameters defined to achieve this are shown in figure 7.7.

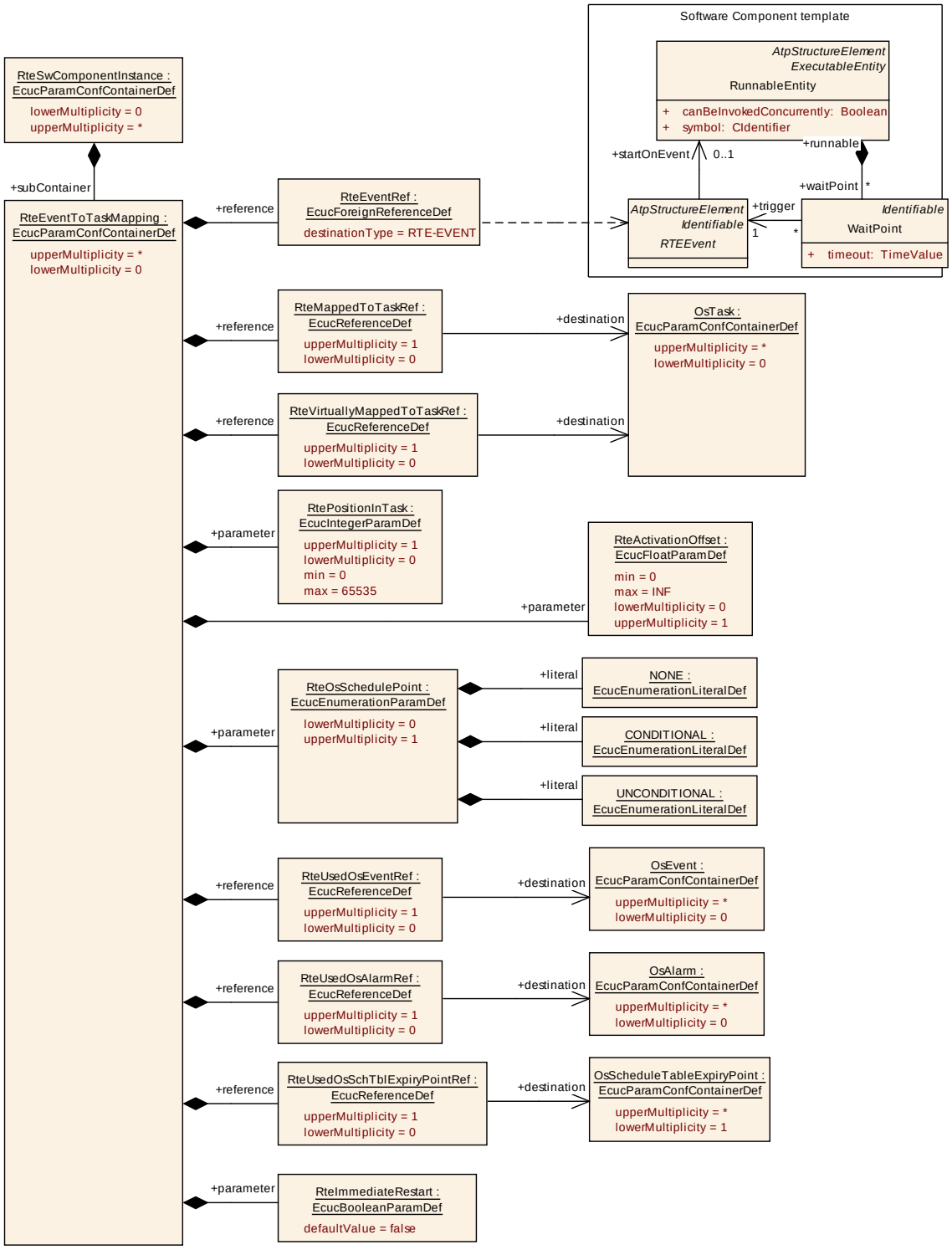


Figure 7.7: RTE Event to task mapping

The mapping is based on the `RTEEvent` because it is the source of the activation. For each `RunnableEntity` which belongs to an AUTOSAR Software-Component instance mapped on the ECU there needs to be a mapping container specifying how this `RunnableEntity` activation shall be handled.

One major constraint is posed by the canBeInvokedConcurrently attribute of each RunnableEntity because data consistency issues have to be considered.

7.6.1.1 Evaluation and execution order

Another important parameter is the PositionInTask which provides an order of RunnableEntitys within the associated OsTask. When the task is executed periodically the PositionInTask parameter defines the order of execution within the test. When the task is used to define a context for event activated RunnableEntitys the PositionInTask parameter defines the order of evaluation which actual RunnableEntity shall be executed. Thus providing means to define a deterministic delay between the beginning of execution of the task and the actual execution of the RunnableEntity's code.

In case of triggered runnables, OnEntry ExecutableEntitys, OnTransition ExecutableEntitys, OnExit ExecutableEntitys, and mode switch acknowledge ExecutableEntitys the PositionInTask parameter defines the order of evaluation which actual RunnableEntity shall be executed. All other parameters or references are not required.

7.6.1.2 Direct function call

If the RunnableEntity is a server runnable, triggered runnable, OnEntry ExecutableEntity, OnTransition ExecutableEntity, OnExit ExecutableEntity, or a mode switch acknowledge ExecutableEntity and shall be executed in the context of the caller (i.e. using a direct function call) the element RunnableEntityMapping still shall be provided to indicate that this RTEEvent has been considered in the mapping. In case of server runnables no further parameters or references are required (e.g. MappedToTaskRef can be left out).

7.6.1.3 Schedule Points

In order to allow explicit calls to the Os scheduler in an non-preemptive scheduling setup, the configuration element RteOsSchedulePoint shall be used.

[rte_sws_5113]⌈ The RTE Generator shall create an unconditional call to the Os API *Schedule* after the execution call of the RunnableEntity if the RteOsSchedulePoint configuration parameter is set to UNCONDITIONAL. In the generated code the call to the Os API *Schedule* shall always be performed, even when the RunnableEntity itself has not been executed (called). ⌋*()*

Since the execution of a RunnableEntity may be performed (e.g. due to mode dependent scheduling) the call of the Os API *Schedule* without any RunnableEntity

execution in between might occur. in order to prohibit such a call chain the CONDITIONAL schedule point is available.

[rte_sws_5114]⌈ The RTE Generator shall create a conditional call to the Os API *Schedule* after the execution call of the RunnableEntity if the RteOsSchedulePoint configuration parameter is set to CONDITIONAL. In the generated code the call to the Os API *Schedule* shall be omitted when there was already a call to the Os API *Schedule* before without any RunnableEntity execution in between. ⌋*()*

[rte_sws_7042]⌈ The Os API *Schedule* according rte_sws_5113 and rte_sws_5114 shall be called after the data written with implicit write access by the RunnableEntity are propagated to other RunnableEntitys as specified in rte_sws_7021, rte_sws_3957, rte_sws_7041 and rte_sws_3584 ⌋*()*

[rte_sws_7043]⌈ The Os API *Schedule* according rte_sws_5113 and rte_sws_5114 shall be called before the Preemption Area specific buffer used for a implicit read access of the successor RunnableEntity are filled with actual data by a copy action according rte_sws_7020. ⌋*()*

[rte_sws_5115]⌈ The RTE Generator shall create no call to the Os API *Schedule* after the execution of the RunnableEntity if the RteOsSchedulePoint configuration parameter is not present or is set to NONE. ⌋*()*

[rte_sws_5116]⌈ The RTE Generator shall reject configurations where not all RteEventToTaskMappings which map the same RunnableEntity, have different RteOsSchedulePoint settings. ⌋*(RTE00018)*

7.6.1.4 Timeprotection support

[rte_sws_7801]⌈ If RteMappedToTaskRef is configured but RteVirtuallyMappedToTaskRef is not configured, the RTE shall implement/evaluate the RTEEvent that activates the RunnableEntity and execute the RunnableEntity in the OsTask referenced by RteMappedToTaskRef. ⌋*()*

[rte_sws_7802]⌈ If both RteMappedToTaskRef and RteVirtuallyMappedToTaskRef are configured, the RTE shall implement/evaluate the RTEEvent that activates the RunnableEntity in the OsTask referenced by RteVirtuallyMappedToTaskRef but execute the RunnableEntity in the OsTask referenced by RteMappedToTaskRef. The RTE shall implement this by an activation of the OsTask referenced by RteMappedToTaskRef when the RTEEvent is evaluated as "TRUE" in the OsTask referenced by RteVirtuallyMappedToTaskRef. ⌋*(RTE00193)*

[rte_sws_7803]⌈ The RTE shall reject the configuration if RteMappedToTaskRef is not configured but RteVirtuallyMappedToTaskRef is configured. ⌋*(RTE00018)*

7.6.1.5 Os Interaction

When an OsEvent is used to activate the OsTask the reference UsedOsEventRef specifies which OsEvent is used.

When an OsAlarm is used to implement a TimingEvent or a BackgroundEvent the reference RteUsedOsAlarmRef specifies which OsAlarm is used.

[rte_sws_7806]⌈ If RteUsedOsAlarmRef is configured and RteEventRef references a TimingEvent the RTE shall implement the TimingEvent with the OsAlarm referenced by RteUsedOsAlarmRef. ⌋*(RTE00232)*

[rte_sws_7179]⌈ If RteUsedOsAlarmRef is configured and RteEventRef references a BackgroundEvent the RTE shall implement the BackgroundEvent with the OsAlarm referenced by RteUsedOsAlarmRef. ⌋*()*

When an OsScheduleTableExpiryPoint is used to implement a TimingEvent or a BackgroundEvent the reference RteUsedOsSchTblExpiryPointRef specifies which OsScheduleTableExpiryPoint is used.

[rte_sws_7807]⌈ If RteUsedOsSchTblExpiryPointRef is configured and RteEventRef references a TimingEvent the RTE shall implement the TimingEvent with the OsScheduleTableExpiryPoint referenced by RteUsedOsSchTblExpiryPointRef. ⌋*(RTE00232)*

[rte_sws_7180]⌈ If RteUsedOsSchTblExpiryPointRef is configured and RteEventRef references a BackgroundEvent the RTE shall implement the BackgroundEvent with the OsScheduleTableExpiryPoint referenced by RteUsedOsSchTblExpiryPointRef. ⌋*()*

If neither RteUsedOsSchTblExpiryPointRef nor RteUsedOsAlarmRef are configured and RteEventRef references a TimingEvent the RTE is free to implement the TimingEvent with the OsAlarm or OsScheduleTableExpiryPoint of its choice.

[rte_sws_7808]⌈ The RTE shall reject the configuration if both RteUsedOsAlarmRef and RteUsedOsSchTblExpiryPointRef are configured. ⌋*(RTE00018)*

[rte_sws_7809]⌈ The RTE shall reject the configuration if RteUsedOsAlarmRef or RteUsedOsSchTblExpiryPointRef is configured and RteEventRef doesn't reference a TimingEvent or a BackgroundEvent. ⌋*(RTE00018)*

7.6.1.6 Background activation

If neither RteUsedOsSchTblExpiryPointRef nor RteUsedOsAlarmRef is configured and RteEventRef references a BackgroundEvent the RteMappedToTaskRef has to reference the OsTask used for *Background* activation of *RunnableEntities* and *Basic Software Schedulable Entities* on the related CPU core where the partition of the software component is mapped.

The OsTask used for BackgroundEvent triggering has to have the lowest priority on the core. There can only be one 'Background' OsTask per CPU core.

[rte_sws_7181]⌈ The RTE shall reject the configuration if

- RteEventRef references a BackgroundEvent and
- neither RteUsedOsAlarmRef nor RteUsedOsSchTblExpiryPointRef are configured and
- if RteMappedToTaskRef reference an OsTask which has not the lowest priority of the core.

⌋*(RTE00018)*

7.6.1.7 Constraints

There are some constraints which do apply when actually mapping the RunnableEntity to an OsTask:

[rte_sws_5082]⌈ The following restrictions apply to RTEEvents which are used to activate RunnableEntity. OsEvents that are used to wakeUpFromWaitPoint shall not be included in the mapping. ⌋*()*

When a wakeUpFromWaitPoint is occurring the RunnableEntity resumes its execution in the context of the originally activated OsTask.

[rte_sws_5083]⌈ The RTE Generator shall reject configurations where a RunnableEntity has its canBeInvokedConcurrently attribute set to *false*, and this RunnableEntity is mapped to different tasks which can preempt each other. ⌋*()*

[rte_sws_7229]⌈ To evaluate rte_sws_5083 in case of triggered runnables which are activated by a direct function call (rte_sws_7214, rte_sws_7224 and rte_sws_7554) the OsTask (context of the caller) is defined by the RunnableEntity's containing the activating InternalTriggeringPoint or ExternalTriggeringPoint. ⌋*(RTE00162, RTE00163, RTE00230)*

[rte_sws_7155]⌈ To evaluate rte_sws_5083 in case of OnEntry ExecutableEntitys, OnTransition ExecutableEntitys, OnExit ExecutableEntitys, and mode switch acknowledge ExecutableEntitys which are activated by a direct function call the OsTask (context of the caller) is defined by the RunnableEntity's containing the activating ModeSwitchPoint. ⌋*(RTE00143, RTE00144)*

RteEventToTaskMapping

SWS Item	[rte_sws_9020_Conf]
Container Name	RteEventToTaskMapping
Description	Maps a RunnableEntity onto one OsTask based on the activating RTEEvent. Even if a RunnableEntity shall be executed via a direct function call this RteEventToTaskMapping shall be specified, but no RteMappedToTask and RtePositionInTask elements given.
Configuration Parameters	

Name	RteActivationOffset [rte_sws_9018_Conf]		
Description	Activation offset in seconds.		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	0 .. INF		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteEventRef [rte_sws_9019_Conf]		
Description	Reference to the description of the RTEEvent which is pointing to the RunnableEntity being mapped. This allows a fine grained mapping of RunnableEntites based on the activating RTEEvent.		
Multiplicity	1		
Type	Foreign reference to RTE-EVENT		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteImmediateRestart [rte_sws_9092_Conf]		
Description	When RteImmediateRestart is set to true the RunnableEntitiy shall be immediately re-started after termination if it was activated by this RTEEvent while it was already started. This parameter shall not be set to true when the mapped RTEEvent refers to a RunnableEntity which minimumStartInterval attribute is > 0.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteMappedToTaskRef [rte_sws_9021_Conf]		
Description	Reference to the OsTask the RunnableEntity activated by the RteEventRef is mapped to. If no reference to the OsTask is specified the RunnableEntity shall be executed via a direct function call.		
Multiplicity	0..1		
Type	Reference to OsTask		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteOsSchedulePoint [rte_sws_9022_Conf]		
Description	Introduce a schedule point by explicitly calling Os Schedule service after the execution of the ExecutableEntity. The Rte generator is allowed to optimize several consecutive calls to Os schedule into one single call if the ExecutableEntity executions in between have been skipped. The absence of this parameter is interpreted as "NONE". It shall be considered an invalid configuration if the task is preemptable and the value of this parameter is not set to "NONE" or the parameter is absent.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CONDITIONAL	A Schedule Point shall be introduced at the end of the execution of this ExecutableEntity. The Schedule Point can be skipped if several Schedule Points would be called without any ExecutableEntity execution in between.	
	NONE	No Schedule Point shall be introduced at the end of the execution of this ExecutableEntity.	
	UNCONDITIONAL	A Schedule Point shall always be introduced at the end of the execution of this ExecutableEntity.	
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RtePositionInTask [rte_sws_9023_Conf]		
Description	Each RunnableEntity mapped to an OsTask has a specific position within the task execution. For periodic activation this is the order of execution. For event driver activation this is the order of evaluation which actual RunnableEntity has to be executed.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteUsedOsAlarmRef [rte_sws_9024_Conf]		
Description	If an OsAlarm is used to activate the OsTask this RteEvent is mapped to it shall be referenced here.		
Multiplicity	0..1		
Type	Reference to OsAlarm		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteUsedOsEventRef [rte_sws_9025_Conf]		
Description	If an OsEvent is used to activate the OsTask this RteEvent is mapped to it shall be referenced here.		
Multiplicity	0..1		
Type	Reference to OsEvent		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteUsedOsSchTblExpiryPointRef [rte_sws_9026_Conf]		
Description	If an OsScheduleTableExpiryPoint is used to activate the OsTask this RteEvent is mapped to it shall be referenced here.		
Multiplicity	0..1		
Type	Reference to OsScheduleTableExpiryPoint		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteVirtuallyMappedToTaskRef [rte_sws_9027_Conf]		
Description	Optional reference to an OsTask where the activation of this RteEvent shall be evaluated. The actual execution of the Runnable Entity shall happen in the OsTask referenced by RteMappedToTaskRef.		
Multiplicity	0..1		
Type	Reference to OsTask		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.6.2 Rte Os Interaction

This section contains configuration items which are closely related to the interaction of the Rte with the Os.

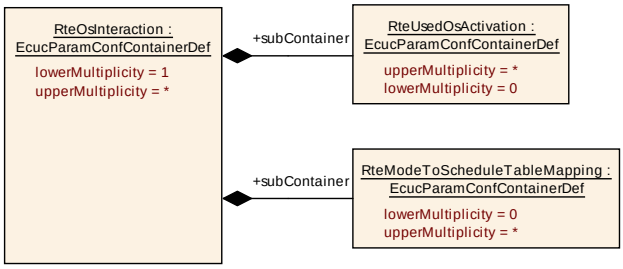


Figure 7.8: Specification of the Rte/Os Interaction

7.6.2.1 Activation using Os features

This is a collection of possible ways how the Rte might utilize Os to achieve various activation scenarios. The used Os objects are referenced in these configuration entities.

