VECTOR

MICROSAR Classic RTE

Technical Reference

Version 4.36.0

Author	PES3.4.1
Status	Released

Document Information

Reference Documents

No.	Title	Version
[1]	Specification of RTE	R20-11
[2]	Virtual Function Bus	R20-11
[3]	Specification of Communication	R20-11
[4]	Specification of Operating System	R20-11
[5]	Specification of NVRAM Manager	R20-11
[6]	Specification of ECU State Manager	R20-11
[7]	Specification of Standard Types	R20-11
[8]	Specification of Platform Types	R20-11
[9]	Specification of Compiler Abstraction	R20-11
[10]	Specification of Memory Mapping	R20-11
[11]	Software Component Template	R20-11
[12]	System Template	R20-11
[13]	Specification of ECU Configuration	R20-11
[14]	Glossary	R20-11
[15]	Specification of BSW Module Description Template	R20-11
[16]	Specification of Module XCP	R20-11
[17]	Specification of Default Error Tracer	R20-11
[18]	Specification of Module Efficient COM for Large Data	R20-11
[19]	Specification of SOME/IP Transformer	R20-11
[20]	Specification of COM Based Transformer	R20-11
[21]	Specification of Module E2E Transformer	R20-11
[22]	Vector DaVinci Configurator Online help	
[23]	Vector DaVinci Developer Online help	
[24]	AUTOSAR Calibration User Guide	1.0

Table 1-1 Reference documents

Scope of the Document

This document describes the MICROSAR Classic RTE. It assumes that the reader is familiar with the AUTOSAR architecture, especially the software component (SWC) design methodology and the AUTOSAR RTE specification. It also assumes basic knowledge of some basic software (BSW) modules like AUTOSAR Os, Com, LdCom, Transformer, NVM and EcuM. The description of those components is not part of this documentation. The related documents are listed in Table 1-1.

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

Contents

Illustrations

Tables

1 Introduction

This document describes the MICROSAR Classic RTE generation process, the RTE configuration with DaVinci Configurator and the RTE API.

The MICROSAR Classic RTE generator supports RTE and contract phase generation. Additionally, application template code can be generated for software components and for VFB trace hooks.

Chapter 2 gives an introduction to the MICROSAR Classic RTE. This brief introduction to the AUTOSAR RTE can and will not replace an in-depth study of the overall AUTOSAR methodology and in particular the AUTOSAR RTE specification, which provides detailed information on the concepts of the RTE.

In addition, chapter 2 describes deviations, extensions and limitations of the MICROSAR Classic RTE compared to the AUTOSAR standard.

The RTE generation process including the command line parameters of the MICROSAR Classic RTE generator is described in chapter 3. This chapter also gives hints for the integration of the generated RTE code into an ECU project. In addition, it describes the memory mapping and compiler abstraction related to the RTE and finally, chapter 3.6 describes the memory protection support of the RTE including hints for the integration with the OS.

The RTE API description in chapter 4 covers the API functionality implemented in the MICROSAR Classic RTE.

The description of the RTE configuration in chapter 4.17 covers the task mapping, memory mapping and the code generation settings in DaVinci Configurator. A more detailed description of the configuration tool including the configuration of AUTOSAR software components and compositions and their integration in an ECU project can be found in the online help of the DaVinci Configurator [22].

Supported Configuration Variants:	pre-compile	
Vendor ID:	RTE_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	RTE_MODULE_ID	2 decimal

1.1 Architecture Overview

The RTE is the realization of the interfaces of the AUTOSAR Virtual Function Bus (VFB) for a particular ECU. The RTE provides both standardized communication interfaces for AUTOSAR software components realized by generated RTE APIs and it also provides a runtime environment for the component code – the runnable entities. The RTE triggers the execution of runnable entities and provides the infrastructure services that enable communication between AUTOSAR SWCs. It is acting as a broker for accessing basic software modules including the OS and communication services.

The following figure shows where the MICROSAR Classic RTE is located in the AUTOSAR architecture.

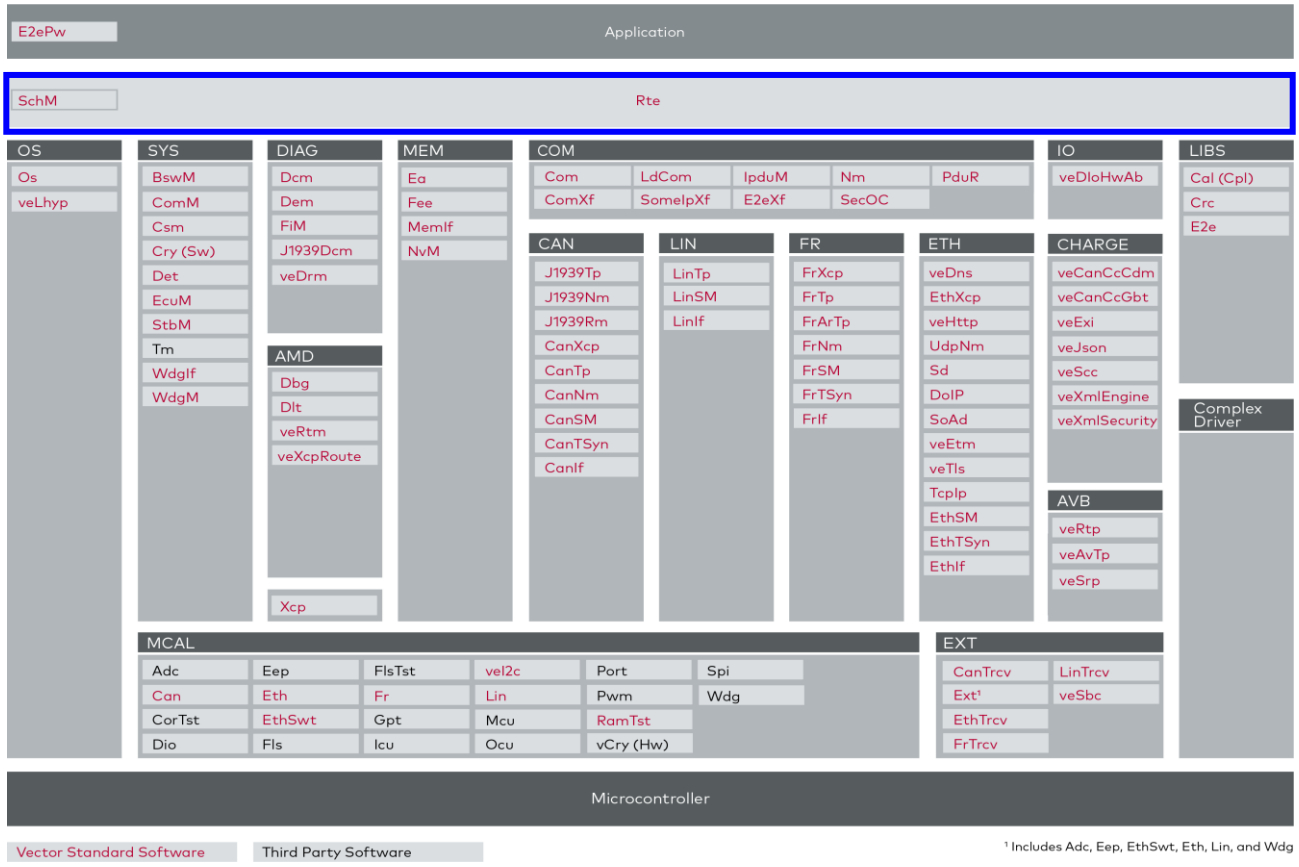


Figure 1-1 AUTOSAR architecture

RTE functionality overview:

- Execution of runnable entities of SWCs on different trigger conditions
- Communication mechanisms between SWCs (Sender/Receiver and Client/Server)
- Mode Management
- Inter-Runnable communication and exclusive area handling
- Per-Instance Memory and calibration parameter handling

- Multiple instantiation of SWCs
- OS task body and COM / LDCOM callback generation
- Automatic configuration of parts of the OS, NVM and COM / LDCOM dependent of the needs of the RTE
- Assignment of SWCs to different memory partitions/cores