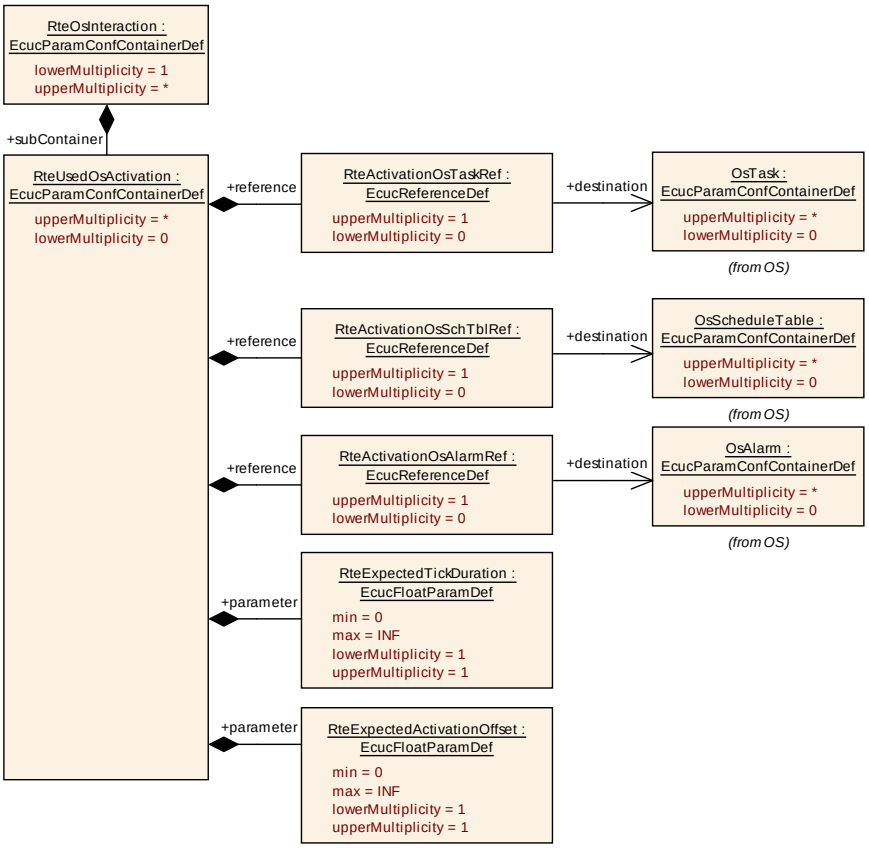


Figure 7.9: Configuration how activation is implemented

RteUsedOsActivation

SWS Item	[rte_sws_9060_Conf]
Container Name	RteUsedOsActivation
Description	Attributes used in the activation of OsTasks and Runnable Entities.
Configuration Parameters	

Name	RteActivationOsAlarmRef [rte_sws_9045_Conf]		
Description	Reference to an OsAlarm.		
Multiplicity	0..1		
Type	Reference to OsAlarm		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteActivationOsSchTblRef [rte_sws_9046_Conf]		
Description	Reference to an OsScheduleTable.		
Multiplicity	0..1		
Type	Reference to OsScheduleTable		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteActivationOsTaskRef [rte_sws_9047_Conf]		
Description	Reference to an OsTask.		
Multiplicity	0..1		
Type	Reference to OsTask		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteExpectedActivationOffset [rte_sws_9048_Conf]		
Description	Activation offset in seconds. Important: This is a requirement from the Rte towards the Os/Mcu setup. The Rte Generator shall assume this activation offset to be fulfilled.		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	0 .. INF		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteExpectedTickDuration [rte_sws_9049_Conf]		
Description	The expected tick duration in seconds which shall be configured to drive the OsScheduleTables or OsAlarm. Important: This is a requirement from the Rte towards the Os/Mcu setup. The Rte Generator shall assume this tick duration to be fulfilled.		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	0 .. INF		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.6.2.2 Modes and Schedule Tables

Optional configuration of the Rte to support the mapping of modes and Os' schedule tables.

[rte_sws_5146]⌈ The referenced schedule table of `RteModeScheduleTableRef` shall be activated if one of the modes referenced in `RteModeSchblMapModeDeclarationRef` is active in the `mode machine instances` from the references of

- `RteModeSchtblMapSwc` or
- `RteModeSchtblMapBsw`.

⌋*()*

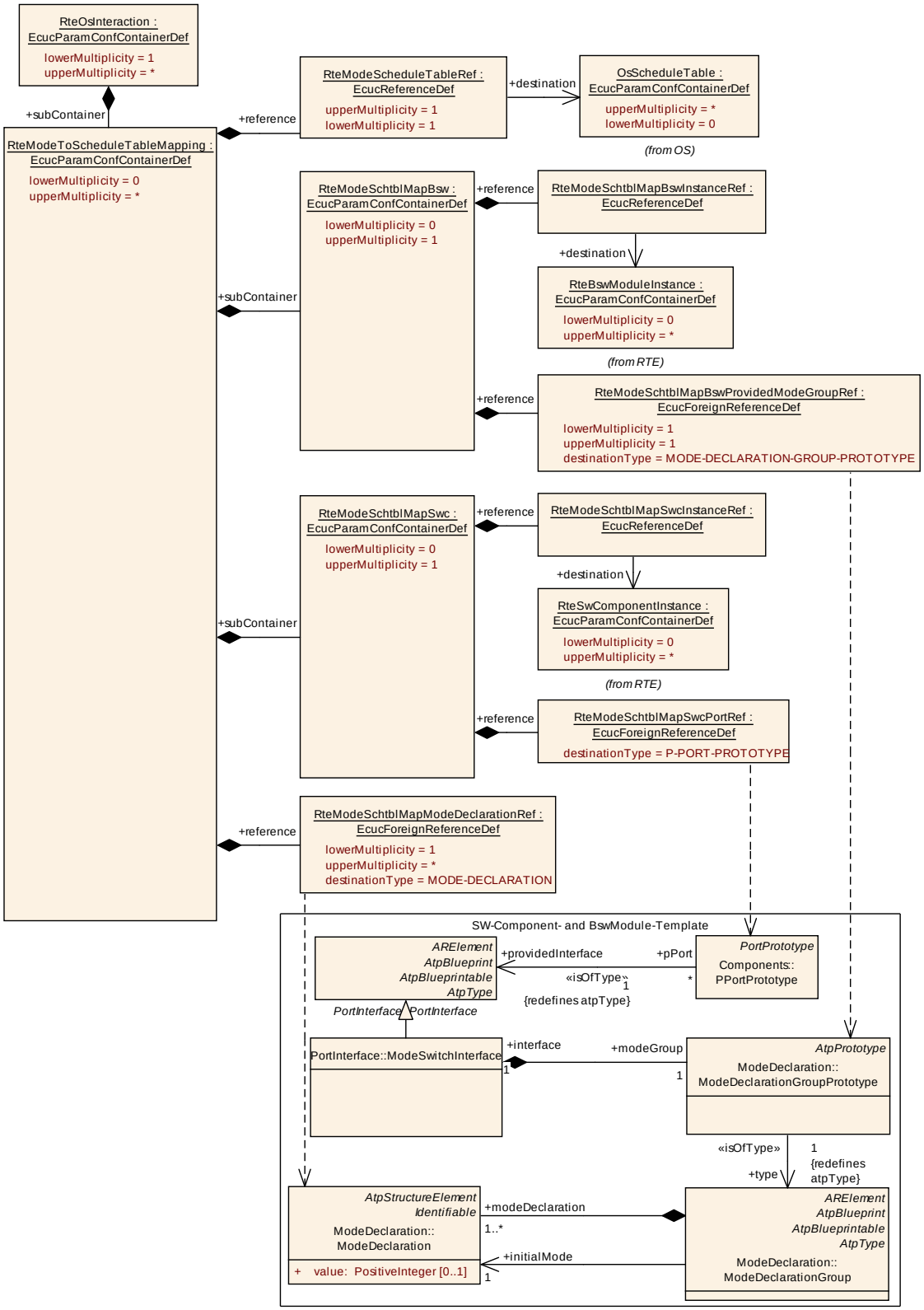


Figure 7.10: Configuration how modes are interacting with schedule tables

[rte_sws_2759]⌈ RTE shall reject a configuration, if the `RteModeSchblMapSwc-PortRef : EcucForeignReferenceDef` does not reference a `pPort` of the type of an `ModeSwitchInterface`. ⌋*()*

[rte_sws_2760]⌈ RTE shall reject a configuration, if the `ModeDeclara-tionGroupPrototype` referenced by a `RteModeScgblMapBswProvided-ModeGroupRef:EcucForeignReferenceDef` is not in the role of a `provided-ModeGroup`. ⌋*()*

RteModeToScheduleTableMapping

SWS Item	[rte_sws_9058_Conf]
Container Name	RteModeToScheduleTableMapping
Description	Provides configuration input in which Modes of a ModeDeclarionGroupPrototype of a Mode Manager a OsScheudleTable shall be active. The Mode Manager is either specified as a SwComponentPrototype (RteModeSchtblMapSwc) or as a BSW-Module (RteModeSchtblMapBsw).
Configuration Parameters	

Name	RteModeScheduleTableRef [rte_sws_9050_Conf]		
Description	Reference to the OsScheduleTable which shall be active in the specified RteModeSchblMapModeDeclarationRefs.		
Multiplicity	1		
Type	Reference to OsScheduleTable		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteModeSchtblMapModeDeclarationRef [rte_sws_9054_Conf]		
Description	Reference to the ModeDeclarations.		
Multiplicity	1..*		
Type	Foreign reference to MODE-DECLARATION		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
RteModeSchtblMapBsw	0..1	Specifies an instance of a ModeDeclarationGroupPrototype of a Bsw-Module.
RteModeSchtblMapSwc	0..1	Specifies an instance of a ModeDeclarationGroupPrototype of a SwComponentPrototype.

RteModeSchtblMapSwc

SWS Item	[rte_sws_9055_Conf]		
Container Name	RteModeSchtblMapSwc		
Description	Specifies an instance of a ModeDeclarationGroupPrototype of a SwComponentPrototype.		
Configuration Parameters			

Name	RteModeSchtblMapSwcInstanceRef [rte_sws_9056_Conf]		
Description	Reference to an instance specification of a SwComponentPrototype.		
Multiplicity	1		
Type	Reference to RteSwComponentInstance		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteModeSchtblMapSwcPortRef [rte_sws_9057_Conf]		
Description	Reference to the PPortPrototype of a SwComponentPrototype.		
Multiplicity	1		
Type	Foreign reference to P-PORT-PROTOTYPE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

RteModeSchtblMapBsw

SWS Item	[rte_sws_9051_Conf]		
Container Name	RteModeSchtblMapBsw		
Description	Specifies an instance of a ModeDeclarationGroupPrototype of a Bsw-Module.		
Configuration Parameters			

Name	RteModeSchtblMapBswInstanceRef [rte_sws_9052_Conf]		
Description	Reference to an instance specification of a Bsw-Module.		
Multiplicity	1		
Type	Reference to RteBswModuleInstance		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteModeSchtblMapBswProvidedModeGroupRef [rte_sws_9053_Conf]		
Description	Reference to an instance of a ModeDeclarationGroupPrototype of a Bsw-Module.		
Multiplicity	1		
Type	Foreign reference to MODE-DECLARATION-GROUP-PROTOTYPE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.6.3 Exclusive Area implementation

The RTE Generator can be configured to implement a different data consistency mechanism for each `ExclusiveArea` defined for an AUTOSAR software-component.

In figure 7.11 the configuration of the actually selected data consistency mechanism is shown.

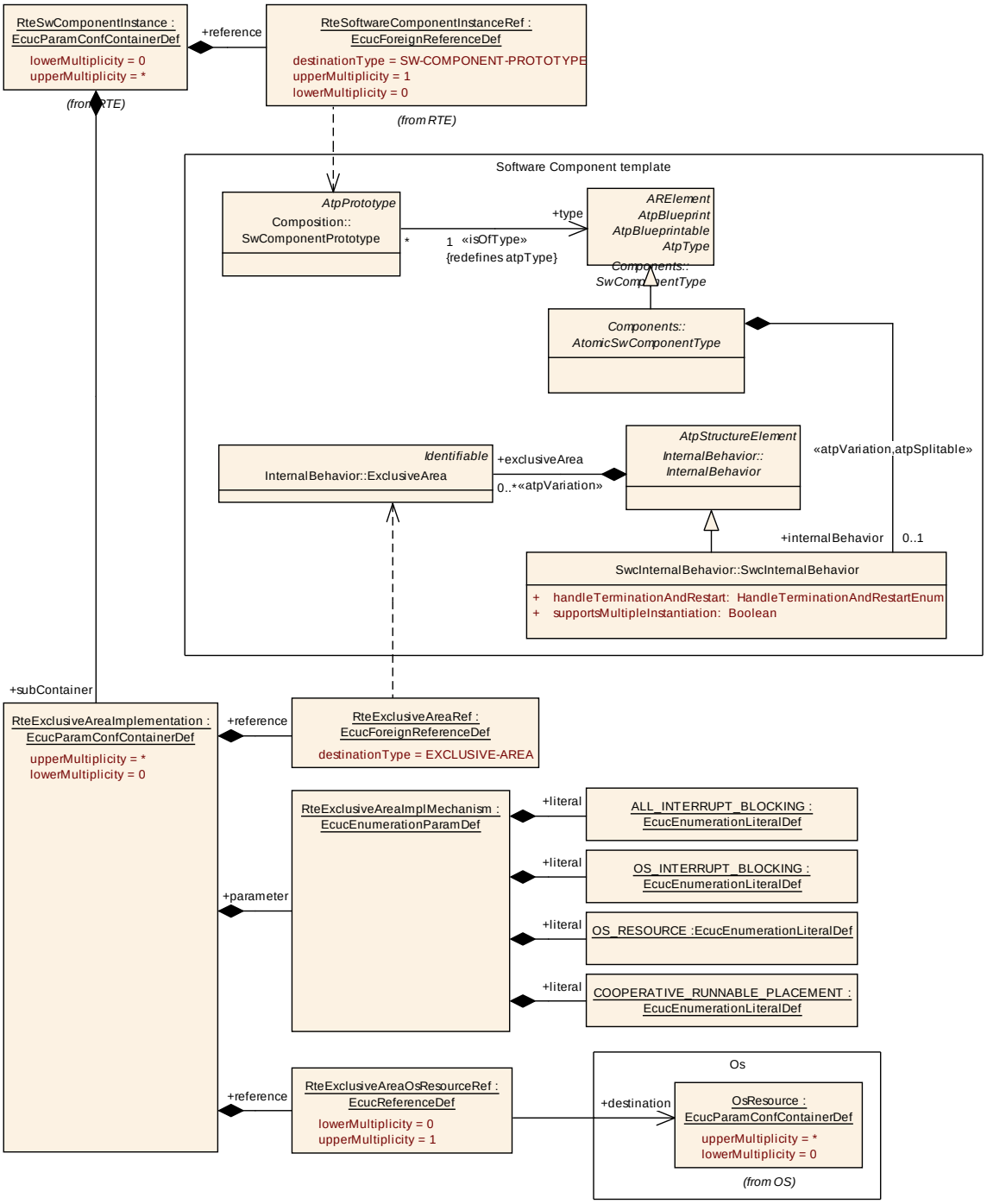


Figure 7.11: Configuration of the ExclusiveArea implementation

RteExclusiveAreaImplementation

SWS Item	[rte_sws_9030_Conf]
Container Name	RteExclusiveAreaImplementation
Description	Specifies the implementation to be used for the data consistency of this ExclusiveArea.
Configuration Parameters	

Name	RteExclusiveAreaImplMechanism [rte_sws_9029_Conf]		
Description	To be used implementation mechanism for the specified ExclusiveArea.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	ALL_INTERRUPT_BLOCKING		
	COOPERATIVE_RUNNABLE_PLACEMENT		
	OS_INTERRUPT_BLOCKING		
	OS_RESOURCE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteExclusiveAreaOsResourceRef [rte_sws_9031_Conf]		
Description	Optional reference to an OsResource in case RteExclusiveAreaImplMechanism is configured to OS_RESOURCE for this ExclusiveArea.		
Multiplicity	0..1		
Type	Reference to OsResource		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteExclusiveAreaRef [rte_sws_9032_Conf]		
Description	Reference to the ExclusiveArea.		
Multiplicity	1		
Type	Foreign reference to EXCLUSIVE-AREA		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.6.4 NVRam Allocation

The configuration of the NVRam access does involve several templates, because it closes the gap between the AUTOSAR software-components, the NVRAM Manager Services and the BSW Modules.

In figure 7.12 the related information from the AUTOSAR Software Component Template is shown.

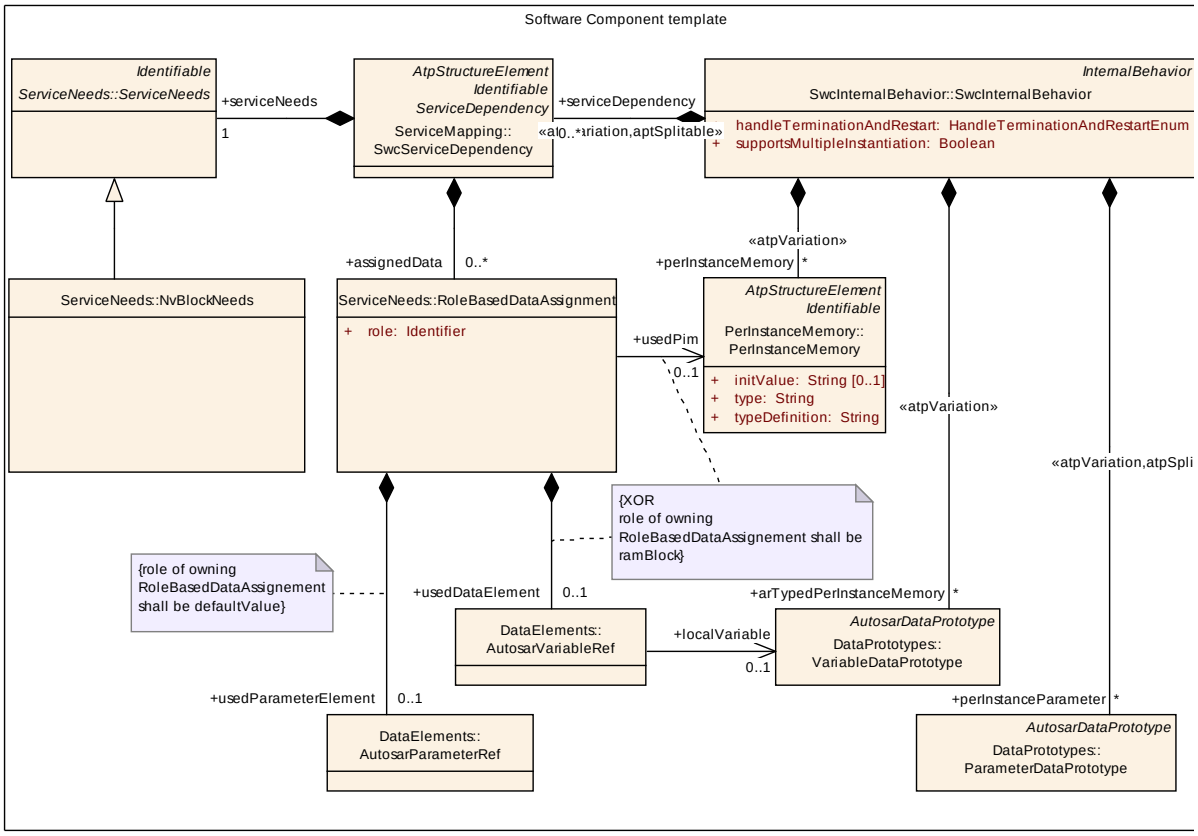


Figure 7.12: software-component information of NVRam Service needs

In figure 7.13 the ECU Configuration part of the NVRam allocation is shown. It relates the software-components' `SwcServiceDependency` and `NvBlockNeeds` information with the NVRam Managers `NvmBlockDescriptor` and the linker symbols of the RAM and ROM sections to be used.

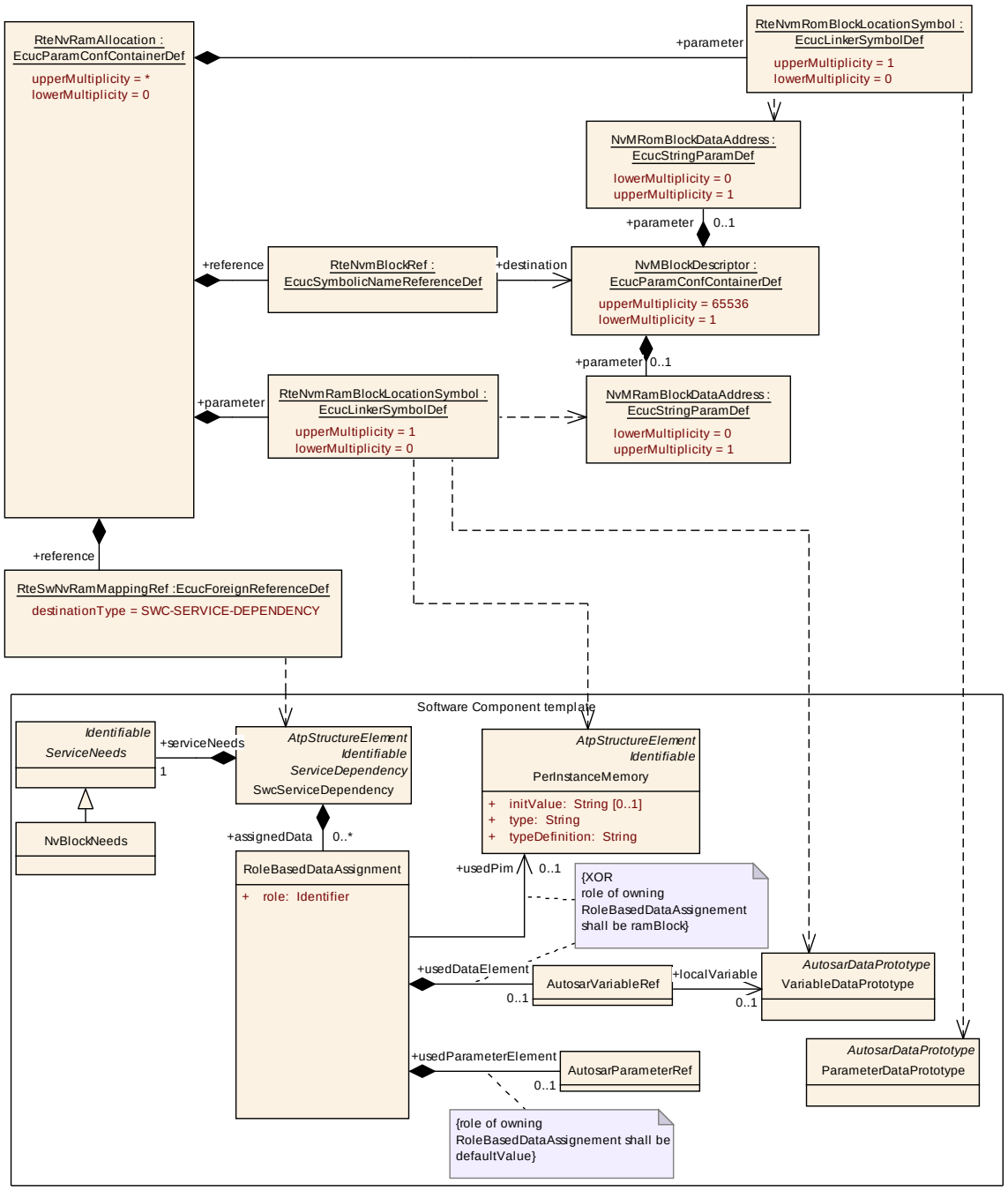


Figure 7.13: ECU Configuration of the NVRam Service

RteNvRamAllocation

SWS Item	[rte_sws_9040_Conf]
Container Name	RteNvRamAllocation
Description	Specifies the relationship between the AtomicSwComponentType's NVRAMMapping / NVRAM needs and the NvM module configuration.
Configuration Parameters	

Name	RteNvmBlockRef [rte_sws_9041_Conf]		
Description	Reference to the used NvM block for storage of the NVRAMMapping information.		
Multiplicity	1		
Type	Symbolic name reference to NvMBlockDescriptor		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteNvmRamBlockLocationSymbol [rte_sws_9042_Conf]		
Description	This is the name of the linker object name where the NVRam Block will be mirrored by the Nvm. This symbol will be resolved into the parameter "NvmRamBlockDataAddress" from the "NvmBlockDescriptor".		
Multiplicity	0..1		
Type	EcucLinkerSymbolDef		
Default Value			
Regular Expression			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteNvmRomBlockLocationSymbol [rte_sws_9043_Conf]		
Description	This is the name of the linker object name where the NVRom Block will be accessed by the Nvm. This symbol will be resolved into the parameter "NvmRomBlockDataAddress" from the "NvmBlockDescriptor".		
Multiplicity	0..1		
Type	EcucLinkerSymbolDef		
Default Value			
Regular Expression			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteSwNvRamMappingRef [rte_sws_9044_Conf]		
Description	Reference to the SwSeriveDependency which is used to specify the NvBlockNeeds.		
Multiplicity	1		
Type	Foreign reference to SWC-SERVICE-DEPENDENCY		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.6.5 SWC Trigger queuing

This configuration determine the size of the queue queuing the issued triggers.

The `RteExternalTriggerConfig` container and `RteInternalTriggerConfig` container is defined in the context of the `RteSwComponentInstance` which already predefines the context of the `Trigger` / `InternalTriggeringPoint`.

[rte_sws_ext_7598] The references `RteSwcTriggerSourceRef` has to be consistent with the `RteSoftwareComponentInstanceRef`. This means the referenced `Trigger` / `InternalTriggeringPoint` has to belong to the `AtomicSwComponentType` which is referenced by the related `SwComponentPrototype`.

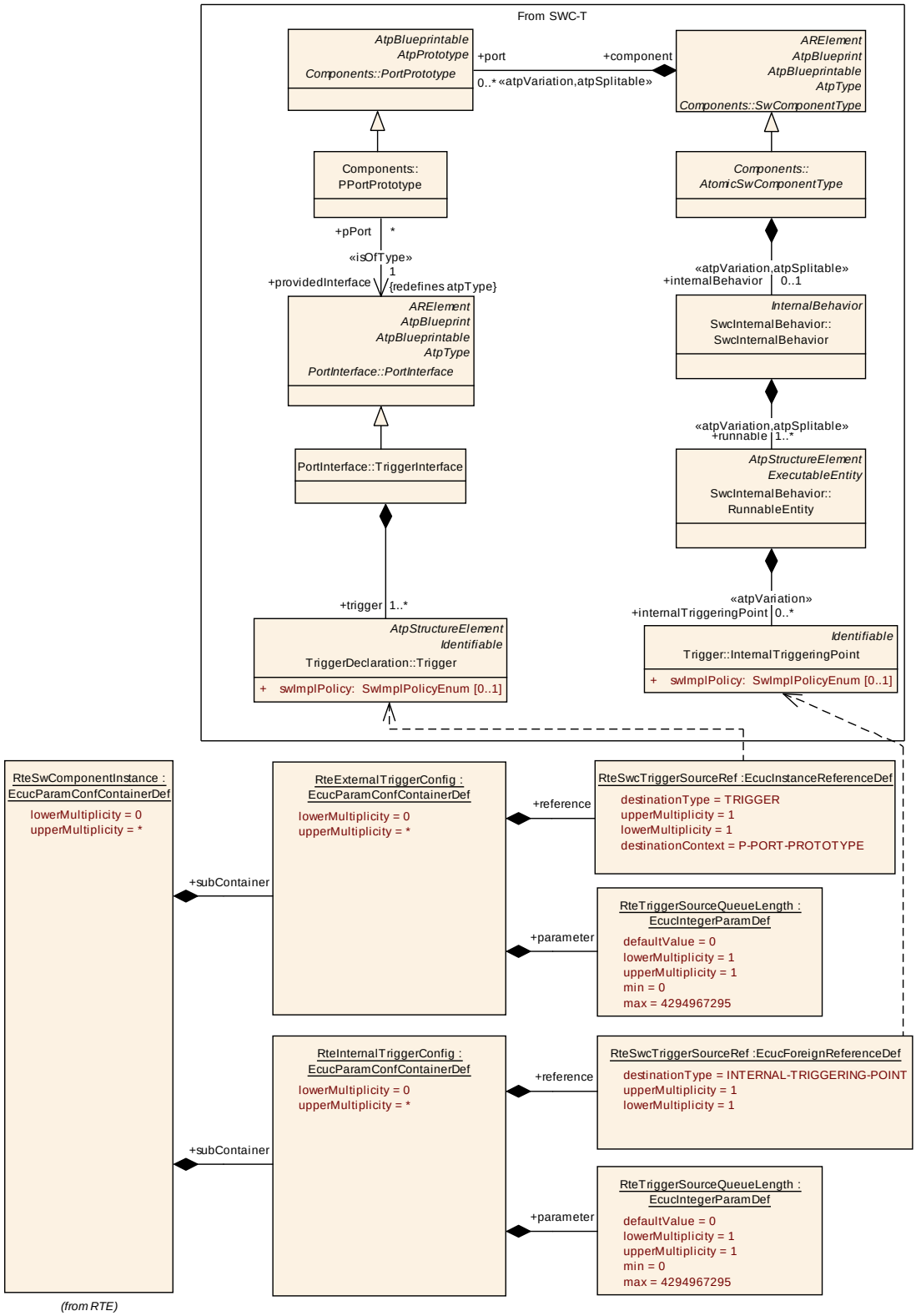


Figure 7.14: Configuration of SWC Trigger queuing

RteExternalTriggerConfig

SWS Item	[rte_sws_9105_Conf]		
Container Name	RteExternalTriggerConfig		
Description	Defines the configuration of External Trigger Event Communication for Software Components		
Configuration Parameters			

Name	RteSwcTriggerSourceRef [rte_sws_9106_Conf]		
Description	Reference to a Trigger instance in the pPortPrototype of the related component instance. The referenced Trigger instance has to belong to the same software component instance as the RteSwComponentInstance owning this parameter configures.		
Multiplicity	1		
Type	Instance reference to TRIGGER context: P-PORT-PROTOTYPE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteTriggerSourceQueueLength [rte_sws_9095_Conf]		
Description	Length of trigger queue on the trigger source side. The queue is implemented by the RTE. A value greater or equal to 1 requests an queued behavior. Setting the value of RteTriggerSourceQueueLength to 0 requests an none queued implementation of the trigger communication. If there is no RteTriggerSourceQueueLength configured for a Trigger Emitter the default value of 0 applies as well.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default Value	0		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

RteInternalTriggerConfig

SWS Item	[rte_sws_9096_Conf]		
Container Name	RteInternalTriggerConfig		
Description	Defines the configuration of Inter Runnable Triggering for Software Components		
Configuration Parameters			

Name	RteSwcTriggerSourceRef [rte_sws_9097_Conf]		
Description	Reference to an InternalTriggeringPoint of the related component instance. The referenced InternalTriggeringPoint has to belong to the same software component instance as the RteSwComponentInstance owning this parameter configures.		
Multiplicity	1		
Type	Foreign reference to INTERNAL-TRIGGERING-POINT		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteTriggerSourceQueueLength [rte_sws_9098_Conf]		
Description	Length of trigger queue on the trigger source side. The queue is implemented by the RTE. A value greater or equal to 1 requests an queued behavior. Setting the value of RteTriggerSourceQueueLength to 0 requests an none queued implementation of the trigger communication. If there is no RteTriggerSourceQueueLength configured for a Trigger Emitter the default value of 0 applies as well.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default Value	0		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.7 Handling of Software Component types

7.7.1 Selection of Software-Component Implementation

During the system development there is no need to select the actual implementation which will be later integrated on one ECU. Therefore the *ECU Extract of System Description* may not specify the `SwcImplementation` information yet.

For RTE Generation the information about the to be used `SwcImplementation` for each `SwComponentType` needs be provided to the RTE Generator (regardless whether the information is from the Ecu Extract or the Ecu Configuration.

The mapping of `SwcImplementation` to `SwComponentType` is done in the Ecu Configuration of the Rte using the two references `RteComponentTypeRef` and `RteImplementationRef` (see figure 7.15). For the mapping in the Ecu Extract please refer to the Specification of the System Template [8].

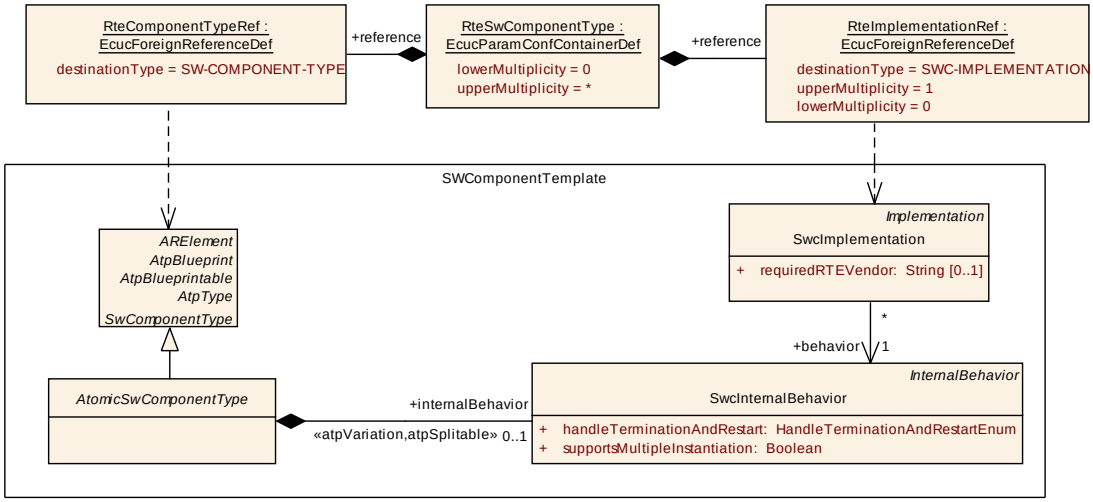


Figure 7.15: Selection of the Implementation for an AtomicSwComponentType

7.7.2 Component Type Calibration

In the AUTOSAR Software Component Template two places may provide calibration data: the `ParameterSwComponentType` and the `AtomicSwComponentType` (or more precisely the subclasses of `AtomicSwComponentType`). Whether the calibration is enabled for a specific `SwComponentType` can be configured as shown in figure 7.16.

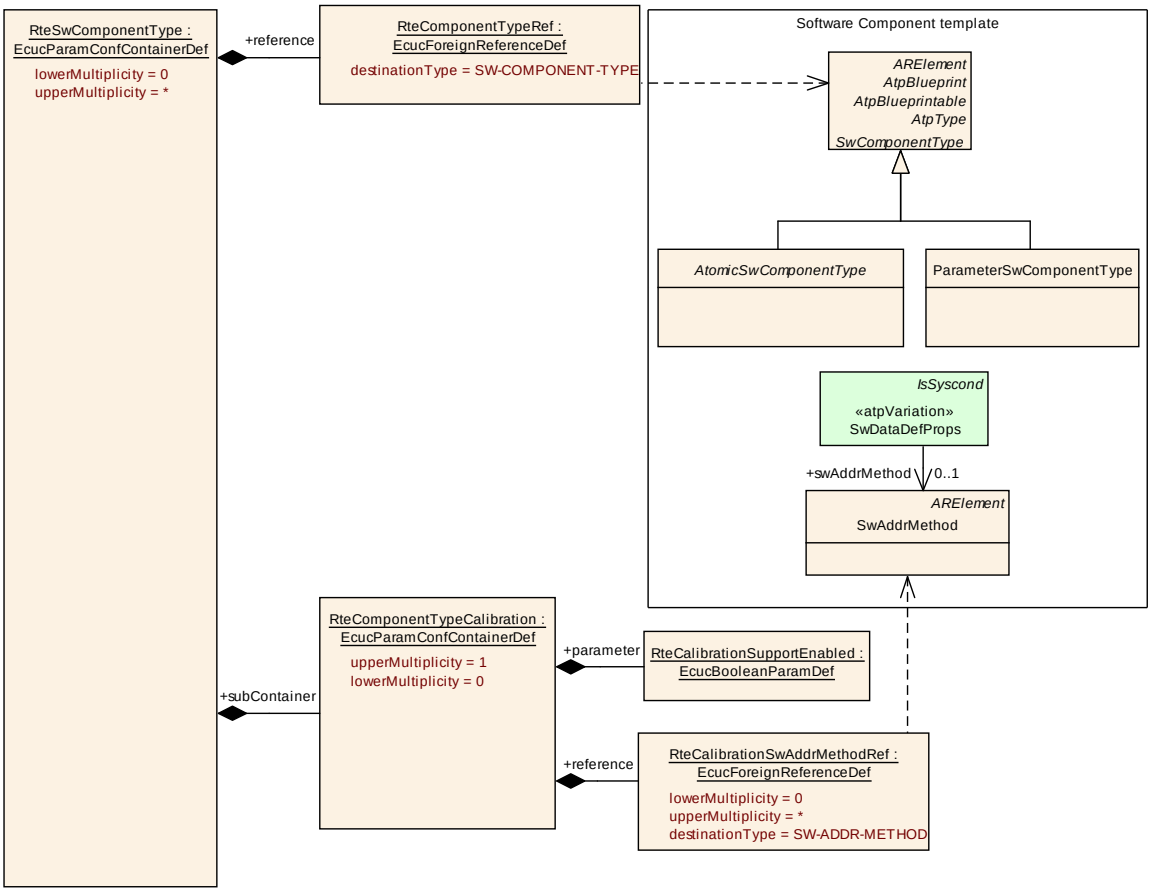


Figure 7.16: Configuration of the calibration for the ParameterSwComponentType

The foreign reference `ComponentTypeRef` identifies the `SwComponentType` (which is limited to `ParameterSwComponentType` and `AtomicSwComponentType`). The boolean parameter `CalibrationSupportEnabled` specifies whether calibration shall be enabled for the specified `SwComponentType`.