SchM functionality overview:

- Execution of cyclic triggered schedulable entities (BSW main functions)
- Exclusive area handling for BSW modules
- OS task body generation

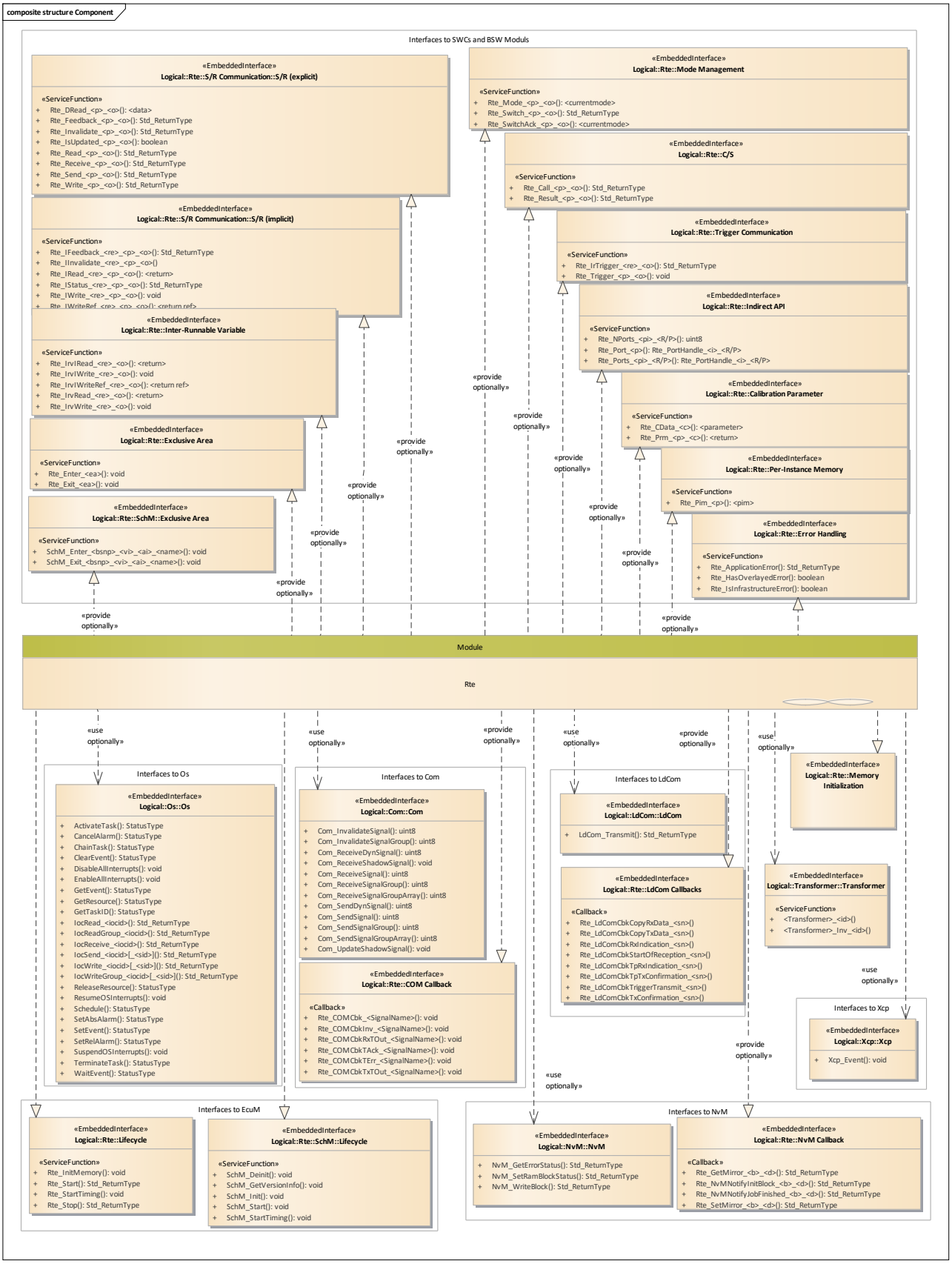


Figure 1-2 Interfaces to adjacent modules of the RTE

2 Functional Description

2.1 Features

The features listed in the following tables cover the complete functionality specified for the RTE.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

- Table 2-1 Supported AUTOSAR standard conform features
- Table 2-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further RTE functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

- Table 2-3 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Explicit S/R communication (last-is-best) [API: Rte_Read, Rte_Write]
Explicit S/R communication (queued polling) [API: Rte_Receive, Rte_Send]
Variable length arrays
Explicit S/R communication (queued blocking) [API: Rte_Receive]
Implicit S/R communication [API: Rte_IRead, Rte_IWrite, Rte_IWriteRef]
Timeout handling (DataReceiveErrorEvent) [API: Rte_IStatus]
Data element invalidation [API: Rte_Invalidate, Rte_IInvalidate]
Intra-Ecu S/R communication
Inter-Ecu S/R communication
1:N S/R communication (including network signal Fan-Out)
N:1 S/R communication (non-queued and queued, pure network signal Fan-In or pure Intra-Ecu)
C/S communication (synchronous, direct calls) [API: Rte_Call]
C/S communication (synchronous, scheduled calls) [API: Rte_Call]
C/S communication (asynchronous calls) [API: Rte_Call]
C/S communication (asynchronous) [API: Rte_Result]
Intra-Ecu C/S communication
Inter-Ecu C/S communication using SOME/IP Transformer
N:1 C/S communication
Intra-Ecu trigger communication (via port) [API: Rte_Trigger]
Inter-Runnable Trigger (SWC internal) [API: Rte_IrTrigger]
Explicit exclusive areas [API: Rte_Enter, Rte_Exit]
Implicit exclusive areas
Explicit Inter-Runnable Variables [API: Rte_IrvRead, Rte_IrvWrite]

Supported AUTOSAR Standard Conform Features
Implicit Inter-Runnable Variables [API: Rte_IrvIRead, Rte_IrvIWrite, Rte_IrvIWriteRef]
Transmission ack. status (polling and blocking) [API: Rte_Feedback]
Runnable category 1a, 1b und 2
RTE Lifecycle-API [API: Rte_Start, Rte_Stop]
Nv Block Software Components
Runnable to task mapping
Data element to signal mapping
Task body generation
VFB-Tracing
Multiple trace clients
ECU-C import / export
Automatic OS configuration according the needs of the RTE (basic and extended task support)
Automatic COM / LDCOM configuration according the needs of the RTE
Primitive data types
Composite data types
Data reception triggered runnables entities (DataReceivedEvent)
Cyclic triggered runnable entities (TimingEvent)
Data transmission triggered runnable entities (DataSendCompletionEvent)
Data reception error triggered runnables entities (DataReceiveErrorEvent)
Mode switch acknowledge triggered runnable entities (ModeSwitchedAckEvent)
Mode switch triggered runnable entities (ModeSwitchEvent)
Background triggered runnable and scheduleable entities (BackgroundEvent)
Contract phase header generation
Port access to services (Port defined argument values)
Port access to ECU-Abstraction
Compatibility mode
Per-Instance Memory [API: Rte_Pim]
Multiple instantiation on ECU-level
Indirect API [API: Rte_Port, Rte_NPorts, Rte_Ports]
SWC internal calibration parameters [API: Rte_CData]
Shared calibration parameters (CalprmComponentType) [API: Rte_Prm]
Mode machine handling [API: Rte_Mode/Rte_Switch]
Mode switch ack. status (polling and blocking) [API: Rte_SwitchAck]
Multi-Core support (S/R communication, C/S communication, Mode communication (partially), NvM)
Memory protection
Unconnected ports
COM-Filter (NewDiffersOld, Always)