[rte_sws_5145]⌈ For a `ParameterDataPrototype` of the referenced `ComponentTypeRef` software calibration support shall be enabled if the parameter `CalibrationSupportEnabled` is set to *true* and in the corresponding container `RteComponentTypeCalibration`

- not a single `RteCalibrationSwAddrMethodRef` exists or
- a reference `RteCalibrationSwAddrMethodRef` to the `SwAddrMethod` of the `ParameterDataPrototype` exists.

⌋*(RTE00154, RTE00156, RTE00158)*

RteComponentTypeCalibration

SWS Item	[rte_sws_9039_Conf]		
Container Name	RteComponentTypeCalibration		
Description	Specifies for each ParameterSwComponentType or AtomicSwComponentType whether calibration is enabled. If references to SwAddrMethod are provided in RteCalibrationSwAddrMethodRef only ParameterDataPrototypes with the referenced SwAddrMethod shall have software calibration support enabled.		
Configuration Parameters			

Name	RteCalibrationSupportEnabled [rte_sws_9037_Conf]		
Description	Enables calibration support for the specified ParameterSwComponentType or AtomicSwComponentType.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteCalibrationSwAddrMethodRef [rte_sws_9038_Conf]		
Description	Reference to the SwAddrMethod for which software calibration support shall be enabled.		
Multiplicity	0..*		
Type	Foreign reference to SW-ADDR-METHOD		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.8 Implicit communication configuration

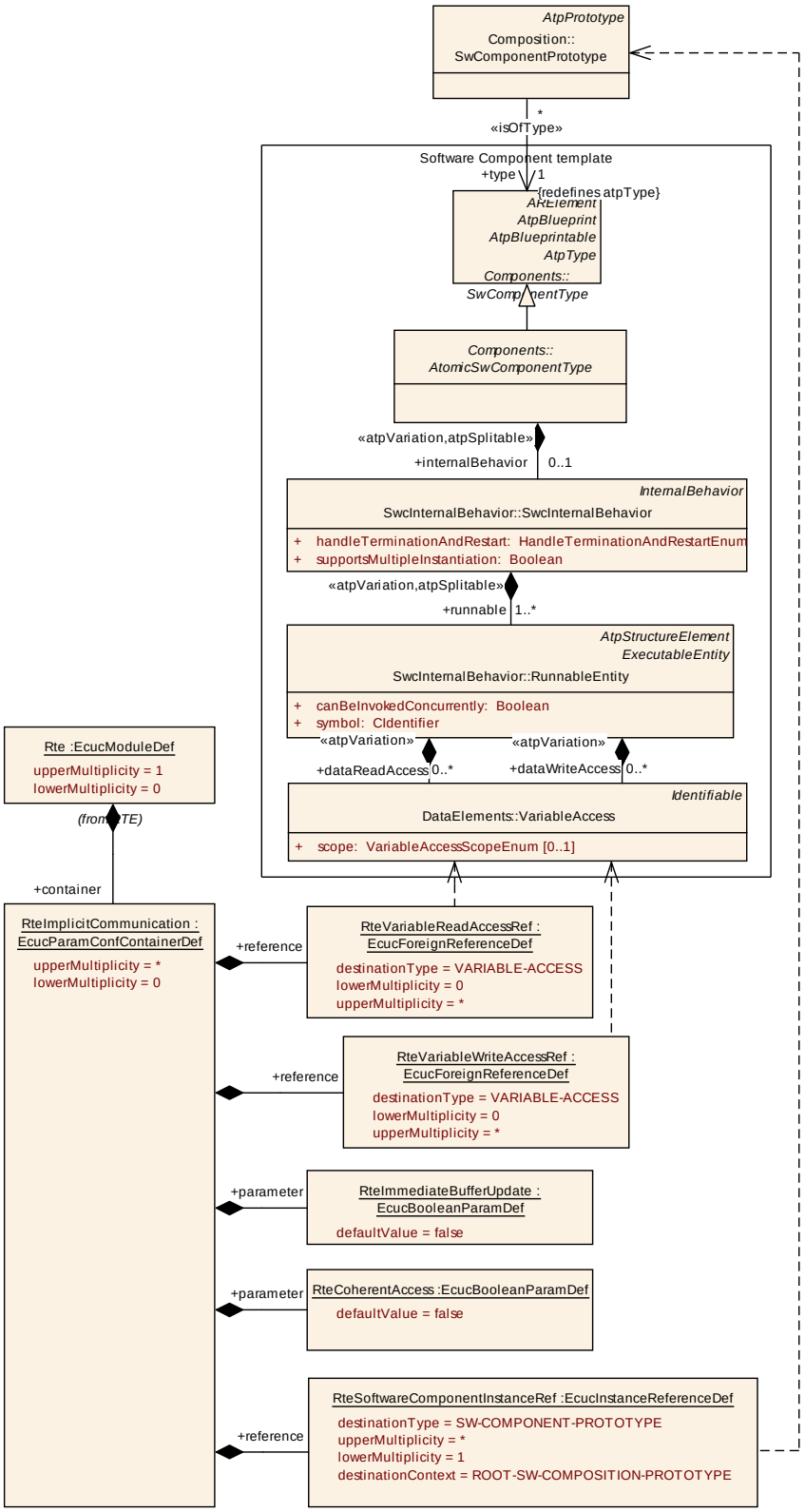


Figure 7.17: Configuration of the implicit communication

RteImplicitCommunication

SWS Item	[rte_sws_9034_Conf]
Container Name	RteImplicitCommunication
Description	Configuration of the Implicit Communication behavior to be generated.
Configuration Parameters	

Name	RteCoherentAccess [rte_sws_9091_Conf]		
Description	If set to true the referenced VariableAccess'es of this RteImplicitCommunication container are in one CoherencyGroup. Data values for Coherent Implicit Read Access'es are read before the first reading RunnbaleEntity starts and are stable during the execution of all the reading RunnableEntitys; except Coherent Implicit Write Access'es belongs to the same Coherency Group. Data values written by Coherent Implicit Write Access'es are available for readers not belonging to the Coherency Group after the last writing RunnableEntity has terminated. Please note that a Coherent Implicit Data Access can be defined for VariableAccess'es to same and different VariableDataElements. Nevertheless all Coherent Implicit Data Access'es of one Coherency Group have to be executed in the same task.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteImmediateBufferUpdate [rte_sws_9033_Conf]		
Description	If set to true the RTE will perform preemption area specific buffer update immediately before (for VariableAccess in the role dataReadAccess) resp. after (for VariableAccess in the role dataWriteAccess) Runnable execution.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteSoftwareComponentInstanceRef [rte_sws_9090_Conf]		
Description	Reference to a SwComponentPrototype. This denotes the instances of the VariableAccess belonging to the RteImplicitCommunication.		
Multiplicity	1..*		
Type	Instance reference to SW-COMPONENT-PROTOTYPE context: ROOT-SW-COMPOSITION-PROTOTYPE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteVariableReadAccessRef [rte_sws_9035_Conf]		
Description	Reference to the VariableAccess in the dataReadAccess role.		
Multiplicity	0..*		
Type	Foreign reference to VARIABLE-ACCESS		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteVariableWriteAccessRef [rte_sws_9036_Conf]		
Description	Reference to the VariableAccess in the dataWriteAccess role.		
Multiplicity	0..*		
Type	Foreign reference to VARIABLE-ACCESS		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

Please note, that `RteImplicitCommunication` is defined as a container of `RteEcucModuleDef` to support the creation of the ECU Configuration Parameter Values related to `RteImplicitCommunication` independent from the other ECU Configuration Parameter Values. Typically the need for `Coherent Implicit Data Accesses` is known by the vendor of a set of software components. As long as `shortNames` of the `RootSwCompositionPrototype` and the referenced `CompositionSwComponentType` - describing the software of a flat ECU Extract - are known the ECU Configuration Parameter Values related to `RteImplicitCommunication` can be prescribed. In this case it is preferable to use relative references to the Vendor Specific Module Definition (VSMD), to `RootSwCompositionPrototype` and `CompositionSwComponentType` describing the software of a flat ECU Extract. With this relative references the ECU Configuration Parameter Values are independent from `ARPackage` structure only known by the ECU integrator. Nevertheless the `shortName` and location of of the `EcucModuleConfigurationValues` must be defined upfront.

7.9 Communication infrastructure

The configuration of the communication infrastructure (interaction of the RTE with the Com-Stack) is entirely predetermined by the ECU Extract provided as an input. The required input can be found in the AUTOSAR System Template [8] sections "Data Mapping" and "Communication".

In case the RTE does utilize the Com module for intra-ECU communication it is up to the vendor-specific configuration of the RTE to ensure configuration consistency.

7.10 Configuration of the BSW Scheduler

The configuration of the BSW Scheduler part of the RTE is shown in the overview in figure 7.18.

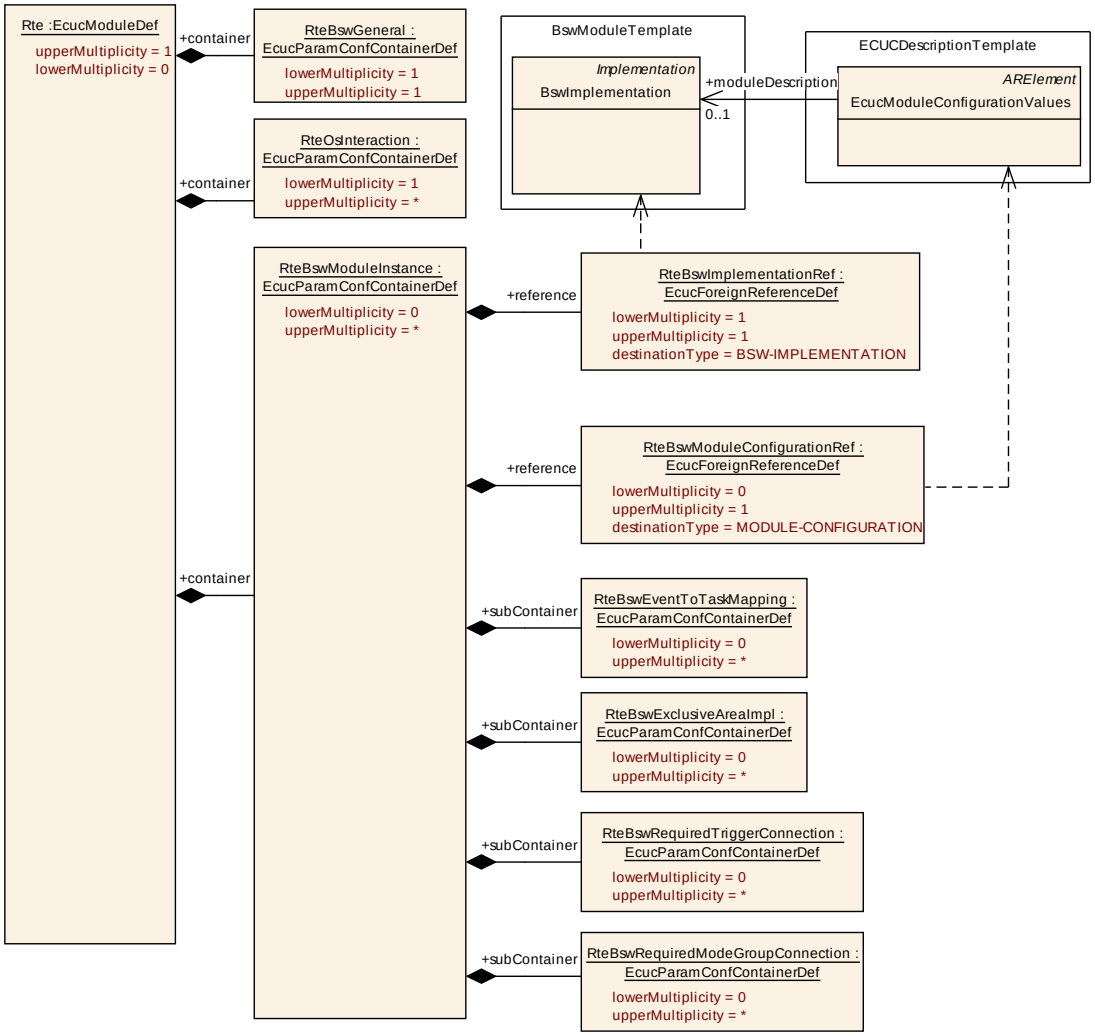


Figure 7.18: Configuration of BSW Scheduler overview

7.10.1 BSW Scheduler General configuration

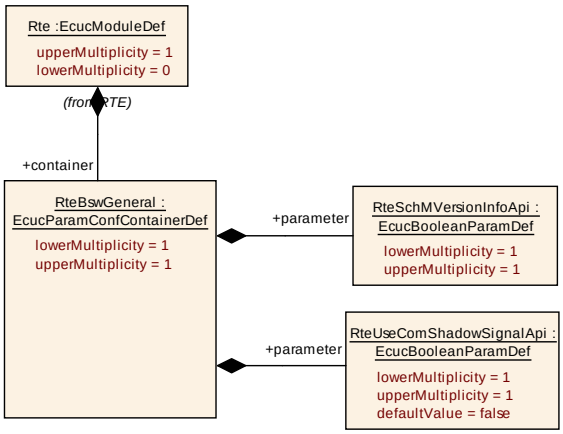


Figure 7.19: General configuration of BSW Scheduler

RteBswGeneral

SWS Item	[rte_sws_9061_Conf]
Container Name	RteBswGeneral
Description	General configuration parameters of the Bsw Scheduler section.
Configuration Parameters	

Name	RteSchMVersionInfoApi [rte_sws_9062_Conf]		
Description	Enables the generation of the SchM_GetVersionInfo() API.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteUseComShadowSignalApi [rte_sws_9107_Conf]		
Description	This parameter defines whether the ComShadowSignalAPIs ((Com_UpdateShadowSignal, Com_InvalidateShadowSignal, Com_ReceiveShadowSignal) are used or not. If this parameter is set to true the ShadowSignal APIs and Signal APIs (Com_SendSignal, Com_InvalidateSignal, Com_ReceiveSignal) are used. If this parameter is set to false only the Signal APIs (Com_SendSignal, Com_InvalidateSignal, Com_ReceiveSignal) are used.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.10.2 BSW Module Instance configuration

RteBswModuleInstance

SWS Item	[rte_sws_9002_Conf]
Container Name	RteBswModuleInstance
Description	Represents one instance of a Bsw-Module configured on one ECU.
Configuration Parameters	

Name	RteBswImplementationRef [rte_sws_9066_Conf]		
Description	Reference to the BswImplementation for which the Rte /SchM is configured.		
Multiplicity	1		
Type	Foreign reference to BSW-IMPLEMENTATION		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswModuleConfigurationRef [rte_sws_9001_Conf]		
Description	Reference to the ECU Configuration Values provided for this BswImplementation.		
Multiplicity	0..1		
Type	Foreign reference to MODULE-CONFIGURATION		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Included Containers		
Container Name	**Multiplicity**	**Scope / Depedency**
RteBswEventToTask Mapping	0..*	Maps a BswSchedulableEntity onto one OsTask based on the activating BswEvent.
RteBswExclusiveArea Impl	0..*	Represents one ExclusiveArea of one BswImplementation. Used to specify the implementation means of this ExclusiveArea.
RteBswExternalTrigger Config	0..*	Defines the configuration of Inter Basic Software Module Entity Triggering
RteBswInternalTrigger Config	0..*	Defines the configuration of internal Basic Software Module Entity Triggering
RteBswRequiredMode GroupConnection	0..*	Defines the connection between one requiredModeGroup of this BSW Module instance and one providedModeGroup instance.
RteBswRequiredTrigger Connection	0..*	Defines the connection between one requiredTrigger of this BSW Module instance and one releasedTrigger instance.

7.10.2.1 BSW ExclusiveArea configuration

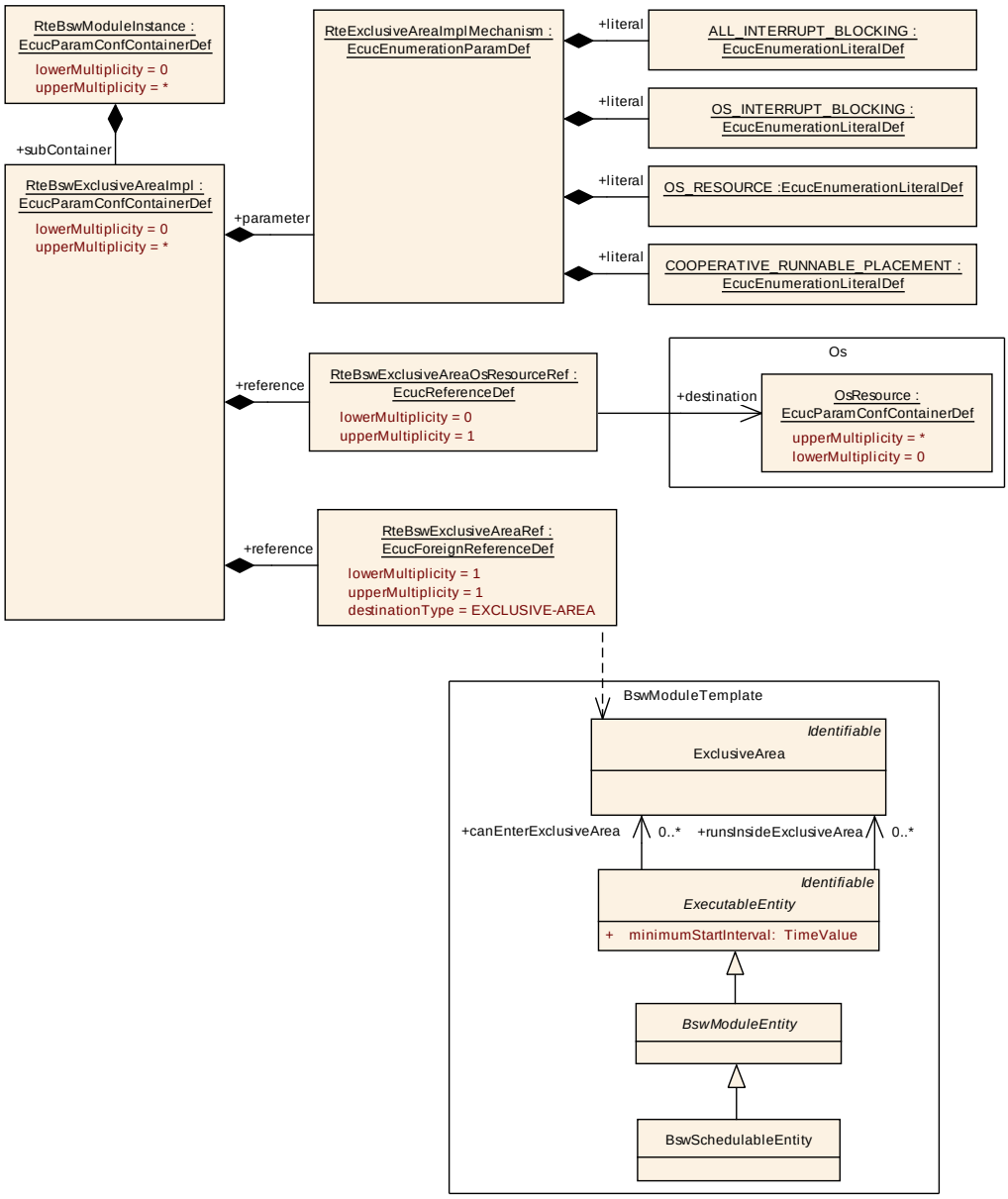


Figure 7.20: Configuration of BSW ExclusiveArea

RteBswExclusiveAreaImpl

SWS Item	[rte_sws_9072_Conf]
Container Name	RteBswExclusiveAreaImpl
Description	Represents one ExclusiveArea of one BswImplementation. Used to specify the implementation means of this ExclusiveArea.
Configuration Parameters	

Name	RteBswExclusiveAreaOsResourceRef [rte_sws_9073_Conf]		
Description	Optional reference to an OsResource in case RteExclusiveAreaImplMechanism is configured to OS_RESOURCE for this ExclusiveArea.		
Multiplicity	0..1		
Type	Reference to OsResource		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswExclusiveAreaRef [rte_sws_9074_Conf]		
Description	Reference to the ExclusiveArea for which the implementation mechanism shall be specified.		
Multiplicity	1		
Type	Foreign reference to EXCLUSIVE-AREA		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteExclusiveAreaImplMechanism [rte_sws_9029_Conf]		
Description	To be used implementation mechanism for the specified ExclusiveArea.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	ALL_INTERRUPT_BLOCKING		
	COOPERATIVE_RUNNABLE_PLACEMENT		
	OS_INTERRUPT_BLOCKING		
	OS_RESOURCE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.10.2.2 BswEvent to task mapping

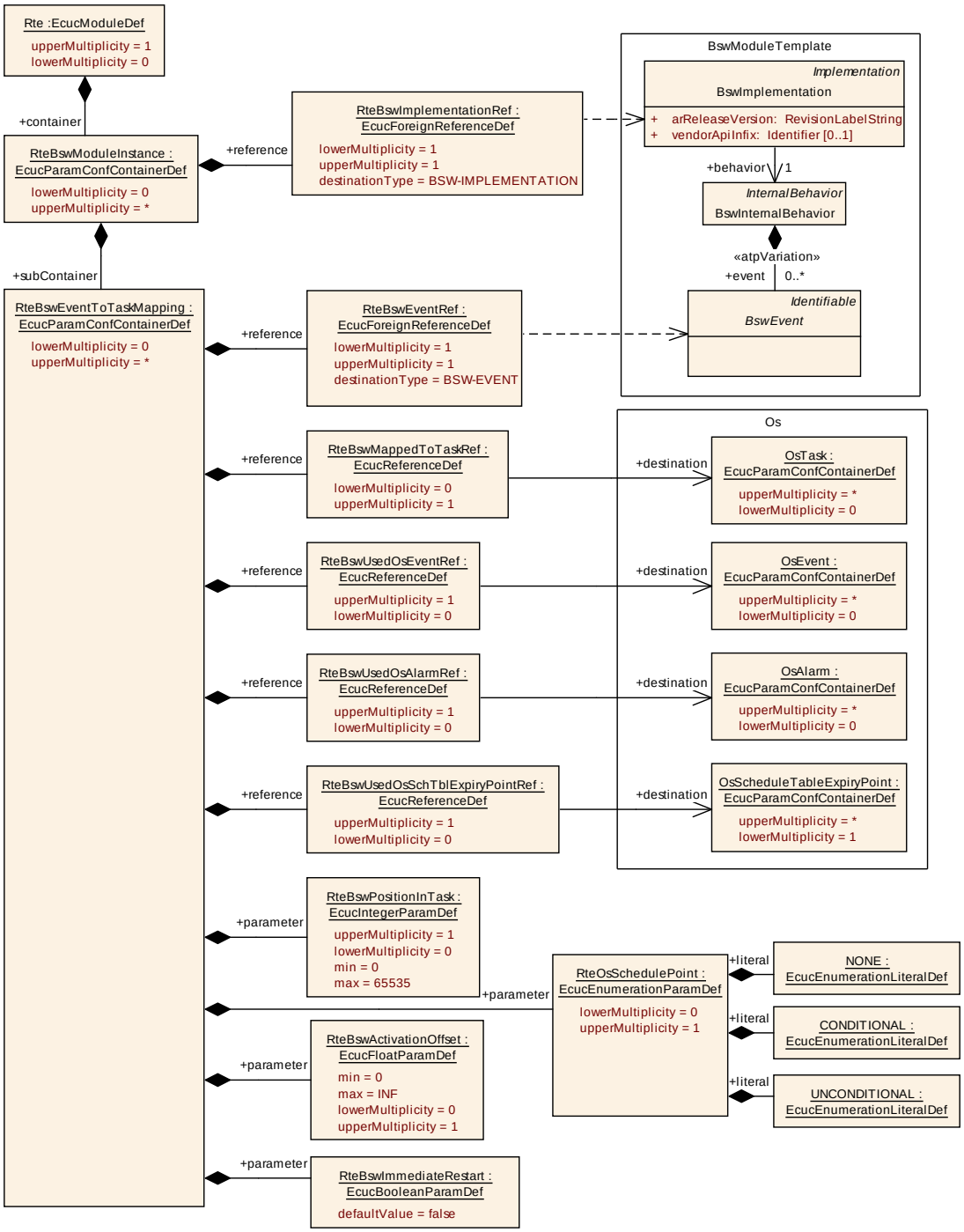


Figure 7.21: Configuration of BSW Event to Task Mapping

RteBswEventToTaskMapping

SWS Item	[rte_sws_9065_Conf]		
Container Name	RteBswEventToTaskMapping		
Description	Maps a BswSchedulableEntity onto one OsTask based on the activating BswEvent.		
Configuration Parameters			

Name	RteBswActivationOffset [rte_sws_9063_Conf]		
Description	Activation offset in seconds.		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	0 .. INF		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswEventRef [rte_sws_9064_Conf]		
Description	Reference to the BswEvent which is pointing to the BswSchedulableEntity being mapped. This allows a fine grained mapping of BswSchedulableEntites based on the activating BswEvent.		
Multiplicity	1		
Type	Foreign reference to BSW-EVENT		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswImmediateRestart [rte_sws_9093_Conf]		
Description	When RteBswImmediateRestart is set to true the BswSchedulableEntitiy shall be immediately re-started after termination if it was activated by this BswEvent while it was already started. This parameter shall not be set to true when the mapped BswEvent refers to a BswSchedulableEntitiy which minimumStartInterval attribute is > 0.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswMappedToTaskRef [rte_sws_9067_Conf]		
Description	Reference to the OsTask the BswSchedulableEntity activated by the RteBswEventRef is mapped to. If no reference to the OsTask is specified the BswSchedulableEntity activated by this BswEvent is executed in the context of the caller.		
Multiplicity	0..1		
Type	Reference to OsTask		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswPositionInTask [rte_sws_9068_Conf]		
Description	Each BswSchedulableEntity activation mapped to an OsTask has a specific position within the task execution. For periodic activation this is the order of execution. For event driver activation this is the order of evaluation which actual BswSchedulableEntity has to be executed.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default Value			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswUsedOsAlarmRef [rte_sws_9069_Conf]		
Description	If an OsAlarm is used to activate the OsTask this BswEvent is mapped to it shall be referenced here.		
Multiplicity	0..1		
Type	Reference to OsAlarm		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswUsedOsEventRef [rte_sws_9070_Conf]		
Description	If an OsEvent is used to activate the OsTask this BswEvent is mapped to it shall be referenced here.		
Multiplicity	0..1		
Type	Reference to OsEvent		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswUsedOsSchTblExpiryPointRef [rte_sws_9071_Conf]		
Description	If an OsScheduleTableExpiryPoint is used to activate the OsTask this BswEvent is mapped to it shall be referenced here.		
Multiplicity	0..1		
Type	Reference to OsScheduleTableExpiryPoint		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteOsSchedulePoint [rte_sws_9022_Conf]		
Description	Introduce a schedule point by explicitly calling Os Schedule service after the execution of the ExecutableEntity. The Rte generator is allowed to optimize several consecutive calls to Os schedule into one single call if the ExecutableEntity executions in between have been skipped. The absence of this parameter is interpreted as "NONE". It shall be considered an invalid configuration if the task is preemptable and the value of this parameter is not set to "NONE" or the parameter is absent.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CONDITIONAL	A Schedule Point shall be introduced at the end of the execution of this ExecutableEntity. The Schedule Point can be skipped if several Schedule Points would be called without any ExecutableEntity execution in between.	
	NONE	No Schedule Point shall be introduced at the end of the execution of this ExecutableEntity.	
	UNCONDITIONAL	A Schedule Point shall always be introduced at the end of the execution of this ExecutableEntity.	
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.10.2.3 BSW Trigger configuration

7.10.2.3.1 BSW Trigger connection

The `RteBswRequiredTriggerConnection` container is defined in the context of the `RteBswModuleInstance` which is the required trigger context. So the reference to the `RteBswRequiredTriggerRef` is sufficient to define the required trigger. For

the released trigger the tuple of `RteBswReleasedTriggerModInstRef` and `RteBswReleasedTriggerRef` is specified.

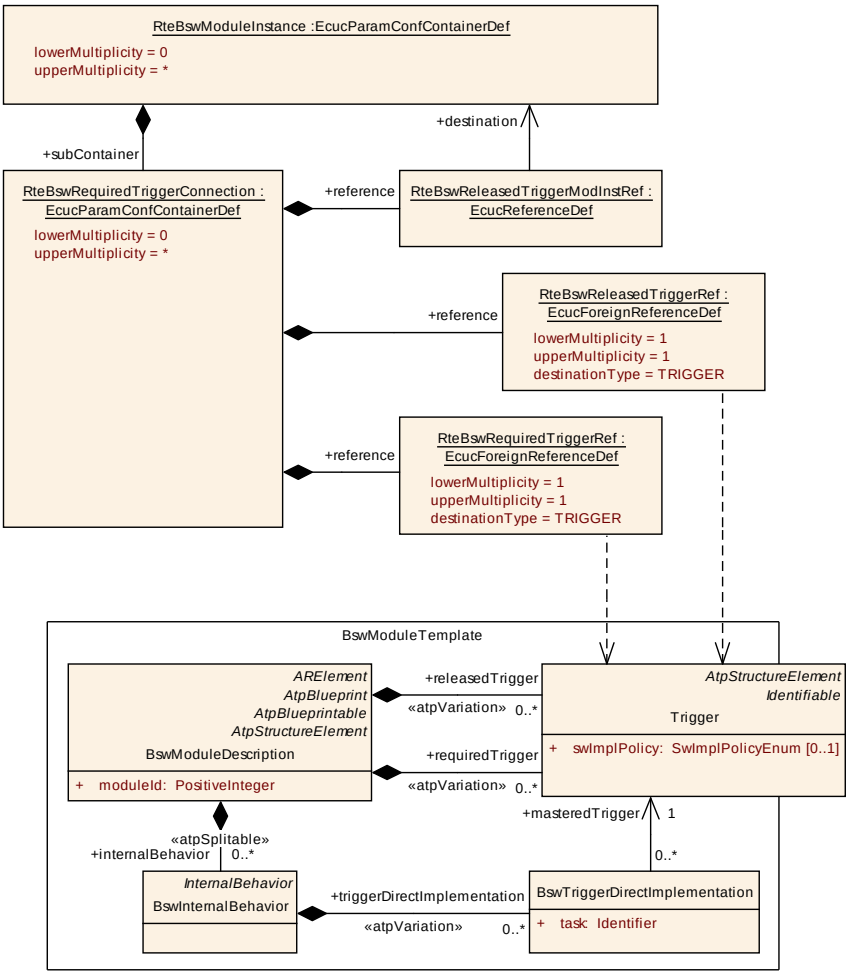


Figure 7.22: Configuration of BSW Trigger connection

RteBswRequiredTriggerConnection

SWS Item	[rte_sws_9077_Conf]
Container Name	RteBswRequiredTriggerConnection
Description	Defines the connection between one requiredTrigger of this BSW Module instance and one releasedTrigger instance.
Configuration Parameters	

Name	RteBswReleasedTriggerModInstRef [rte_sws_9075_Conf]		
Description	Reference to the RteBswModuleInstance configuration container which identifies the instance of the BSW Module. Used with the RteBswReleasedTriggerRef to unambiguously identify the Trigger instance.		
Multiplicity	1		
Type	Reference to RteBswModuleInstance		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswReleasedTriggerRef [rte_sws_9076_Conf]		
Description	References the releasedTrigger to which this requiredTrigger shall be connected.		
Multiplicity	1		
Type	Foreign reference to TRIGGER		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswRequiredTriggerRef [rte_sws_9078_Conf]		
Description	References one requiredTrigger which shall be connected to the releasedTrigger.		
Multiplicity	1		
Type	Foreign reference to TRIGGER		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.10.2.3.2 BSW Trigger queuing

This configuration determine the size of the queue queuing the issued triggers.

The `RteBswExternalTriggerConfig` container and `RteBswInternalTriggerConfig` container is defined in the context of the `RteBswModuleInstance` which already predefines the context of the provided `Trigger` / `BswInternalTriggeringPoint`.

[rte_sws_ext_7597] The references `RteBswTriggerSourceRef` has to be consistent with the `RteBswImplementationRef`. This means the referenced `Trigger` / `BswInternalTriggeringPoint` has to belong to the `BswModuleDescription` which is referenced by the related `BswImplementation`.