Supported AUTOSAR Standard Conform Features
Measurement (S/R-Communication, Mode-Communication, Inter-Runnable Variables)
Runnable de-bouncing (Minimum Start Interval)
Online calibration support
Never received status
S/R update handling [API: Rte_IsUpdated]
Contract Phase Header generation for BSW-Scheduler
PR-Ports
Optimized S/R communication [API: Rte_DRead]
Variant Handling support (Postbuild selectable for variant data mappings and COM signals)
Data prototype mapping (see limitations below)
Subelement mapping for Rx GroupSignals
Bit field texttable mapping
Activation reason for runnable entities (no support for multicore and memory protection)
RteUseComShadowSignalApi
Service BSW multiple partition distribution
S/R and C/S Serialization using SOME/IP Transformer
LdCom Support
ComXf Support
DiagXf Support
E2E Transformer Support
Transformer Error Handling for S/R
Transformer Error Handling for client-server communication
Triggering of executable entities on TransformerHardErrorEvent
Initialization of send buffers for implicit S/R communication
Data conversion (for inter-ECU communication limited to S/R communication with integer network signal(s) mapped to floating point data types on SWC ports, compu methods of type LINEAR, IDENTICAL or SCALE_LINEAR_AND_TEXTTABLE and data type policy LEGACY or OVERRIDE, for intra-ECU communication limited to S/R communication with compu methods of type LINEAR, IDENTICAL and (BITFIELD-)TEXTTABLE)
LDCOM TP API for S/R communication
Support generation of functions for implicit access
Enhanced Rte Lifecycle API Rte_StartTiming
Rte Implementation Plugins for access protection of explicit sender-receiver communication (buffer provided by RTE)
Rte Implementation Plugins explicit sender-receiver communication (buffer provided by plugin, cross or local, no multiple instantiation, no transformerError parameter)
Rte Implementation Plugins for implicit sender-receiver communication (buffer provided by plugin, cross or local, no status flags, no multiple instantiation, no transformerError parameter)
Rte Implementation Plugins for Fill and Flush Routines

Supported AUTOSAR Standard Conform Features
Rte Implementation Plugins for mode communication (only cross, mode switch task needs to be a basic task, mode provider needs to use a mode receiver in the same cluster, mode switch triggers need to be configured for every mode, no multiple instantiation, no transformerError parameter)
Rule Based Value Specifications
Support BaseType category VOID
Enhanced SchM Lifecycle APIs SchM_Start and SchM_StartTiming
Tx-Ack for implicit communication [API: Rte_IFeedback]
Data write completed triggered runnables entities (DataWriteCompletedEvent)
Support for different alive timeout values for the same signal in different SWCs
Support for internal event handling
Support for Logical Execution Time (LET)
Support for McSupportData
Generation on windows and linux (x86_64 glibc 2.26)
Support for Nv Sub-Element Mapping
Support for direct connection of SenderReceiver and Calibration Ports
NotAvailableValueSpecification for optional elements
DET check if API is called from the wrong entity RTE_E_DET_ILLEGAL_INVOCATION
VFB Trace Hook generation according to AUTOSAR 20-11.
BSW-Scheduler Calibration Parameter Access [API: SchM_CData]
Support of SwRecordLayout

Table 2-1 Supported AUTOSAR standard conform features

2.1.1 Deviations

The following features specified in [1] are not supported:

Category	Description
Functional	COM-Filter (only partially supported: only on rx side, filter settings)
Functional	Measurement (Client-Server arguments, previous and next mode of mode queues, RAM mirror of NvBlockSWCs)
Functional	Inter-Ecu trigger communication (via port) [API: Rte_Trigger] (only partially supported, E.g. usage without SomeIpXf and LDCOM IF API, FanIn, FanOut, Variant Signals are not supported)
API	BSW-Scheduler Mode Handling [API: SchM_Mode, SchM_Switch, SchM_SwitchAck]
Functional	external Trigger between BSW modules [API: SchM_Trigger]
API	BSW-Scheduler Trigger [API: SchM_ActMainFunction]
API	BSW-Scheduler queued S/R communication [API: SchM_Send, SchM_Receive]
API	BSW-Scheduler C/S communication [API: SchM_Call, SchM_Result]

Category	Description
API	BSW-Scheduler Per-Instance Memory Access [API: SchM_Pim]
Functional	Post Build Variant Sets
Functional	Debugging and Logging Support
Functional	Variant Handling support (Pre-Compile variability including variability of array sizes and enabling of APIs per variant, Postbuild variability for Connectors and ComponentPrototypes)
Functional	Multi-Core support (Rte_ComSendSignalProxyImmediate, RteIocInteractionReturnValue=RTE_COM)
Functional	Restarting of partitions including mode error reaction behavior and SwcModeManagerErrorEvent
Functional	Re-scaling of ports / Data conversion (only partially supported)
Functional	Pre-Build data set generation phase
Functional	Post-Build data set generation phase
Functional	Special handling for cluster generation phase (checks for extended tasks etc.)
Functional	Asynchronous Mode Handling
Functional	Generated BSWMD
Functional	Range checks for transformed signals, LDCOM signals, Internal-/External communication, RTE implementation plugins, signals that are assigned to a different partition than the application or behavior other than SATURATE. Moreover the out of range return value is not supported.
Functional	RTE memory section initialization strategies and configuration of dedicated SwAddrMethods for every generated element. SwAddrMethod configuration is only supported for per-instance memories, calibration parameters and NvBlockDescriptors. No partition specific headers are generated. The partition specific implementation file names contain the name of the OS application.
Functional	Configuration of coherency groups for implicit communication: Currently only supported for internal sender-receiver communication without status flags and triggers. Coherent and immediate buffer update need to be set to true. All runnables need to use the same trigger conditions and need to be mapped to the same task. Runnable are not allowed to have read and write accesses to the same data at the same time. The consistency needs are not validated against the RTE configuration.
Config	External configuration switch strictConfigurationCheck
Functional	ScaleLinear and ScaleLinearTexttable CompuMethods with more than one CompuScale
Functional	LDCOM TP API for C/S communication
Functional	Pointer data types (when not used as in parameters on client-server service port interfaces)
Functional	Multiple instantiation for NvBlock SWCs
Config	Instance specific SwDataDefProps and event props
Functional	Record layout annotations for maps & curves

Category	Description
Functional	Return value RTE_E_COM_STOPPED for Rte_Read when MICROSAR Classic COM is not used
Functional	Return value RTE_E_COM_STOPPED for Rte_IStatus
Functional	Return values RTE_E_HARD_TRANSFORMER_ERROR and RTE_E_SOFT_TRANSFORMER_ERROR for Rte_Feedback and Rte_IFeedback
Functional	Minimum Start Interval for runnables with a single cyclic trigger
Functional	Bypassing of RTE APIs including RptHooks
API	Rte_Init
Functional	Mapping of init runnables to basic tasks that contain runnables with other triggers
Functional	Usage of APIs with waitpoints in init runnables
Functional	Initialization of implicit sender-receiver buffers for N:1 communication
Functional	Usage of COM and LDCOM for the same data element
Functional	PortInterface Mapping (Degradation on a different level than the first, Degradation with different element names, Connection of record data elements with different record element names for internal communication, Connection of record data elements with different record element order, sub element mapping of array elements, sub element mapping in combination with bitfield texttable mapping, bitfield texttable mapping for external communication, subelement mapping for queued and implicit communication, mapping of mode, client-server and trigger interfaces, usage of bitfield texttable mappings for connections to SWCs outside the local partition, usage of E2EXF in combination with subelement mapping or degradation, etc.)
Config	AUTOSAR modelling for optional elements on record data types
Config	Mapping of SWCs in the extract. It can only be done in the ECUC configuration.
Config	SWC to BSW mapping. The mapping happens implicitly through the entity symbols.
Functional	Custom transformers
Config	SenderReceiverCompositeElementToSignalMapping
Functional	Inter-ECU client-server communication without SomeIpXf
Config	Client ID and Sequence Counter mapping for Inter-ECU client-server communication
Functional	Nesting of exclusive areas
Config	Resource consumption generation
Functional	External replacement of invalid values and timeout substitution values
Config	Constant mapping
Config	Timex extensions
Functional	Variation proxies and system constants
Functional	Reception of data smaller than the minimum possible serialized length. Transformers might thus report no data instead of invalid data. The MICROSAR Classic RTE drops incomplete data already in the callbacks to reduce CPU usage for invalid data.
Config	RteModeToScheduleTableMapping