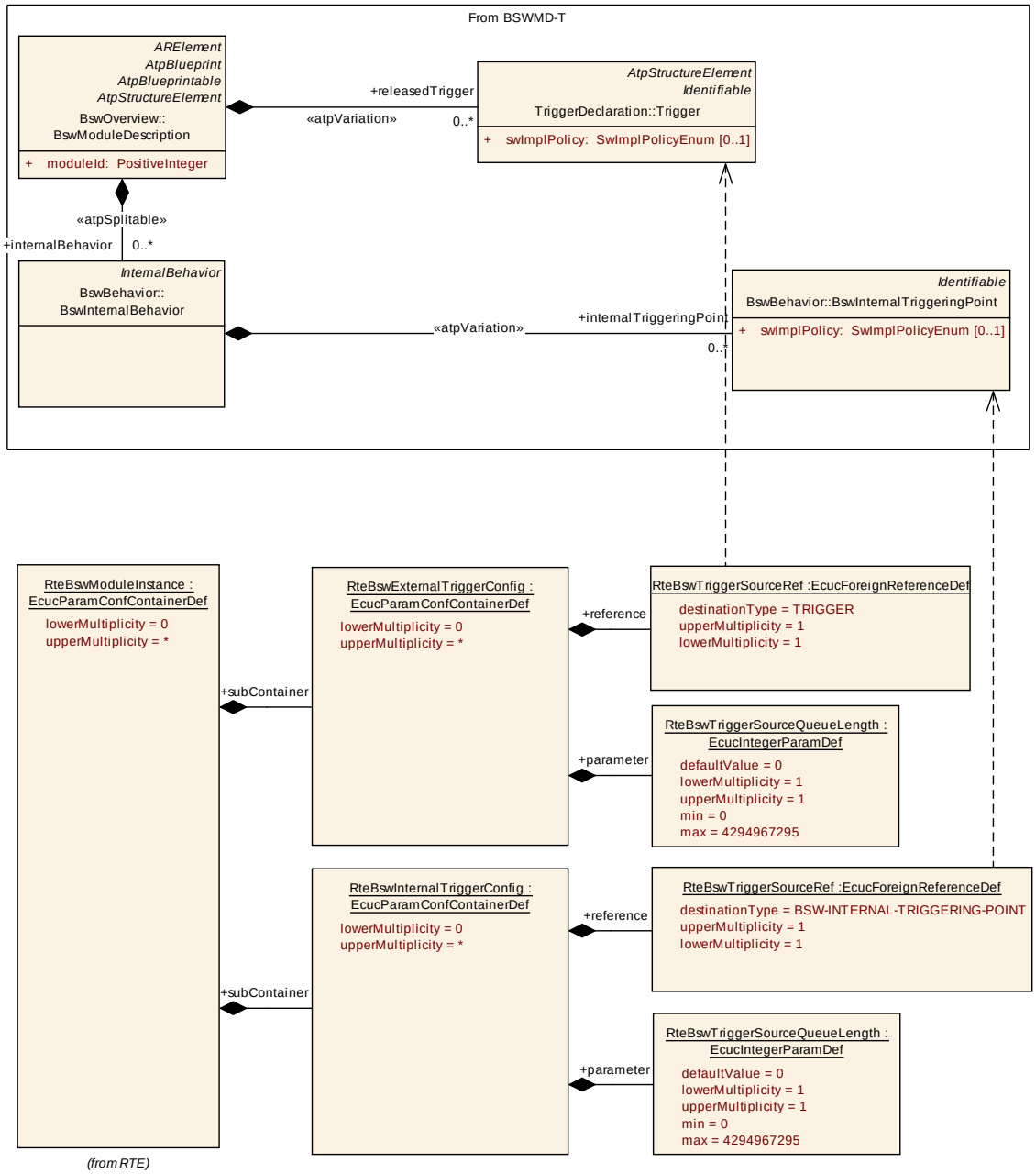


Figure 7.23: Configuration of BSW Trigger queuing

RteBswExternalTriggerConfig

SWS Item	[rte_sws_9099_Conf]
Container Name	RteBswExternalTriggerConfig
Description	Defines the configuration of Inter Basic Software Module Entity Triggering
Configuration Parameters	

Name	RteBswTriggerSourceQueueLength [rte_sws_9101_Conf]		
Description	Length of trigger queue on the trigger source side. The queue is implemented by the RTE. A value greater or equal to 1 requests an queued behavior. Setting the value of RteTriggerSourceQueueLength to 0 requests an none queued implementation of the trigger communication. If there is no RteBswTriggerSourceQueueLength configured for a Trigger Emitter the default value of 0 applies as well.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default Value	0		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswTriggerSourceRef [rte_sws_9100_Conf]		
Description	Reference to a Trigger instance in the role releasedTrigger of the related BSW Module instance. The referenced Trigger has to belong to the same BSW Module instance as the RteBswModuleInstance owning this parameter configures.		
Multiplicity	1		
Type	Foreign reference to TRIGGER		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

RteBswInternalTriggerConfig

SWS Item	[rte_sws_9102_Conf]
Container Name	RteBswInternalTriggerConfig
Description	Defines the configuration of internal Basic Software Module Entity Triggering
Configuration Parameters	

Name	RteBswTriggerSourceQueueLength [rte_sws_9104_Conf]		
Description	Length of trigger queue on the trigger source side. The queue is implemented by the RTE. A value greater or equal to 1 requests an queued behavior. Setting the value of RteTriggerSourceQueueLength to 0 requests an none queued implementation of the trigger communication. If there is no RteBswTriggerSourceQueueLength configured for a Trigger Emitter the default value of 0 applies as well.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default Value	0		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswTriggerSourceRef [rte_sws_9103_Conf]		
Description	Reference to a BswInternalTriggeringPoint of the related BSW Module instance. The referenced BswInternalTriggeringPoint has to belong to the same BSW Module instance as the RteBswModuleInstance owning this parameter configures.		
Multiplicity	1		
Type	Foreign reference to BSW-INTERNAL-TRIGGERING-POINT		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.10.2.4 BSW ModeDeclarationGroup configuration

The `RteBswRequiredModeGroupConnection` container is defined in the context of the `RteBswModuleInstance` which is the required mode group context. So the reference to the `RteBswRequiredModeGroupRef` is sufficient to define the required mode group. For the provided mode group the tuple of `RteBswProvidedModeGrp-ModInstRef` and `RteBswProvidedModeGroupRef` is specified.

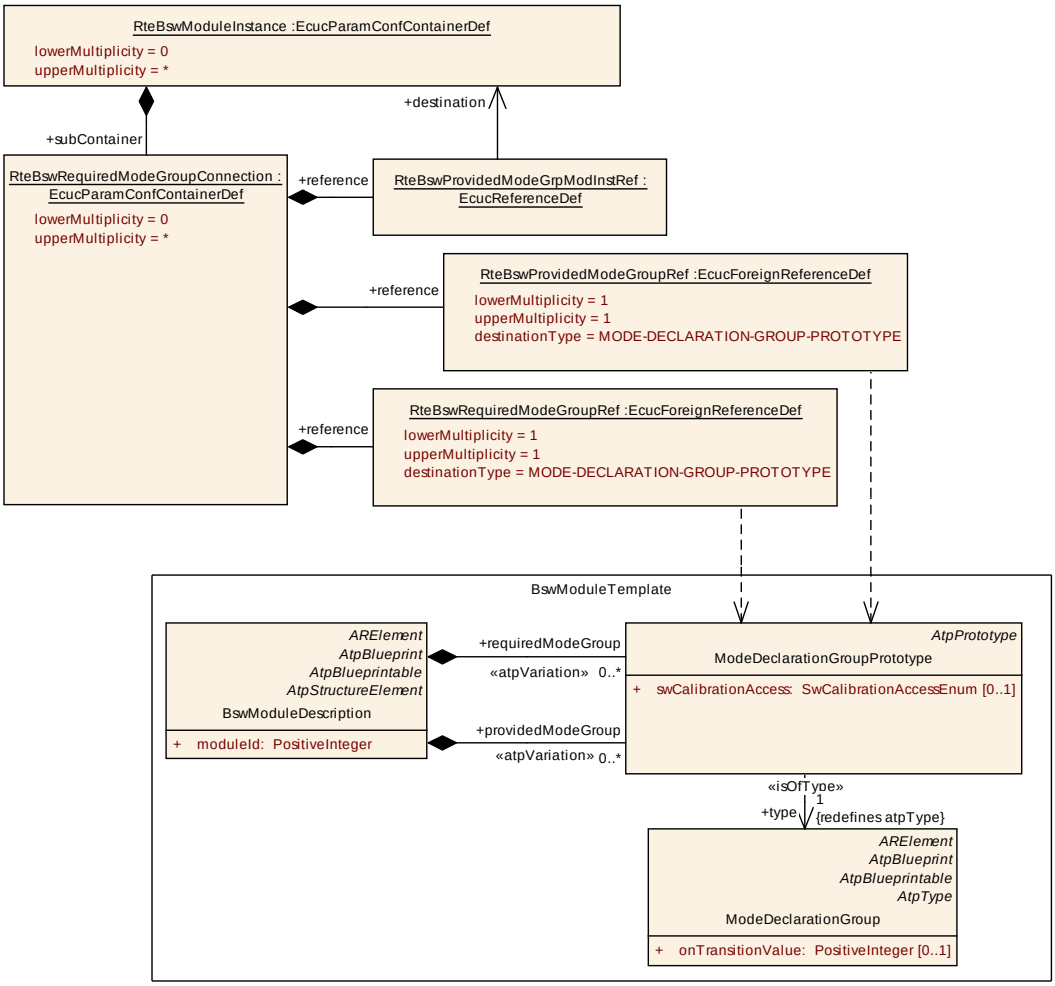


Figure 7.24: Configuration of BSW Scheduler overview

RteBswRequiredModeGroupConnection

SWS Item	[rte_sws_9081_Conf]
Container Name	RteBswRequiredModeGroupConnection
Description	Defines the connection between one requiredModeGroup of this BSW Module instance and one providedModeGroup instance.
Configuration Parameters	

Name	RteBswProvidedModeGroupRef [rte_sws_9079_Conf]		
Description	References the providedModeGroupPrototype to which this requiredModeGroup shall be connected.		
Multiplicity	1		
Type	Foreign reference to MODE-DECLARATION-GROUP-PROTOTYPE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswProvidedModeGrpModInstRef [rte_sws_9080_Conf]		
Description	Reference to the RteBswModuleInstance configuration container which identifies the instance of the BSW Module. Used with the RteBswProvidedModeGroupRef to unambiguously identify the ModeDeclarationGroupPrototype instance.		
Multiplicity	1		
Type	Reference to RteBswModuleInstance		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	RteBswRequiredModeGroupRef [rte_sws_9082_Conf]		
Description	References requiredModeGroupPrototype which shall be connected to the providedModeGroupPrototype.		
Multiplicity	1		
Type	Foreign reference to MODE-DECLARATION-GROUP-PROTOTYPE		
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

7.11 Configuration of Initialization

In order to support different interactions with the start up code of the ECU the RTE supports different initialization strategies for variables implementing `VariableDataPrototype`s. Basically the initialization can be done either by start-up code or by the `Rte_Start` function. Further on it is possible to avoid any initialization for data which has to be reset safe or is explicitly initialized by other SW, e.g. the NVRAM Blocks might be initialized by NVRAM Manager.

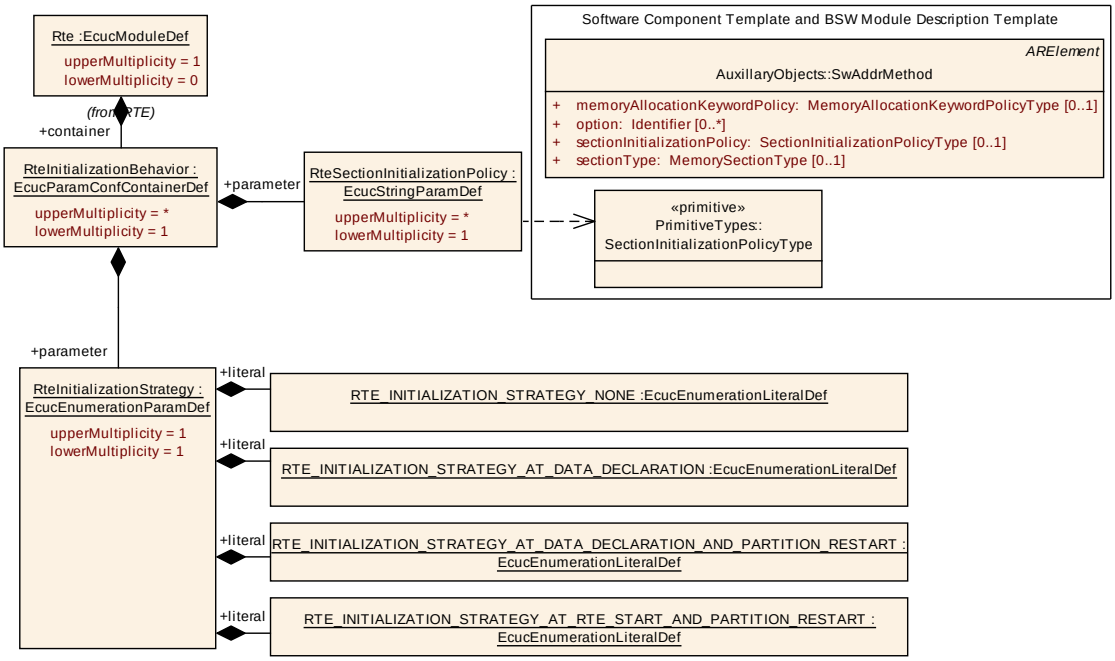


Figure 7.25: Configuration of initialization strategy

RteInitializationBehavior

SWS Item	[rte_sws_9087_Conf]
Container Name	RteInitializationBehavior
Description	Specifies the initialization strategy for variables allocated by RTE with the purpose to implement VariableDataPrototypes. The container defines a set of RteSectionInitializationPolicys and one RteInitializationStrategy which is applicable for this set.
Configuration Parameters	

Name	RteInitializationStrategy [rte_sws_9089_Conf]		
Description	Definition of the initialization strategy applicable for the SectionInitializationPolicys selected by RteSectionInitializationPolicy.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	RTE_INITIALIZATION_STRATEGY_AT_DATA_DECLARATION	Variables shall be initialized at its declaration to the value defined by the related initValue attribute.	
	RTE_INITIALIZATION_STRATEGY_AT_DATA_DECLARATION_AND_PARTITION_RESTART	Variables shall be initialized at its declaration to the value defined by the related initValue attribute and during execution of Rte_RestartPartition to the value defined by the related initValue attribute.	
	RTE_INITIALIZATION_STRATEGY_AT_RTE_START_AND_PARTITION_RESTART	Variables shall be initialized during execution of Rte_Start and Rte_RestartPartition to the value defined by the related initValue attribute.	
	RTE_INITIALIZATION_STRATEGY_NONE	Variables shall not be initialized at all.	
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	RteSectionInitializationPolicy [rte_sws_9088_Conf]		
Description	This parameter describes the SectionInitializationPolicys for which a particular RTE initialization strategy applies. The SectionInitializationPolicy describes the intended initialization of MemorySections. The following values are standardized in AUTOSAR Methodology: * "'NO-INIT"': No initialization and no clearing is performed. Such data elements must not be read before one has written a value into it. * "'INIT"': To be used for data that are initialized by every reset to the specified value (initValue). * "'POWER-ON-INIT"': To be used for data that are initialized by "Power On" to the specified value (initValue). Note: there might be several resets between power on resets. * "'CLEARED"': To be used for data that are initialized by every reset to zero. * "'POWER-ON-CLEARED"': To be used for data that are initialized by "Power On" to zero. Note: there might be several resets between power on resets.		
Multiplicity	1..*		
Type	EcucStringParamDef		
Default Value			
Regular Expression			
Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		
No Included Containers			

[rte_sws_7075]⌈ The RTE generator shall reject configurations where not all occurring `SectionInitializationPolicy` attribute values are configured to an `RteInitializationStrategy`. ⌋*(RTE00018)*

A Metamodel Restrictions

This chapter lists all the restrictions to the AUTOSAR meta-model this version of the AUTOSAR RTE specification document relies on. The RTE generator shall reject configuration where any of the specified restrictions are violated.

A.1 Restrictions concerning `WaitPoint`

1. **[rte_sws_1358]**⌈ An error shall be raised if `RunnableEntity` has `WaitPoint` connected to any of the following `RTEEvent`s:
   - `OperationInvokedEvent`
   - `SwcModeSwitchEvent`
   - `TimingEvent`
   - `BackgroundEvent`
   - `DataReceiveErrorEvent`
   - `ExternalTriggerOccurredEvent`
   - `InternalTriggerOccurredEvent`
   - `DataWriteCompletedEvent`

   The runnable can only be started with these events. ⌋*(RTE00092, RTE00018)*

   Rationale: For `OperationInvokedEvent`s, `SwcModeSwitchEvent`s, `TimingEvent`s, `BackgroundEvent`s `DataReceiveErrorEvent`, `ExternalTriggerOccurredEvent`, `InternalTriggerOccurredEvent`, and `DataWriteCompletedEvent` it suffices to allow the activation of a `RunnableEntity`.

2. **[rte_sws_7402]**⌈ The RTE generator shall reject a model where two (or more) different `RunnableEntity`s in the same internal behavior each have a `WaitPoint` referencing the same `DataReceivedEvent`, and the runnables are mapped to different tasks. ⌋*(RTE00092, RTE00018)*

   Rationale: In the same software components, the two runnables will attempt to read from the same queue, and only the one that accesses the queue first will actually receive the data.

A.2 Restrictions concerning RTEEvent

1. **[rte_sws_3526]**⌈ The RTE generator shall reject configurations in which a `RunnableEntity` is triggered by multiple `OperationInvokedEvent`s but vio-

lating the constraint [2000] *Compatibility of ClientServerOperations triggering the same RunnableEntity* as defined in document [2] ⌋*(RTE00072, RTE00018)*

Rationale: The signature of the `RunnableEntity` is dependent on its connected `RTEEvent`. Multiple `OperationInvokedEvent`s are only supported if all referred `Operation`s would result in the same `RunnableEntity` prototype for the server runnable (see 5.7.5.6).

2. **[rte_sws_3010]**⌈ One runnable entity shall only be resumed by one single RTEEvent on its WaitPoint. The RTE doesn't support the `WaitPoint` of one runnable entity connected to several RTEEvents. ⌋*(RTE00092, RTE00018)*

   Rationale: The `WaitPoint` of the runnable entity is caused by calling of the RTE API. One runnable entity can only call one RTE API at a time, and so it can only wait for one RTEEvent.

3. **[rte_sws_7007]**⌈ The RTE generator shall reject configurations where different execution instances of a runnable entity, which use implicit data access, are mapped to different Preemption Areas. ⌋*(RTE00018, RTE00128, RTE00129, RTE00133, RTE00142)*

   Rationale: Buffers used for implicit communication shall be consistent during the whole task execution. If it is guaranteed that one task does not preempt the other, direct accesses to the same copy buffer from different tasks are possible.

4. **[rte_sws_7403]**⌈ The RTE generator shall reject a model where in the same `SwcInternalBehavior` two (or more) different `DataReceivedEvent`s, that reference the same `VariableDataPrototype` with event semantics, trigger different runnable entities mapped to different tasks. ⌋*(RTE00072, RTE00018)*

   Rationale: In the same software components, the two runnables will attempt to read from the same queue, and only the one that accesses the queue first will actually receive the data.

A.3 Restrictions concerning queued implementation policy

1. **[rte_sws_3012]**⌈ Access with `VariableAccess`es in the `dataReadAccess` role is only allowed for `VariableDataPrototype`s whose `swImplPolicy` attribute is not set to queued. ⌋*(RTE00128, RTE00018)*

   Rationale: By access with `VariableAccess` in the `dataReadAccess` role always the last value of the `VariableDataPrototype` will be read in the runnable. There is no meaning to provide a queue of values for the `dataReadAccess` role.

2. **[rte_sws_3018]**⌈ RTE does not support receiving with `WaitPoint` for `VariableDataPrototype`s with their `swImplPolicy` attribute is not set to queued. ⌋*(RTE00109, RTE00092, RTE00018)*

Rationale: unqueued implementation policy indicates that the receiver shall not wait for the `VariableDataPrototype`.

3. All the `VariableAccesses` in the `dataSendPoint` role referring to one `VariableDataPrototype` through one PPortPrototype are considered to have the same behavior by sending and acknowledgment reception. All `DataSendCompletedEvents` that reference `VariableAccesses` in the `dataSendPoint` role referring to the same `VariableDataPrototype` are considered equivalent.

   Rationale: The API `Rte_Send/Rte_Write` is dependent on the port name and the `VariableDataPrototype` name, not on the `VariableAccesses`. For each combination of one `VariableDataPrototype` and one port only one API will be generated and implemented for sending or acknowledgement reception.

A.4 Restrictions concerning ServerCallPoint

1. **[rte_sws_3014]**⌈ All the ServerCallPoints referring to one `ClientServerOperation` through one RPortPrototype are considered to have the same behavior by calling service. The RTE generator shall reject configuration where this is violated. ⌋*(RTE00051, RTE00018)*

   Rationale: The API `Rte_Call` is dependent on the port name and the operation name, not on the ServerCallPoints. For each combination of one operation and one port only one API will be generated and implemented for calling a service. It is e.g. not possible to have different timeout values specified for different ServerCallPoints of the same `ClientServerOperation`. It is also not allowed to specify both, a synchronous and an asynchronous server call point for the same `ClientServerOperation` instance.

2. **[rte_sws_3605]**⌈ If several require ports of a software component are categorized by the same client/server interface, all invocations of the same operation of this client/server interface have to be either synchronous, or all invocations of the same operation have to be asynchronous. This restriction applies under the following conditions:

   - the usage of the indirect API is specified for at least one of the respective port prototypes **and/or**
   - the software component supports multiple instantiation, **and** the RTE generation shall be performed in compatibility mode.

   ⌋*(RTE00051, RTE00018)*

   Rationale: The signature of `Rte_Call` and the existence of `Rte_Result` depend on the kind of invocation.

3. **[rte_sws_7170]**⌈ An AsynchronousServerCallPoint shall be referenced by one AsynchronousServerCallResultPoints only. The RTE generator shall reject configuration where this is violated. ⌋*(RTE00051, RTE00018)*

   Rationale: The support of several AsynchronousServerCallResultPoints per `AsynchronousServerCallPoint` would potentially support multiple AsynchronousServerCallReturnsEvents as well as multiple `WaitPoint`s for the same `AsynchronousServerCallPoint`.

A.5 Restriction concerning multiple instantiation of software components

1. **[rte_sws_7101]**⌈ The RTE does not support configurations in which a *PortAPIOption* with *enableTakeAddress* = *TRUE* is defined by a software-component supporting multiple instantiation. ⌋*(RTE00018)*

   Rationale: The main focus of the feature is support for configuration of AUTOSAR Services which are limited to single instances.

A.6 Restrictions concerning runnable entity

1. **[rte_sws_3527]**⌈ The RTE does NOT support multiple Runnable Entities sharing the same entry point (symbol attribute of RunnableEntity). ⌋*(RTE00072, RTE00018)*

   Rationale: The handle to data shared by `VariableAccess`es in the `dataReadAccess` and `dataWriteAccess` roles has to be coded in the runnable code. An alternative would be an additional parameter to the runnable (a runnable handle) to provide this indirection information.

2. **[rte_sws_2733]**⌈ The RTE Generator shall reject a configuration where a runnable has the attribute `canBeInvokedConcurrently` set to true and the attribute `minimumStartInterval` set to greater zero. ⌋*(RTE00018)*

   Rationale: If a runnable should run concurrently (i.e., have several `ExecutableEntity execution-instances`), this implies that the minimum interval between the start of the runnables is zero. The configuration to be rejected is inconsistent.

A.7 Restrictions concerning runnables with dependencies on modes

1. Operations may not be disabled by a ModeDisablingDependency.

   [rte_sws_2706]⌈ RTE shall reject configurations that contain OperationInvokedEvents with a `ModeDisablingDependency`. ⌋*(RTE00143, RTE00018)*

   Rationale: It is a preferable implementation, if the server responds with an explicit application error, when the server operation is not supported in a mode. To implement the disabling of operations would require a high amount of book keeping even for internal client server communication to prevent that the unique request response mapping gets lost.

2. Only a category 1 runnable may be triggered by

   - a SwcModeSwitchEvent
   - an RteEvent with a mode disabling dependency

   [rte_sws_2500]⌈ The RTE generator shall reject configurations with category 2 runnables connected to `SwcModeSwitchEvent`s and `RTEEvent`s / `BswEvent`s with `ModeDisablingDependency`s if the mode machine instance is synchronous. The rejection may be reduced to a warning when the RTE generator is explicitly set to a non strict mode. ⌋*(RTE00143, RTE00213, RTE00018)*

   Rationale: The above runnables are executed or terminated on the transitions between different modes. To execute the mode switch withing finite time, also these runnables have to be executed within finite execution time.

3. All `OnEntry ExecutableEntitys`, `OnTransition ExecutableEntitys`, and `OnExit ExecutableEntitys` of the same `mode machine instance` should be mapped to the same task in case of synchronous mode switching procedure.

   [rte_sws_2662]⌈ The RTE generator shall reject configurations with OnEntry, OnTransition, or OnExit ExecutableEntity's of the same `mode machine instance` that are mapped to different tasks in case of synchronous mode switching procedure. ⌋*(RTE00143, RTE00213, RTE00018)*

   In case of asynchronous mode switching procedure, a mapping of all affected runnables to no task is also possible.

   Rationale: This restriction simplifies the implementation of the semantics of a synchronous mode switch.

4. To guarantee that all `mode disabling dependent ExecutableEntitys` of a `mode machine instance` have terminated before the start of the `OnExit`

ExecutableEntitys of the transition, the mode disabling dependent ExecutableEntitys should run with higher or equal priority.

[rte_sws_2663]⌈ The RTE generator shall reject configurations with mode disabling dependent ExecutableEntitys that are mapped to a task with lower priority than the task that contains the OnEntry ExecutableEntitys and OnExit ExecutableEntitys of that mode machine instance supporting a synchronous mode switching procedure. ⌋*(RTE00143, RTE00213, RTE00018)*

5. **[rte_sws_2664]**⌈ The RTE generator shall reject configurations of a task with

   - OnExit ExecutableEntitys mapped after OnEntry ExecutableEntitys or
   - OnTransition ExecutableEntitys mapped after OnEntry ExecutableEntitys or
   - OnExit ExecutableEntitys mapped after OnTransition ExecutableEntitys

   of the same mode machine instance supporting a synchronous mode switching procedure. ⌋*(RTE00143, RTE00213, RTE00018)*

   Rationale: This restriction simplifies the implementation of the semantics of a synchronous mode switch.

6. **[rte_sws_7157]**⌈ The RTE generator shall reject configurations with

   - OnExit ExecutableEntitys mapped after OnEntry ExecutableEntitys or
   - OnTransition ExecutableEntitys mapped after OnEntry ExecutableEntitys or
   - OnExit ExecutableEntitys mapped after OnTransition ExecutableEntitys

   of the same software component or Basic Software Module for a mode machine instance supporting an asynchronous mode switching procedure. ⌋*(RTE00143, RTE00213, RTE00018)*

   Rationale: This restriction simplifies the implementation of the semantics of an asynchronous mode switch.

7. If a mode is used to trigger a runnable for entering or leaving the mode, but this runnable has a ModeDisablingDependency on the same mode, the ModeDisablingDependency inhibits the activation of the runnable on the transition (see section 4.4.4).

   To prevent such a misleading configuration, it is strongly recommended not to configure a ModeDisablingDependency for an OnEntry ExecutableEntity or OnExit ExecutableEntity, using the same mode.

A.8 Restriction concerning SwcInternalBehavior

1. **[rte_sws_7686]**⌈ The RTE Generator shall reject configurations where an `AtomicSwComponentType` does not contain a `SwcInternalBehavior`. ⌋*(RTE00018)*

A.9 Restrictions concerning Initial Value

1. **[rte_sws_4525]**⌈ All VariableDataPrototype that are connected to the same sender, or connected to the same receiver, or mapped to the same SystemSignal, must have identical init values. ⌋*(RTE00108, RTE00018)*

   Rationale: In the meta model init values are specified in the data receiver or sender com spec. Since a separate data receiver com spec exists for each port that categorizes a specific interface, it would be (theoretically) possible to define a different init value for a certain data element in each port. But COM allows only one init value per signal.

2. **[rte_sws_7642]**⌈ When the *external configuration switch* `strictInitialValuesCheck` is enabled, the RTE Generator shall reject configurations where a `SwAddrMethod` has a `sectionInitializationPolicy` set to `init` but no `initValue`s are specified on the sender or receiver side. ⌋*(RTE00068, RTE00108, RTE00018)*

   Rationale: The `initValue` is used to guarantee that the RTE won't deliver undefined values.

3. **[rte_sws_7681]**⌈ If strict checking of initial values is enabled (see rte_sws_7680), the RTE Generator shall reject configurations where a `ParameterDataPrototype` has no `initValue`s. ⌋*(RTE00108, RTE00018)*

   Rationale: This allows to provide the values with a calibration without any involvements from the RTE Generator, and still permits to enable a stricter check on projects where it is required.

A.10 Restriction concerning PerInstanceMemory

1. **[rte_sws_3790]**⌈ The `<typeDefinition>` attribute of a PerInstanceMemory is not allowed to contain a function pointer. ⌋*(RTE00013, RTE00077)*

   Rationale: Using the type definition `typedef <typedefinition> <typename>` does not work for function pointers.

2. **[rte_sws_7045]**⌈ The RTE generator shall reject configurations where the `type` attribute of a `'C' typed PerInstanceMemory` is equal to the name of a `Im-`

plementationDataType contained in the input configuration. ⌋*(RTE00013, RTE00077)*

Rationale: This would lead to equally named C type definitions.

A.11 Restrictions concerning unconnected r-port

1. **[rte_sws_3019]**⌈ If strict checking has been enabled (see rte_sws_5099) there shall not be unconnected r-port. The RTE generator shall in this case reject the configuration with unconnected r-port. ⌋*(RTE00139, RTE00018)*

   Rationale: Unconnected r-port is considered as wrong configuration of the system.

2. **[rte_sws_2750]**⌈ The RTE Generator shall reject configurations where an r-port typed with a ParameterInterface is not connected and an initValue of a ParameterRequireComSpec is not provided for each ParameterDataPrototypes of this ParameterInterface. ⌋*(RTE00139, RTE00159, RTE00018)*

A.12 Restrictions regarding communication of mode switch notifications

1. **[rte_sws_2670]**⌈RTE shall not support connections with multiple senders (n:1 communication) of mode switch notifications connected to the same receiver. The RTE generator shall reject configurations with multiple senders of mode switch notifications connected to the same receiver. ⌋*(RTE00131, RTE00018)*

   Rationale: No use case is known to justify the required complexity.

2. **[rte_sws_2724]**⌈RTE shall reject configurations where one ModeDeclarationGroupPrototype of a provide port is connected to ModeDeclarationGroupPrototypes of require ports from more than one partition. ⌋*(RTE00131, RTE00018)*

   RTE does not support a configuration in which the mode users of one mode machine instance are distributed over several partitions.

   Note that the mode manager does not have to be in the same partition as the mode users.

3. For each ModeDeclarationGroup, used in the SW-C's ports, RTE needs a unique mapping to an ImplementationDataType.

   [rte_sws_2738]⌈ RTE shall reject a configuration, in which there is not exactly one ModeRequestTypeMap referencing the ModeDeclarationGroup used

in a ModeDeclarationGroupPrototype of the SW-C's ports. ⌋*(RTE00144, RTE00018)*

A.13 Restrictions regarding Measurement and Calibration

1. **[rte_sws_3951]**⌈ RTE does not support measurement of queued communication. ⌋*(RTE00153, RTE00018)*

   Rationale: Measurement of queued communication is not supported yet. Reasons are:

   - A queue can be empty. What's to measure then? Data interpretation is ambiguous.
   - Which of the queue entries the measurement data has to be taken from (first pending entry, last entry, an intermediate one, mean value, min. or max. value)? Needs might differ out of user view? Data interpretation is ambiguous.
   - Compared e.g. to sender-receiver last-is-best approach only inefficient solutions are possible because implementation of queues entails storage of information dynamically at different memory locations. So always additional copies are required.

2. **[rte_sws_3970]**⌈ The RTE generator shall reject configurations containing require ports attached to ParameterSwComponentTypes. ⌋*(RTE00154, RTE00156, RTE00018)*

   Rationale: Require ports on ParameterSwComponentTypes don't make sense. ParameterSwComponentTypes only have to provide calibration parameters to other SwComponentTypes.

A.14 Restriction concerning ExclusiveAreaImplMechanism

1. **[rte_sws_ext_3811]** If an exclusive area's configuration value for *ExclusiveAreaImplMechanism* is *InterruptBlocking* or *OsResource*, no runnable entity shall contain any WaitPoint inside this exclusive area.

   Please note that a wait point can either be a modelling WaitPoint e. g. a WaitPoint in the SW-C description caused by the usage of a blocking API (e. g. Rte_Receive) or an implementation wait point caused by a special implementation to fulfill the requirements of the ECU configuration, e. g. the runnable-to-task mapping.

   Rationale: The operating system has the limitation that a WaitEvent call is not allowed with disabled interrupts. Therefore the implementation mechanism *InterruptBlocking* cannot be used if the exclusive area contains a waitpoint.

Further the operating system has the limitation that an OS waitpoint cannot be entered with occupied OS Resources. This implies that the implementation mechanism *OsResource* cannot be used if the exclusive area contains a waitpoint.

A.15 Restrictions concerning `AtomicSwComponentType`s

1. **[rte_sws_7190]**⌈ The RTE generator shall reject configurations where multiple `SwComponentType`s have the same `component type symbol` regardless of the `ARPackage` hierarchy. ⌋*(RTE00018)*

   Rational: This is required to generated unique names for the *Application Header Files* and component data structures.

2. **[rte_sws_7191]**⌈ The RTE generator shall reject configurations where a `SwComponentType` has `PortPrototype`s typed by different `PortInterface`s with equal short name but conflicting `ApplicationError`s. `ApplicationError`s are conflicting if `ApplicationError`s with same name do have different `errorCode`s. ⌋*(RTE00018)*

   Rational: This is required to generated unique symbolic names for `ApplicationError`s. (see also rte_sws_2576)

A.16 Restriction concerning the `enableUpdate` attribute of `NonqueuedReceiverComSpec`s

1. **[rte_sws_7654]**⌈ The RTE Generator shall reject configurations where a `VariableDataPrototype` is referenced by an `NonqueuedReceiverComSpec` with the `enableUpdate` attribute enabled, when this `VariableDataPrototype` is referenced by a `VariableAccess` in the `dataReadAccess` role. ⌋*(RTE00179, RTE00018)*

   Rational: the update flag is restricted to explicit communication currently.

A.17 Restrictions concerning the large and dynamic data type

1. **[rte_sws_7810]**⌈ The RTE shall reject the configuration if a `dataElement` with an `ImplementationDataType` with `subElement`s with `arraySizeSemantics` equal to `variableSize` resolves to another type than uint8[n]. ⌋*(RTE00018)*

   Rationale: COM limits the dynamic signals to the `ComSignalType` UINT_8DYN (see the requirement COM569). COM doesn't support dynamic signals included into signal groups. See more explanations in chapter 4.3.1.14.

2. **[rte_sws_7811]**⌈ The RTE shall reject the configuration if a `dataElement` mapped to a PDU with `ComIPduType` equal to `TP` has a `swImplPolicy` different from `queued`. ⌋*(RTE00018)*

   Rationale: Otherwise COM might return `COM_BUSY`. See more explanations in chapter 4.3.1.15.

3. **[rte_sws_7812]**⌈ The RTE shall reject the configuration if a `dataElement` with an `ImplementationDataType` with `subElement`s with `arraySizeSemantics` equal to `variableSize` has a `swImplPolicy` different from `queued`. ⌋*(RTE00018)*

   Rationale: Otherwise COM might return `COM_BUSY`. See more explanations in chapter 4.3.1.15.

A.18 Restriction concerning REFERENCE types

1. **[rte_sws_7670]**⌈ The RTE shall reject the configuration if an `ImplementationDataType` with category `DATA_REFERENCE` is used in a `PortInterface` and neither sender nor receiver component is a service, complex device driver or ECU abstraction. ⌋*(RTE00018)*

   Rationale: Only for AUTOSAR services, complex device drivers or ECU abstraction, the use of references is allowed to prevent the misuse of references for communication via the referenced memory (intra-partition scope). For example, such a misuse could occur with application software components communicating together and mapped to different partitions or ECUs.

B External Requirements

[rte_sws_ext_7521] The RunnableEntitys or BswSchedulableEntitys worst case execution time shall be less than the GCD of all BswSchedulableEntitys and RunnableEntitys period and offset in activation offset context for RunnableEntitys and BswSchedulableEntitys.

[rte_sws_ext_7816] Category 1 interrupts shall not access the RTE.

[rte_sws_ext_7351] The NVM block associated to the NvBlockDescriptors of a NvBlockSwComponentType shall be configured with the NvmBlockUseSyncMechanism feature enabled, and the NvmWriteRamBlockToNvm and NvmReadRamBlockFromNvm parameters set to the Rte_GetMirror and Rte_SetMirror API of the NvBlockDescriptor.

[rte_sws_ext_2542] Whenever any *Runnable Entity* or *Basic Software Schedulable Entity* is running, there shall always be exactly one mode or one mode transition active of each ModeDeclarationGroupPrototype.

[rte_sws_ext_7565] Only one of two synchronized *ModeDeclarationGroupPrototype*s shall mutual exclusively be referenced by *ModeSwitchPoint*(s) or *managedModeGroup* association(s).

[rte_sws_ext_7547] A *releasedTrigger Trigger* shall not be referenced by both a *issuedTrigger* and a *BswTriggerDirectImplementation*.

[rte_sws_ext_7040] The same Trigger in a *Trigger Sink* must not be connected to multiple *Trigger Source*s.