Category	Description
Functional	Variable size streams when SomeIpXf is used in combination with COM. This functionality is only supported for LDCOM.
Functional	Error codes RTE_E_IN_EXCLUSIVE_AREA, RTE_E_SEG_FAULT, RTE_E_OUT_OF_RANGE
Functional	Rte_Send and Rte_Write return RTE_E_TIMEOUT for TP signals when the TP buffer is locked.
Functional	Rte_Send returns RTE_E_INVALID when an invalid size is given for a variable length signal.
Functional	Rte Implementation Plugins (only partially supported, see Chapter 4.17 for details) [API: Rte_Rips_Prm, Rte_Rips_Enter, Rte_Rips_Exit, Rte_Rips_EnterModeQueue, Rte_Rips_ExitModeQueue, Rte_Rips_DsmqSwitch, Rte_Rips_DsmqTransitionStart, Rte_Rips_DsmqTransitionSync, Rte_Rips_DsmqTransitionEnd, Rte_Rips_Invoke Rte_Rips_ReturnResult, Rte_Rips_InvocationHandler, Rte_Rips_NotifyRxAck Rte_Rips_NotifyRxTOut, Rte_Rips_NotifyTxAck, Rte_Rips_NotifyTxErr Rte_Rips_NotifyTxTOut, Rte_Rips_Trigger, Rte_Rips_DataIsUpdated, Rte_Rips_SchM_Init, Rte_Rips_Rte_Start, Rte_Rips_Rte_Stop, Rte_Rips_SchM_Deinit], Global copy defines, Rips for inter-runnable variables, fill-flush routines for non-cyclic runnables
Config	Calculation of AccessingApplications in the OS configuration based on individual runnables (currently it is a superset for all runnables in one component)
Functional	Metadata handling (currently only supported for Client-Server with SoAd, LDCOM and PduR on the same core. The RTE always appends the metadata to the payload. This is also done in the copy and rx callbacks. This implementation is compatible with MICROSAR Classic SoAd but different from the AUTOSAR specification.) [API: Rte_SetMetaDataItem, Rte_GetMetaDataItem, Rte_GetMetaDataLength]
Config	Connections between sender-receiver and calibration ports
Functional	Rte_IFeedback when no DataWriteCompletedEvent is configured. The API is generated but will always return RTE_E_TRANSMIT_ACK
Functional	N:1 communication when the receiver uses RTE implementation plugins Combination of local and cross cluster plugins on a receiver port Status flags for implicit cross cluster plugins
Config	AUTOSAR ECUC parameters for implementation plugins (URI References)
Functional	Dynamic length signals without transformers are only supported for COM.
Functional	Never received handling for NvBlockSWCs
Functional	IsUpdated handling for NvBlockSWCs
Functional	Mapping of individual bits in NvBlock data mapping
Functional	Distributed Shared Mode Queues
Functional	Usage of mode switch triggers for NvRunnables
Functional	SetRamBlockStatus API call inside the writing API, it is called in the NvRunnable instead
Functional	Usage of NvM_WritePRAMBlock in NvRunnables, NvM_WriteBlock is used instead

Category	Description
Functional	Invalidation for LDCOM signals
Functional	RoleBasedDataAssignment
Functional	Support for BulkNvDataDescriptor
Functional	ExclusiveAccess optimization
Functional	RTE_E_RAM_UNCHANGED return value for Rte_NvMNotifyInitBlock
Config	Multipartition NVM including NvMBlockEcucPartitionRef
Functional	swImplPolicy measurement point
Functional	Activation reasons and minimum start interval for schedulable entities
Functional	OsTaskPeriod/Trigger polling
Config	Overwriting of settings with the FlatInstanceDescriptor (initial value etc.)
Functional	No <task> and <application> extension in the API names for the OS hooks, no entity generation for the hooks
Functional	Monitoring of runnable execution time
Functional	Task chaining
Functional	Cyclic activation of background tasks
Functional	Trusted function generation for direct client-server calls
Functional	Configuration of synchronization points
Config	Different filter and invalidation settings for different receivers that are connected to the same sender.
Functional	Generation of return type void instead of Std_ReturnType based on the returnValueProvision parameter
Functional	ImmediateRestart of entities
Functional	Entity to ISR mapping
Config	RteToolChainSignificantCharacters
Functional	ArrayImplPolicy PayloadAsPointerToArray
Functional	Read access by reference [API: Rte_IReadRef, Rte_IrvIReadRef]
Functional	Data provision for Com_MainFunctionTx on request [API: Rte_COMCbkTxPrep]
Functional	RTE_RUNNABLEAPI Support
Functional	Generation of Rte_IStatus for implicit read accesses without never received, alive timeout handling and invalidation when handleStatusData is set to TRUE
Functional	Transformer status forwarding
Functional	Transformer protocol header peeking
Functional	ServerArgumentImplPolicy UseVoid
Functional	Entity extension for exclusive area APIs
Functional	Argument accessibility checks RTE_E_SEG_FAULT
Functional	Error Defines RTE_E_SERIALIZATION_ERROR, RTE_E_SERIALIZATION_LIMIT, RTE_E_TRANSFORMER_LIMIT
Functional	Rte prefix for data types

Category	Description
Functional	Checking if API is called in exclusive area, Return codes RTE_E_IN_EXCLUSIVE_AREA, RTE_E_DET_WAIT_IN_EXCLUSIVE_AREA, parameter RteInExclusiveAreaCheckEnabled
Config	BswSchedulerNamePrefix
Config	RteModeMachineQueueLength ECUC parameters
Functional	Queued external trigger communication
Functional	OsTaskExecutionEvent and BswOsTaskExecutionEvent triggered entities
Functional	ExclusiveAreaImplementation methods NONE, OS_SPINLOCK and RTE_PLUGIN for RTE exclusive areas
Functional	TransformerError for Rte_Hook functions.
Functional	TransformerHardErrorEvent in combination with external trigger.
Functional	Only one TransformerHardErrorEvent trigger per runnable is allowed.
Functional	Only a single client task is supported for asynchronous client-server calls with polling result and timeout
Config	Only one data mapping per systemsignal/systemsignalgroup is possible. If multiple SWCs send or receive the same data, the mapping needs to be done on a connected delegation port.
Functional	The following data types are currently not supported if the ArrayImplPolicy is set to PayloadAsPointerToArray: Unions, CalibrationParameters, Inter Runnable Variables, Per Instance Memory. In addition, array data types cannot be used in operation ports of a client server port.
Functional	The following use cases are currently not supported if the the ArrayImplPolicy for the used data type is set to PayloadAsPointerToArray: Queued / Implicit – communication.
Functional	For Inter-ECU client-server communication all call and return signals need to be mapped to the same partition and for COM, need to be handled by the same COM mainfunction.
Functional	TxAck is not supported when the COM instance of the signals is located in a different partition.
Functional	Dynamic length signals consisting of uint8 arrays without SomeIpXf are only supported for COM, not LDCOM. They can also only be received in the same partition as the relevant COM signal.

Table 2-2 Not supported AUTOSAR standard conform features

Note
Additional limitations due to AUTOSAR requirements or technical limitations (in the generator) may apply when combining the features listed in chapter 2.1. Contact us if in doubt.

2.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Rte_InitMemory API function. See Chapter 4.14.4 for details.
VFB Trace Hooks for SchM APIs. See Chapter 4.16.3 and 4.16.4 for details.
Measurement support for mode communication. See Chapter 5.7 for details.
Measurement with XCP Events. See Chapter 5.7 for details.

Table 2-3 Features provided beyond the AUTOSAR standard

2.1.3 Limitations

There are no known limitations.

2.2 Initialization

The RTE is initialized by calling `Rte_Start (and Rte_StartTiming, if enabled)`. Initialization is done by the ECU State Manager (EcuM).

The Basis Software Scheduler (SchM) is initialized by calling `SchM_Init (and SchM_Start and SchM_StartTiming, if enabled)`. Initialization is done by the ECU State Manager (EcuM).