[rte_sws_ext_7550] A synchronized *Trigger* shall only be referenced by either *ExternalTriggeringPoint*s, *issuedTrigger*s or *BswTriggerDirectImplementation*s.

[rte_sws_ext_1190] The arraySize defining number of elements in one dimension of an *Array Implementation Data Type* shall be an integer that is ≥ 1 for each dimension.

[rte_sws_ext_1192] A structure shall include at least one element defined by a ImplementationDataTypeElement.

[rte_sws_ext_7147] A *Union Implementation Data Type* shall include at least two elements defined by ImplementationDataTypeElements.

[rte_sws_ext_7818] The Rte_Write APIs may only be used by the runnable that contains the corresponding VariableAccess in the dataSendPoint role

[rte_sws_ext_7819] The Rte_Send APIs may only be used by the runnable

that contains the corresponding `VariableAccess` in the `dataSendPoint` role

[rte_sws_ext_2681] The `Rte_Switch` API may only be used by the runnable that contains the corresponding *ModeSwitchPoint*

[rte_sws_ext_2682] The `Rte_Invalidate` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataSendPoint` role

[rte_sws_ext_2687] A blocking `Rte_Feedback` API may only be used by the runnable that contains the corresponding `WaitPoint`

[rte_sws_ext_2726] A blocking `Rte_SwitchAck` API may only be used by the runnable that contains the corresponding `WaitPoint`

[rte_sws_ext_2683] The `Rte_Read` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByArgument` role

[rte_sws_ext_7397] The `Rte_DRead` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByValue` role

[rte_sws_ext_2684] The `Rte_Receive` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByArgument` role

[rte_sws_ext_2685] The `Rte_Call` API may only be used by the runnable that contains the corresponding ServerCallPoint

[rte_sws_ext_2686] The blocking `Rte_Result` API may only be used by the runnable that contains the corresponding `WaitPoint`

[rte_sws_ext_7679] The reference returned by `Rte_IWriteRef` shall not be used by the runnables for reading the value previously written.

[rte_sws_ext_2601] The `Rte_IStatus` API shall only be used by a RunnableEntity that either has a `VariableAccess` in the `dataReadAccess` role referring to the `VariableDataPrototype` or is triggered by a DataReceiveErrorEvent referring to the `VariableDataPrototype`.

[rte_sws_ext_7171] The `Rte_Enter` and `Rte_Exit` API may only be used by *Runnable Entities* that contain a corresponding *canEnterExclusiveArea* association

[rte_sws_ext_7172] The `Rte_Enter` and `Rte_Exit` API may only be called nested if different exclusive areas are invoked; in this case exclusive areas shall exited in the reverse order they were entered.

[rte_sws_ext_7568] The `Rte_Mode` API may only be used by the runnable that contains the corresponding *ModeAccessPoint*

[rte_sws_ext_8502] The `Rte_Mode` API may only be used by the runnable that contains the corresponding *ModeAccessPoint*

[rte_sws_ext_7202] The `Rte_Trigger` API may only be used by the runnable that contains the corresponding `ExternalTriggeringPoint`.

[rte_sws_ext_7205] The `Rte_IrTrigger` API may only be used by the runnable that contains the corresponding `InternalTriggeringPoint`.

[rte_sws_ext_7603] The `Rte_IsUpdated` API may only be used by the runnable that contains the corresponding `VariableAccess` in the `dataReceivePointByArgument` or `dataReceivePointByValue` role.

[rte_sws_ext_2704] Only the least significant six bit of the return value of a server runnable shall be used by the application to indicate an error. The upper two bit shall be zero.

[rte_sws_ext_2582] `Rte_Start` shall be called only once by the EcuStateManager from trusted OS context on a core after the basic software modules required by RTE are initialized.

[rte_sws_ext_7577] The `Rte_Start` API may only be used after the *Basic Software Scheduler* is initialized (after termination of the `SchM_Init`).

[rte_sws_ext_2714] The `Rte_Start` API shall be called on every core that hosts AUTOSAR software-components of the ECU.

[rte_sws_ext_2583] `Rte_Stop` shall be called by the EcuStateManager before the basic software modules required by RTE are shut down.

[rte_sws_ext_7332] `Rte_PartitionTerminated` shall be called only once by the ProtectionHook.

[rte_sws_ext_7618] `Rte_PartitionRestarting` shall be called only once by the ProtectionHook.

[rte_sws_ext_7337] `Rte_RestartPartition` shall be called only in the context of the RestartTask of the given partition.

[rte_sws_ext_7512] Each BSW module implementation shall include its *Module Interlink Header File* if it uses *Basic Software Scheduler* API or if it implements `BswSchedulableEntity`s.

[rte_sws_ext_7285] The SchM_Enter and SchM_Exit API may only be used by BswModuleEntitys that contain a corresponding canEnterExclusiveArea association

[rte_sws_ext_7529] The SchM_Enter and SchM_Exit API may only be called nested if different exclusive areas are invoked; in this case exclusive areas shall exited in the reverse order they were entered.

[rte_sws_ext_7189] The SchM_Exit API may only be used by BswModuleEntitys that contain a corresponding canEnterExclusiveArea association

[rte_sws_ext_7257] The SchM_Switch API may only be used by BswModuleEntitys that contain a corresponding managedModeGroup association

[rte_sws_ext_7587] The SchM_Mode API may only be used by BswModuleEntitys that contain a corresponding *managedModeGroup* association or accessedModeGroup association

[rte_sws_ext_8508] The SchM_Mode API may only be used by BswModuleEntitys that contain a corresponding *managedModeGroup* association or accessedModeGroup association

[rte_sws_ext_7567] The SchM_SwitchAck API may only be used by BswModuleEntitys that contain a corresponding managedModeGroup association

[rte_sws_ext_7265] The SchM_Trigger API may only be used by the BswModuleEntity that contains the corresponding issuedTrigger association.

[rte_sws_ext_7268] The SchM_ActMainFunction API may only be used by the BswModuleEntity that contains the corresponding activationPoint association.

[rte_sws_ext_7287] The *Basic Software Scheduler* requires that the BswModuleEntry has no service arguments and no return value.

[rte_sws_ext_7272] SchM_Init shall be called only once by the *EcuStateManager* on each core after the basic software modules required by the *Basic Software Scheduler* part of the RTE are initialized.

[rte_sws_ext_7576] The SchM_Deinit API may only be used after the RTE finalized (after termination of the Rte_Stop)

[rte_sws_ext_7276] SchM_Deinit shall be called by the *EcuStateManager* before the basic software modules required by *Basic Software Scheduler* part are shut down.

[rte_sws_ext_7598] The references `RteSwcTriggerSourceRef` has to be consistent with the `RteSoftwareComponentInstanceRef`. This means the referenced `Trigger` / `InternalTriggeringPoint` has to belong to the `AtomicSwComponentType` which is referenced by the related `SwComponentPrototype`.

[rte_sws_ext_7597] The references `RteBswTriggerSourceRef` has to be consistent with the `RteBswImplementationRef`. This means the referenced `Trigger` / `BswInternalTriggeringPoint` has to belong to the `BswModuleDescription` which is referenced by the related `BswImplementation`.

[rte_sws_ext_3811] If an exclusive area's configuration value for *ExclusiveAreaImplMechanism* is *InterruptBlocking* or *OsResource*, no runnable entity shall contain any `WaitPoint` inside this exclusive area.

C MISRA C Compliance

In general, all RTE code, whether generated or not, shall conform to the HIS subset of the MISRA C standard rte_sws_1168 [25]. This chapter lists all the MISRA C rules of the HIS subset that may be violated by the generated RTE.

The MISRA C standard was defined with having mainly hand-written code in mind. Part of the MISRA C rules only apply to hand-written code, they do not make much sense in the context of automatic code generation. Additonally, there are some rules that are violated because of technical reasons, mainly to reduce RTE overhead.

The rules listed in this chapter are expected to be violated by RTE code. Violations to the rules listed here do not need to be documented as non-compliant to MISRA C in the generated code itself.

MISRA rule	11
Description	Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
Violations	The defined RTE naming convention may result in identifiers with more than 31 characters. The compliance to this rule is under user's control.

MISRA rule	23
Description	All declarations at file scope should be static where possible.
Violations	E.g. for the purpose of monitoring during calibration or debugging it may be necessary to use non-static declarations at file scope.

MISRA rule	42
Description	The comma operator shall not be used, except in the control expression of a *for* loop.
Violations	Function-like macros may have to use the comma operator. Function-like macros are required for efficiency reasons [BSW00330].

MISRA rule	45
Description	Type casting from any type to or from pointers shall not be used.
Violations	Casting to/from pointer type may be needed for the interface with COM. Casting from a pointer to a `Data Element with Status` to a pointer to a `Data Element without Status`.

MISRA rule	54

Description	A null statement shall only occur on a line by itself, and shall not have any other text on the same line.
Violations	In an optimized RTE, API calls may result in a null statement. Therefore the compliance to this rule cannot be guaranteed.

D Changes History

D.1 Changes in Rel. 4.0 Rev. 2 compared to Rel. 4.0 Rev. 1

D.1.1 Deleted SWS Items

The following SWS Items were removed in Rel. 4.0 Rev. 2: rte_sws_1254, rte_sws_3552, rte_sws_3557, rte_sws_3559, rte_sws_3563, rte_sws_3564, rte_sws_3568, rte_sws_3588, rte_sws_3593, rte_sws_3743, rte_sws_5512.

D.1.2 Changed SWS Items

The following SWS Items were changed in Rel. 4.0 Rev. 2: rte_sws_1086, rte_sws_1111, rte_sws_1113, rte_sws_1114, rte_sws_1118, rte_sws_1156, rte_sws_1355, rte_sws_2517, rte_sws_2527, rte_sws_2528, rte_sws_2613, rte_sws_2615, rte_sws_2679, rte_sws_2728, rte_sws_2730, rte_sws_2747, rte_sws_2752, rte_sws_2753, rte_sws_3001, rte_sws_3560, rte_sws_3562, rte_sws_3567, rte_sws_3598, rte_sws_3599, rte_sws_3774, rte_sws_3827, rte_sws_3837, rte_sws_3930, rte_sws_3953, rte_sws_3954, rte_sws_3955, rte_sws_3956, rte_sws_3957, rte_sws_5021, rte_sws_5156, rte_sws_5506, rte_sws_5509, rte_sws_6010, rte_sws_6633, rte_sws_7020, rte_sws_7021, rte_sws_7041, rte_sws_7184, rte_sws_7187, rte_sws_7195, rte_sws_7262, rte_sws_7280, rte_sws_7282, rte_sws_7293, rte_sws_7294, rte_sws_7375, rte_sws_7376, rte_sws_7409, rte_sws_7586, rte_sws_7589, rte_sws_7632, rte_sws_7636, rte_sws_7637, rte_sws_7667, rte_sws_7680, rte_sws_7683, rte_sws_ext_3811.

D.1.3 Added SWS Items

The following SWS Items were added in Rel. 4.0 Rev. 2: rte_sws_2761, rte_sws_3850, rte_sws_3851, rte_sws_3852, rte_sws_3853, rte_sws_7045, rte_sws_7046, rte_sws_7047, rte_sws_7048, rte_sws_7049, rte_sws_7050, rte_sws_7051, rte_sws_7052, rte_sws_7053, rte_sws_7054, rte_sws_7055, rte_sws_7056, rte_sws_7057, rte_sws_7058, rte_sws_7059, rte_sws_7060, rte_sws_7061, rte_sws_7062, rte_sws_7063, rte_sws_7064, rte_sws_7065, rte_sws_7066, rte_sws_7067, rte_sws_7068, rte_sws_7069, rte_sws_7070, rte_sws_7071, rte_sws_7072, rte_sws_7073, rte_sws_7074, rte_sws_7075, rte_sws_7076, rte_sws_7077, rte_sws_7078, rte_sws_7079, rte_sws_7080, rte_sws_7081, rte_sws_8000, rte_sws_8001, rte_sws_8002, rte_sws_8300, rte_sws_8301, rte_sws_8302.

D.2 Changes in Rel. 4.0 Rev. 3 compared to Rel. 4.0 Rev. 2

D.2.1 Deleted SWS Items

The following SWS Items were removed in Rel. 4.0 Rev. 3: rte_sws_3838, rte_sws_3844, rte_sws_3850, rte_sws_5171, rte_sws_7106, rte_sws_7108, rte_sws_7164, rte_sws_7165, rte_sws_7168, rte_sws_7176, rte_sws_7674.

D.2.2 Changed SWS Items

The following SWS Items were changed in Rel. 4.0 Rev. 3: rte_sws_1018, rte_sws_1019, rte_sws_1020, rte_sws_1156, rte_sws_1171, rte_sws_1238, rte_sws_1239, rte_sws_1248, rte_sws_1249, rte_sws_1300, rte_sws_2500, rte_sws_2568, rte_sws_2576, rte_sws_2627, rte_sws_2628, rte_sws_2629, rte_sws_2631, rte_sws_2648, rte_sws_2659, rte_sws_2662, rte_sws_2664, rte_sws_2675, rte_sws_2732, rte_sws_3526, rte_sws_3714, rte_sws_3731, rte_sws_3782, rte_sws_3793, rte_sws_3809, rte_sws_3810, rte_sws_3813, rte_sws_3827, rte_sws_3828, rte_sws_3829, rte_sws_3831, rte_sws_3832, rte_sws_3833, rte_sws_3837, rte_sws_3839, rte_sws_3840, rte_sws_3841, rte_sws_3842, rte_sws_3843, rte_sws_3845, rte_sws_3846, rte_sws_3847, rte_sws_3848, rte_sws_3849, rte_sws_3851, rte_sws_3907, rte_sws_3949, rte_sws_4526, rte_sws_5051, rte_sws_5052, rte_sws_5059, rte_sws_5062, rte_sws_5078, rte_sws_5127, rte_sws_5128, rte_sws_6513, rte_sws_6515, rte_sws_6518, rte_sws_6519, rte_sws_6520, rte_sws_6530, rte_sws_6532, rte_sws_6535, rte_sws_6536, rte_sws_7022, rte_sws_7030, rte_sws_7036, rte_sws_7037, rte_sws_7038, rte_sws_7047, rte_sws_7048, rte_sws_7069, rte_sws_7104, rte_sws_7109, rte_sws_7110, rte_sws_7111, rte_sws_7113, rte_sws_7114, rte_sws_7116, rte_sws_7133, rte_sws_7136, rte_sws_7144, rte_sws_7148, rte_sws_7149, rte_sws_7157, rte_sws_7162, rte_sws_7163, rte_sws_7166, rte_sws_7175, rte_sws_7182, rte_sws_7185, rte_sws_7190, rte_sws_7194, rte_sws_7195, rte_sws_7200, rte_sws_7203, rte_sws_7214, rte_sws_7224, rte_sws_7250, rte_sws_7253, rte_sws_7255, rte_sws_7260, rte_sws_7261, rte_sws_7263, rte_sws_7266, rte_sws_7282, rte_sws_7292, rte_sws_7293, rte_sws_7294, rte_sws_7295, rte_sws_7310, rte_sws_7315, rte_sws_7381, rte_sws_7382, rte_sws_7383, rte_sws_7501, rte_sws_7503, rte_sws_7504, rte_sws_7543, rte_sws_7544, rte_sws_7552, rte_sws_7554, rte_sws_7555, rte_sws_7556, rte_sws_7670, rte_sws_7682, rte_sws_8300.

D.2.3 Added SWS Items

The following SWS Items were added in Rel. 4.0 Rev. 3: rte_sws_3854, rte_sws_3855, rte_sws_3856, rte_sws_3857, rte_sws_3858, rte_sws_3859, rte_sws_3860, rte_sws_3861, rte_sws_6700, rte_sws_6701, rte_sws_6702, rte_sws_6703, rte_sws_6704, rte_sws_6705, rte_sws_6706, rte_sws_6707, rte_sws_6708,

rte_sws_6709, rte_sws_6710, rte_sws_6711, rte_sws_6712, rte_sws_6713, rte_sws_6714, rte_sws_6715, rte_sws_6716, rte_sws_6717, rte_sws_6718, rte_sws_6719, rte_sws_6720, rte_sws_6721, rte_sws_6722, rte_sws_6723, rte_sws_6724, rte_sws_6725, rte_sws_6726, rte_sws_7082, rte_sws_7083, rte_sws_7084, rte_sws_7085, rte_sws_7086, rte_sws_7087, rte_sws_7088, rte_sws_7089, rte_sws_7090, rte_sws_7091, rte_sws_7092, rte_sws_7093, rte_sws_7094, rte_sws_7095, rte_sws_7096, rte_sws_7097, rte_sws_7099, rte_sws_7593, rte_sws_7594, rte_sws_7595, rte_sws_7596, rte_sws_7692, rte_sws_7693, rte_sws_7694, rte_sws_7920, rte_sws_7921, rte_sws_7922, rte_sws_7923, rte_sws_7924, rte_sws_8004, rte_sws_8005, rte_sws_8007, rte_sws_8008, rte_sws_8009, rte_sws_8016, rte_sws_8017, rte_sws_8018, rte_sws_8020, rte_sws_8021, rte_sws_8022, rte_sws_8023, rte_sws_8024, rte_sws_8025, rte_sws_8026, rte_sws_8027, rte_sws_8028, rte_sws_8029, rte_sws_8030, rte_sws_8031, rte_sws_8032, rte_sws_8033, rte_sws_8034, rte_sws_8035, rte_sws_8036, rte_sws_8037, rte_sws_8038, rte_sws_8039, rte_sws_8040, rte_sws_8041, rte_sws_8042, rte_sws_8043, rte_sws_8044, rte_sws_8045, rte_sws_8303, rte_sws_8304, rte_sws_8305, rte_sws_8306, rte_sws_8307, rte_sws_8308, rte_sws_8400, rte_sws_8401, rte_sws_8402, rte_sws_8403, rte_sws_8404, rte_sws_8500, rte_sws_8501, rte_sws_8503, rte_sws_8504, rte_sws_8505, rte_sws_8506, rte_sws_8507, rte_sws_8509, rte_sws_8510, rte_sws_ext_7597, rte_sws_ext_7598, rte_sws_ext_8502, rte_sws_ext_8508.