The usage of the Api for the initialization of cyclic trigger (`SetRelAlarm` or `SetAbsAlarm`) can be selected with the parameter `RteGeneration/RteUseAbsAlarmInitialization`. The problem with the use of `SetRelAlarm` is that between the different calls for different alarm initialization the time advances so that in the end the alarm expire has different OS tick times. With the use of `SetAbsAlarm` it can be achieved that all runnables are executed in the same OS tick. This ensures that the execution order of runnables is exactly as defined in the task mapping. The usage of `SetAbsAlarm` requires the configuration of proper offsets, see Chapter 5.11, because if the activation offset is not sufficient the alarm activation may occur after the next counter overrun.

It is possible to configure a central start offset for all cyclic runnables. To enable this feature the RTE BSWMD parameter RteAlarmGlobalStartOffset has to be configured and set to a value other than 0 and 1. The `SetRelAlarm` or `SetAbsAlarm` Api will use the tick offset as additional offset to the task mapping specific offsets. If nothing is configured, the default value of 1 is used.

2.3 AUTOSAR ECUs

Besides the basic software modules each AUTOSAR ECU has a single instance of the RTE to manage the application software of the ECU. The application software is modularized and assigned to one or more AUTOSAR software components (SWC).

2.4 AUTOSAR Software Components

AUTOSAR software components (SWC) are described by their ports for communication with other SWCs and their internal behavior in form of runnable entities realizing the smallest schedulable code fragments in an ECU.

The following communication paradigms are supported for port communication:

- Sender-Receiver (S/R): queued and last-is-best, implicit and explicit
- Client-Server (C/S): synchronous and asynchronous
- Mode communication
- Calibration parameter communication

S/R and C/S communication may occur Intra-ECU or Inter-ECU (between different ECUs). Mode communication and calibration parameters can only be accessed ECU internally.

In addition to Inter-SWC communication over ports, the description of the internal behavior of SWCs contains also means for Intra-SWC communication and synchronization of runnable entities.

- Inter-Runnable Variables
- Per-Instance Memory
- Exclusive Areas
- Calibration Parameters

The description of the internal behavior of SWCs finally contains all information needed for the handling of runnable entities, especially the events upon which they are triggered.

2.5 Runnable Entities

All application code is organized into runnable entities, which are triggered by the RTE depending on certain conditions. They are mapped to OS tasks and may access the communication and data consistency mechanisms provided by the SWC they belong to.

The trigger conditions for runnable entities are described below, together with the signature of the runnable entities that results from these trigger conditions. A detailed description of the signature of runnable entities may be found in section 4.3 Runnable Entities.

2.6 Triggering of Runnable Entities

AUTOSAR has introduced the concept of RTEEvents to trigger the execution of runnable entities. The following RTEEvents are supported by the MICROSAR Classic RTE:

- TimingEvent
- DataReceivedEvent
- DataReceiveErrorEvent
- DataSendCompletedEvent
- OperationInvokedEvent
- AsynchronousServerCallReturnsEvent
- ModeSwitchEvent
- ModeSwitchedAckEvent
- InitEvent
- BackgroundEvent
- ExternalTriggerOccurredEvent
- InternalTriggerOccurredEvent
- DataWriteCompletedEvent

The RTEEvents can lead to two different kinds of triggering:

- Activation of runnable entity
- Wakeup of waitpoint

`Activation of runnable entity` starts a runnable entity at its entry point while `wakeup of waitpoint` resumes runnable processing at a waitpoint. A wakup of waitpoint is not applicable for all kind of RTEEvents and needs to be set up inside the runnable entity code using a dedicated RTE API.

Depending on the existence of a waitpoint, runnable entities are categorized into category 1 or category 2 runnables. A runnable becomes a category 2 runnable if at least one waitpoint exists.

Caution
The RTE assumes that runnables without triggers are never called. However, if they are called, the APIs may not work as intended as the RTE optimizes the APIs based on the call context.

2.6.1 Time Triggered Runnables

AUTOSAR defines the `TimingEvent` for periodic triggering of runnable entities. The `TimingEvent` can only trigger a runnable entity at its entry point. Consequently, there exists no API to set up a waitpoint for a `TimingEvent`. The signature of a time triggered runnable is:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.2 Data Received Triggered Runnables

AUTOSAR defines the `DataReceivedEvent` to trigger a runnable entity on data reception (queued or last-is-best) or to continue to receive queued data in a blocking `Rte_Receive` call. Both intra ECU and inter ECU communication is supported. Data reception triggered runnables have the following signature:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.3 Data Reception Error Triggered Runnables

AUTOSAR defines the `DataReceiveErrorEvent` to trigger a runnable entity on data reception error. A reception error could be a timeout (`aliveTimeout`) or an invalidated data element. The `DataReceiveErrorEvent` can only trigger a runnable entity at its entry point. Consequently, there exists no API to set up a waitpoint for a `DataReceiveErrorEvent`. The signature of a data reception error triggered runnable is:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.4 Data Send Completed Triggered Runnables

AUTOSAR defines the `DataSendCompletedEvent` to signal a successful or an erroneous transmission of explicit S/R communication. The `DataSendCompletedEvent` can either trigger the execution of a runnable entity or continue a runnable, which is waiting at a waitpoint for the transmission status or the mode switch in a blocking `Rte_Feedback` call.

The DataSendCompletedEvent is raised in the communication callbacks. Beware that the communication callbacks are not triggered when the communication is already stopped at the time of the transmission. In this case the RTE APIs return RTE_E_COM_STOPPED immediately.

Both intra ECU and inter ECU communication is supported. Data send completed triggered runnables have the following signature:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.5 Mode Switch Triggered Runnables

AUTOSAR defines the `ModeSwitchEvent` to trigger a runnable entity on either entering or exiting of a specific mode of a mode declaration group. The `ModeSwitchEvent` can only trigger a runnable entity at its entry point. Consequently, there exists no API to set up a waitpoint for a `ModeSwitchEvent`. The signature of a mode switch triggered runnable is:

```
void <RunnableName>([IN Rte_Instance instance]
             [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.6 Mode Switched Acknowledge Triggered Runnables

AUTOSAR defines the `ModeSwitchedAckEvent` to signal a successful mode transition. The `ModeSwitchedAckEvent` can either trigger the execution of a runnable entity or continue a runnable, which is waiting at a waitpoint for the mode transition status. Only intra ECU communication is supported. Runnables triggered by a mode switch acknowledge have the following signature:

```
void <RunnableName>([IN Rte_Instance instance]
             [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.7 Operation Invocation Triggered Runnables

The `OperationInvokedEvent` is defined by AUTOSAR to always trigger the execution of a runnable entity. The signature of server runnables depends on the parameters defined at the C/S port. Its general appearance is as follows:

```
{void|Std_ReturnType} <Runnable>([IN Rte_Instance inst] {,paramlist}*)
```

The return value depends on application errors being assigned to the operation that the runnable represents. The parameter list contains input input/output and output parameters. Input parameters for primitive data type are passed by value. Input parameters for composite data types and all input/output and output parameters independent whether they are primitive or composite types are passed by reference. The string data type is handled like a composite type. The server queue size from the server COM spec can be overwritten with the parameter

/MICROSAR/Rte/RteSwComponentInstance/RteEventToTaskMapping/RteServerQueueLength.

2.6.8 Asynchronous Server Call Return Triggered Runnables

The `AsynchronousServerCallReturnsEvent` signals the end of an asynchronous server execution and triggers either a runnable entity to collect the result by using `Rte_Result` or resumes the waitpoint of a blocking `Rte_Result` call.

The runnables have the following signature:

```
void <RunnableName>([IN Rte_Instance instance]
             [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.9 Init Triggered Runnables

Initialization runnables are called once during startup and have the following signature:

```
void <RunnableName>([IN Rte_Instance instance])
```

2.6.10 Background Triggered Runnables

Background triggered runnables have to be mapped to tasks with the lowest priority. These tasks are repeatedly scheduled by the RTE in the background while no other task with higher priority is active. The signature of a background triggered runnable is:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.11 ExternalTriggerOccurredEvent Triggered Runnables

AUTOSAR defines the `ExternalTriggerOccuredEvent` to trigger a runnable entity directly. The `ExternalTriggerOccuredEvent` can only trigger a runnable entity at its entry point. Consequently, there exists no API to set up a waitpoint for a `ExternalTriggerOccuredEvent`. The signature of an external triggered runnable is:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.12 InternalTriggerOccurredEvent Triggered Runnables

The `InternalTriggerOccuredEvent` is raised when a runnable is triggered by an internal triggering point of the same software-component instance. The `InternalTriggerOccurredEvent` can only trigger a runnable entity at its entry point. The signature of an internal triggered runnable is:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.6.13 Data Write Competed Event Triggered Runnables

AUTOSAR defines the DataWriteCompletedEvent to signal successful or an erroneous transmission of implicit S/R communication. The DataWriteCompletedEvent is raised in the communication callbacks and triggers the execution of a runnable. Beware that the communication callbacks are not triggered when the communication is already stopped at the time of the transmission. In this case the RTE APIs return RTE_E_COM_STOPPED immediately. Inside the runnable with DataWriteCompletedEvent, the transmission status is obtained with Rte_IFeedback API. Data write completed triggered runnables have the following signature:

```
void <RunnableName>([IN Rte_Instance instance]
              [,IN Rte_ActivatingEvent_<RunnableEntity> activation])
```

2.7 Exclusive Areas

An exclusive area (EA) can be used to protect either only a part of runnable code (explicit EA access) or the complete runnable code (implicit EA access). AUTOSAR specifies four implementation methods which are configured during ECU configuration of the RTE. See also Chapter 5.10.

- OS Interrupt Blocking
- All Interrupt Blocking
- OS Resource
- Cooperative Runnable Placement

In addition, the BSW modules EAs have the implementation method

- OS Spinlock

All of them have to ensure that the current runnable is not preempted while executing the code inside the exclusive area.

Additionally, it is also possible to set the implementation method to one of the following.

- None
- Custom

The MICROSAR Classic RTE analyzes the accesses to the different RTE exclusive areas and eliminates unneeded ones if that is possible e.g. runnable entities accessing the same EA cannot preempt each other.

Info
For SchM exclusive areas the automatic optimization is currently not supported. Optimization must be done manually by setting the implementation method to `NONE`.

Caution
If the user selects implementation method `NONE` or `CUSTOM` it is in the responsibility of the user that the code between the `Rte_Enter` or `SchM_Enter` and the `Rte_Exit` or `SchM_Exit` still provides exclusive access to the protected area.

2.7.1 OS Interrupt Blocking

When an exclusive area uses the implementation method `OS_INTERRUPT_BLOCKING`, it is protected by calling the OS APIs `SuspendOSInterrupts()` and `ResumeOSInterrupts()`. The OS does not allow the invocation of event and resource handling functions while interrupts are suspended. Hence, any call to a RTE API that is based upon an explicitly modeled waitpoint like: blocking `Rte_Receive`, `Rte_Feedback`, `Rte_SwitchAck` or `Rte_Result` API, as well as synchronous server calls (which sometimes use waitpoints that are not explicitly modeled or other rescheduling points) are not allowed. Additionally, all APIs that might trigger a runnable entity on the same ECU or cause a blocking queue access to be released are forbidden, as well as exclusive areas implemented as OS Resource.

2.7.2 All Interrupt Blocking

When an exclusive area uses the implementation method `ALL_INTERRUPT_BLOCKING`, it is protected by calling the OS APIs `SuspendAllInterrupts()` and `ResumeAllInterrupts()`. The OS does not allow the invocation of event and resource handling functions while interrupts are suspended. This precludes calls to any RTE API that is based upon an explicitly modeled waitpoint (blocking `Rte_Receive`, `Rte_Feedback`, `Rte_SwitchAck` or `Rte_Result` API) as well as synchronous server calls (which sometimes use waitpoints that are not explicitly modeled or other rescheduling points). Additionally, all APIs that might trigger a runnable entity on the same ECU or cause a blocking queue access to be released are forbidden, as well as exclusive areas implemented as OS Resource.

2.7.3 OS Resource

An exclusive area using implementation method `OS_RESOURCE` is protected by OS resources entered and released via `GetResource()` / `ReleaseResource()` calls, which raise the task priority so that no other task using the same resource may run. The OS does not allow the invocation of `WaitEvent()` while an OS resource is occupied. This again precludes calls to any RTE API that is based upon an explicitly modeled waitpoint and synchronous server calls.

2.7.4 Cooperative Runnable Placement

For exclusive areas with implementation method `COOPERATIVE_RUNNABLE_PLACEMENT`, the RTE generator only applies a check whether any of the tasks accessing the exclusive area are able to preempt any other task of that group. This again precludes calls to any RTE API that is based upon an explicitly modeled waitpoint and synchronous server calls.

2.7.5 OS Spinlock

The implementation method `OS_SPINLOCK` is available for BSW modules EAs. It allows the EA to be protected against race conditions in multicore systems. For this a Spinlock is created in the OS. Furthermore, this method ensures that the current runnable is not preempted while executing the code inside the exclusive area. For this `SuspendAllInterrupts()` and `ResumeAllInterrupts()` are called, if the Spinlock is not locking its area by default.

2.7.6 None

For exclusive areas with implementation method `NONE` the RTE generator creates functionally empty implementations for all required APIs. It therefore does not provide any exclusive access to the protected area. This is then in the responsibility of the user.

2.7.7 Custom

When an exclusive area uses the implementation method `CUSTOM`, the RTE generator doesn’t generate the `Rte_Enter` / `SchM_Enter` and `Rte_Exit` / `SchM_Exit` APIs. It only generates their prototypes. It is necessary to implement those APIs manually by the customer.

Caution
The custom exclusive area APIs need to implement a sequentially consistent acquire and release fence/compiler barrier to prevent memory reordering of any read or write which precedes them in program order with any read or write which follows them in program order.

2.8 Error Handling

2.8.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [21], if development error reporting is enabled in the `RteGeneration` parameters (i.e. by setting the parameters `DevErrorDetect` and / or `DevErrorDetectUninit)`.

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`. The reported RTE ID is 2.

The reported service IDs identify the services which are described in chapter 4. The following table presents the service IDs and the related services:

Service ID	Service
0x00	SchM_Init
0x01	SchM_Deinit
0x03	SchM_Enter
0x04	SchM_Exit
0x70	SchM_Start
0x76	SchM_StartTiming
0x13	Rte_Send
0x14	Rte_Write
0x15	Rte_Switch
0x16	Rte_Invalidate
0x17	Rte_Feedback
0x18	Rte_SwitchAck
0x19	Rte_Read
0x1A	Rte_DRead
0x1B	Rte_Receive
0x1C	Rte_Call
0x1D	Rte_Result
0x1F	Rte_CData
0x20	Rte_Prm
0x28	Rte_IrvRead
0x29	Rte_IrvWrite
0x2A	Rte_Enter
0x2B	Rte_Exit
0x2C	Rte_Mode
0x2D	Rte_Trigger
0x2E	Rte_IrTrigger
0x30	Rte_IsUpdated
0x70	Rte_Start

Service ID	Service
0x71	Rte_Stop
0x90	Rte_COMCbkTAck_<sn>
0x91	Rte_COMCbkTErr_<sn>
0x92	Rte_COMCbkInv_<sn>
0x93	Rte_COMCbkRxTOut_<sn>
0x94	Rte_COMCbkTxTOut_<sn>
0x95	Rte_COMCbk_<sg>
0x96	Rte_COMCbkTAck_<sg>
0x97	Rte_COMCbkTErr_<sg>
0x98	Rte_COMCbkInv_<sg>
0x99	Rte_COMCbkRxTOut_<sg>
0x9A	Rte_COMCbkTxTOut_<sg>
0x9B	Rte_SetMirror_<b>_<d>
0x9C	Rte_GetMirror_<b>_<d>
0x9D	Rte_NVMNotifyJobFinished_<b>_<d>
0x9E	Rte_NVMNotifyInitBlock_<b>_<d>
0x9F	Rte_COMCbk_<sn>
0xA0	Rte_LdComCbkRxIndication_<sn>
0xA1	Rte_LdComCbkStartOfReception_<sn>
0xA2	Rte_LdComCbkCopyRxData_<sn>
0xA3	Rte_LdComCbkTpRxIndiciation_<sn>
0xA4	Rte_LdComCbkCopyTxData_<sn>
0xA5	Rte_LdComCbkTpTxConfirmation_<sn>
0xA6	Rte_LdComCbkTriggerTransmit_<sn>
0xA7	Rte_LdComCbkTxConfirmation_<sn>
0xF0	Rte_Task
0xF1	Rte_IncDisableFlags
0xF2	Rte_DecDisableFlags

Table 2-4 Service IDs

The errors reported to DET are described in the following table:

Error Code	Description
RTE_E_DET_ILLEGAL_NESTED_EXCLUSIVE_AREA	The same exclusive area was called nested or exclusive areas were not exited in the reverse order they were entered
RTE_E_DET_UNINIT	Rte/SchM is not initialized
RTE_E_DET_MODEARGUMENT	Rte_Switch was called with an invalid mode parameter
RTE_E_DET_TRIGGERDISABLECOUNTER	Counter of mode disabling triggers is in an invalid state
RTE_E_DET_MODESTATE	Mode machine is in an invalid state
RTE_E_DET_MULTICORE_STARTUP	Initialization of the master core before all slave cores were initialized
RTE_E_DET_ILLEGAL_SIGNAL_ID	RTE callback was called for a signal that is not active in the current variant.
RTE_E_DET_ILLEGAL_VARIANT_CRITERION_VALUE	SchM_Init called with wrong variant
RTE_E_DET_BLOCKSIZECHECK	Nv Block size mismatch between the RTE and NVM
RTE_E_DET_SCHM_STARTUP	SchM_Init called before SchM_Start
RTE_E_DET_ILLEGAL_INVOCATION	API is called from the wrong entity

Table 2-5 Errors reported to DET

The error RTE_E_DET_UNINIT will only be reported if the parameter `DevErrorDetectUninit` is enabled. The reporting of all other errors can be enabled by setting the parameter `DevErrorDetect`.

Caution
If `DevErrorDetect` is enabled in multicore systems, the DET module needs to provide a multicore reentrant `Det_ReportError` method.

2.9 Dirty Flag Handling

The MICROSAR Classic RTE supports NvBlock SWCs that can interact with the NVM to automatically persist the written values to NV memory when the dirty flag handling is configured. The following settings are possible:

Shutdown	Immediate	Cyclic	Behavior
			The RTE does not call WriteBlock or SetRamBlockStatus automatically. Use the service ports instead.
		■	NvRunnable with cyclic trigger periodically calls WriteBlock when the data element was written.
	■		NvRunnable with data reception triggers calls WriteBlock when the data elements are written for which the data reception triggers are configured.
	■	■	NvRunnable with data reception triggers calls WriteBlock when the data elements are written for which the data reception triggers are configured. NvRunnable with cyclic trigger periodically calls WriteBlock if the data element was written.
■			NvRunnable with data reception trigger calls SetRamBlockStatus when the element is written by any port.
■		■	NvRunnable with data reception trigger calls SetRamBlockStatus when the element is written by any port. NvRunnable with cyclic trigger periodically calls WriteBlock if the data element was written.
■	■		NvRunnable with data reception trigger calls SetRamBlockStatus when the element is written by any port. NvRunnable with data reception triggers calls WriteBlock when the data elements are written for which the data reception triggers are configured.
■	■	■	NvRunnable with data reception trigger calls SetRamBlockStatus when the element is written by any port. NvRunnable with data reception triggers calls WriteBlock when the data elements are written for which the data reception triggers are configured. NvRunnable with cyclic trigger periodically calls WriteBlock if the data element was written.

Table 2-6 Dirty Flag Handling

In case an NVM job is still pending when the NvRunnable tries to write the block, the RTE retries the operation when the NVM notifies the RTE that the previous job is finished.

2.10 Illegal Invocation Detection

The MICROSAR Classic RTE supports the detection of illegal RTE API invocations. A check routine can be activated, that recognizes RTE API Calls called by a Runnable which should not call the RTE API according to the Runnables contract. In the case, that such an illegal

API invocation is detected, the RTE reports an `RTE_E_DET_ILLEGAL_INVOCATION` to the DET.

This check applies for any RTE API, for which a VFB Trace Hook can be configured. Supported RTE APIs therefore are:

- Enter
- Exit
- Write
- Read
- DRead
- Send
- Receive
- Invalidate
- IsUpdated
- Feedback
- SwitchAck
- Switch
- Mode
- Call
- Result
- IrvWrite
- IrvRead
- Trigger
- IrTrigger

Info
This Check is **not** activated by default. A description how to activate it can be found in 5.17.

Caution
This Check will **not** work for RTE APIs, for which no VFB Trace Hook can be configured. These RTE APIs remain unchecked, even with fully activated Illegal Invocation Detection.

Caution
This check will not work, if any OS other than the Vector Os Gen7 is used.

3 RTE Generation and Integration

This chapter gives necessary information for the integration of the MICROSAR Classic RTE into an application environment of an ECU. Moreover the RTE generation process including a description about the different RTE generation modes is described.

3.1 Embedded Implementation

The delivery of the RTE consists of:

File	Description	Integration Tasks
Rte_<CT>.h	Generated file that contains the APIs for one SWC. It needs to be included into the SWC code. This header file is the only file to be included in the component code. It is generated to the `Components` subdirectory by default.	-
Rte_<CT>_Type.h	Generated file that contains SWC specific type definitions. It is generated to the `Components` subdirectory by default.	-
SchM_<BSWM>.h	Generated file that contains the APIs for one BSW module. It needs to be included into the BSW module code.	-
SchM_<BSWM>_Type.h	Generated file that contains BSW module specific type definitions.	-
<CT>_MemMap.h	Generated file with template areas that can be adapted by the user. Template contains SWC specific part of the memory mapping. It is generated to the `Components` subdirectory by default. The file is only generated by the RTE generator when the SWC provides no resource consumption or when the memmap generator is not available.	Adapt the dedicated code areas within that file. See hints within that file.
Rte.c	Generated file that contains the main implementation of the RTE.	-
Rte_<OsApplication>.c	Generated file that contains OsApplication specific parts of the generated RTE (only generated when OsApplications are configured).	-
Rte_PBCfg.c	Generated file that contains the data structures for the postbuild RTE initialization.	-
Rte.h	Generated file that contains RTE internal declarations.	-
Rte_Main.h	Generated file that contains the lifecycle API.	-

Rte_Cfg.h	Generated file that contains the configuration for the RTE.	-
Rte_Cbk.h	Generated file that contains prototypes for COM callbacks.	-
Rte_Hook.h	Generated file that contains relevant information for VFB tracing.	-
Rte_Type.h	Generated file that contains the application defined data type definitions and RTE internal data types.	-
Rte_DataHandleType.h	Generated file that contains the data handle type declarations required for the component data structures.	-
Rte_PBCfg.h	Generated file that contains the declarations for the data structures that are used for the postbuild RTE initialization.	-
Rte_UserTypes.h	Generated file with template areas that can be adapted by the user. It is generated if either user defined data types are required for Per-Instance memory or if a data type is used by the RTE but generation is skipped with the `typeEmitter` attribute.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_MemMap.h	Generated file with template areas that can be adapted by the user. It contains the RTE specific part of the memory mapping. The file is only generated by the RTE generator when the memmap generator is not available.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_Compiler_Cfg.h	Generated file with template areas that can be adapted by the user. It contains the RTE specific part of the compiler abstraction.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_Buffers.h	Generated file with global copies of data instantiated by the RTE for RTE Implementation Plugins. The file is only generated by the RTE generator if RTE Implementation Plugins are configured.	-
usrostyp.h	Generated file with template areas that can be adapted by the user. It is only generated if memory protection support is enabled. In that case this file is included by the MICROSAR Classic OS.	Adapt the dedicated code areas within that file. See hints within that file.
Rte.oil	Generated file that contains the OS configuration for the RTE.	-
Rte_Needs.ecuc.arxml	Generated file that contains the RTE requirements on BSW module configuration for Os, Com, LdCom, Xcp and NVM.	-

Rte.a2l	Generated A2L file containing CHARACTERISTIC and MEASUREMENT objects for the generated RTE.	-
Rte_MemSeg.a2l	Generated A2L file containing MEMORY_SEGMENT objects for the generated RTE.	-
Rte.html	Generated file that contains information about RAM / CONST consumption of the generated RTE as well as a listing of all triggers and their OS events and alarms.	-

Table 3-1 Generated Files of RTE Generation Phase

3.2 RTE Generation

The MICROSAR Classic RTE generator can be called either from the command line application `DVCfgCmd` or directly from within the DaVinci Configurator.

Please refer to the CFG5 command line help for an explanation of the general parameters.

3.2.1 Command Line Options

Option		Description
`--project <file>`	`-p <file>`	Specifies the absolute path to the DPA project file. When -g is used to generate. Specifies the absolute path to an ARXML file if –generateSwcsOnly is used. Multiple ARXML files can be passed by specifying the parameter multiple times.
`--generate`	`-g`	Generate the given project specified in `<file>`. Only valid when -p references a DPA project.
`--generateSwcsOnly`		Generates contract phase headers or implementation templates from individual ARXML files that are passed to the -p parameter.
`--modulesToGenerate`	`-m <module>`	Specifies the module definition references, which should be generated by the `-g` switch. Separate multiple modules by providing a comma separted list of the module names. E.g. /MICROSAR/Rte, /MICROSAR/Nm
`--genArg="<module>: <params>"`		Passes the specified parameters `<params>` to the generator of the specified module `<module>`. For details of the possible parameters of the RTE module see Table 3-3.
`--swcsToGenerate <SWC>`		Specifies the applications software component types for which a template and contract phase headers should be generated. If empty "", then no templates are generated. Multiple SWCs can be be separated by ",". The identifier is the short name or AutosarPath. E.g. --swcsToGenerate "SWC_A, SWC_B, /ComponentTypes/SWC_C". With argument “default” a template will be generated for all software components activated in DPA project file. If no arguments are passed to this option, all available SWCs are generated. Use the –generateSwcsOnly parameter in case ARXML files shall be used as input.
`--help`	`-h`	Displays the general help information of `DVCfgCmd.exe`

Table 3-2 DVCfgCmd Command Line Options

3.2.2 RTE Generator Command Line Options

Option	Description
`-m <obj>`	Selects the DaVinci model object, where `<obj>` is either `<ECUProjectName>` or `<ComponentTypeName>`. Note: While the options `-g i` and `-g c` can be combined, the component type object is prioritized in case <ComponentTypeName> and <ECUProjectName> are the same. When the workspace contains only a single ECUProject or a single ComponentType, the `-m` parameter can be omitted. With the `-m` parameter also multiple component types can be selected, delimited with semicolons.
`-g [r|c|i|h]`	Selects generation of the RTE with the following sub options:
	`r` — Selects the RTE generation for the ECU project specified via option `-m <ECUProjectName>`. This is the default option. Therefore `-g` is equal to `-g r`.
	`c` — Selects the RTE contract phase header generation for a single component type or BSW module if `-m <ComponentTypeName/BswModuleName>` or for multiple component types and BSW modules if `-m <ComponentType1Name/BswModule1Name>;<ComponentType2Name/BswModule2Name>` or for all non-service component types and BSW modules of an ECU project if `-m <ECUProjectName>`.
	`i` — Selects implementation template generation for a single component type if `-m <ComponentTypeName>` or for multiple component types if `-m <ComponentType1Name>;<ComponentType2Name>` or for all non- service component types of an ECU project if `-m <ECUProjectName>`. The optional `-f <file>` parameter specifies the file name to use for the implementation template file. If the `-f <file>` parameter is not given, or `-m` contains an ECU project name, the filename defaults to `<ComponentTypeName>.c`. Already existing implementation files are updated and a backup is created.
	`h` — Selects VFB trace hook template generation for the ECU project specified via option `-m <ECUProjectName>`. The optional `-f <file>` parameter specifies the file name to use for the VFB trace hook template file. If the `-f <file>` parameter is not given, the filename defaults to `VFBTraceHook_<ECUProjectName>.c`. Already existing implementation files are updated and a backup is created.
	This parameter can be used more than one time to generate several modes in one step.

-o <path> -o r=<path> -o c=<path> -o i=<path> -o h=<path> -o s=<path> -o a=<path>	Output path for the generated files. If more than one generation mode is active, a special path can be specified for each generation mode by assigning the path to the character that is used as sub option for the -g parameter. Furthermore, the path for the application header files in the RTE generation mode can be selected via option -o s=<path>. By default, they are generated into the subdirectory Components. The path for A2L files can be specified with the option -o a=<path>. These files are generated into the RTE directory by default. Note: The <path> configured with -o parameter overwrites the path which is specified in the dpa project file.
-f <file>	Optional parameter to specify the output file name for options -g i and -g h. Note: For option -g i the output file name can only be specified if -m specifies a component type. The output file name cannot be specified when -m specifies multiple component types.
--generateSwcts	Generates software component headers.
--disablerteanalyzer	Disables RteAnalyzer and skips stub generation.
-v	Enables verbose mode which includes help information for error, warning and info messages.
-h	Displays the general help information.
--rtecachepath=<path>	Set the path to cache folder. Default is GenData folder. e.g., --rtecachepathe=C:\cacheFolder
-dc	Disable conditional generation and print timestamps to the generated files in the -generateSwcts and -generateSwcsOnly modes

Table 3-3 RTE Generator Command Line Options

3.2.3 Generation Path

The RTE output files are generated into the path which is either specified within the dpa project file or which is specified in the -o command line option. If several generation modes are activated in parallel, for each phase a special path can be specified with the -o command line option.

In the RTE generation phase (command line option -g or -g r), the component type specific application header files are generated into the subdirectory Components. This subdirectory can be changed in the RTE generation phase with the option -o "s=<path>". In addition, the directory for the A2L files, which are generated into the RTE directory by default, can be specified with the option -o "a=<path>".

During generation, the RTE generator also writes files to the temporary directory of the system. Unicode characters are currently not supported for the path of this directory.

3.3 MICROSAR Classic RTE generation modes

The sections give an overview of the files generated by the MICROSAR Classic RTE generator in the different RTE generation modes and some examples how the command line call looks like.

3.3.1 RTE Generation Phase

The files from chapter 3.1 are generated by the RTE generation: (Option `-g` or `-g r`)

Example:

```
DVCfgCmd -p "InteriorLight.dpa" -m /MICROSAR/Rte -g
```

Caution

The linux version of the RTE Generator runs with reduced checks. It is recommended to check the model in Davinci Developer before generation.

The windows version of the RTE Generator automatically invokes additional model consistency checks that are based on Davinci Developer during generation. These checks are not available on linux.

3.3.2 RTE Contract Phase Generation

The following files are generated by the RTE contract phase generation: (Option `-g c` or `-swcsToGenerate` or `--generateSwcsOnly`)

File	Description	Integration Tasks
Rte_<CT>.h	Generated file that contains the APIs for one SWC. It must be included into the SWC code. This header file is the only file to be included in the component code.	-
Rte_<CT>_Type.h	Generated file that contains SWC specific type definitions.	-
<CT>_MemMap.h	Generated file with template areas that contains SWC specific part of the memory mapping.	Adapt the dedicated code areas within that file. See hints within that file.
Rte.h	Generated file with RTE internal declarations.	-
Rte_Type.h	Generated file that contains the application defined data type definitions and RTE internal data types.	-
Rte_DataHandleType.h	Generated file that contains the data handle type declarations required for the component data structures.	-
Rte_UserTypes.h	Generated file with template areas which is generated if either user defined data types are required for Per-Instance memory or if a data type is used by the RTE but generation is skipped with the `typeEmitter` attribute.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_MemMap.h	Generated file with template areas that contains RTE specific part of the memory mapping.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_Compiler_Cfg.h	Generated file with template areas that contains RTE specific part of the compiler abstraction.	Adapt the dedicated code areas within that file. See hints within that file.
SchM_<BSWM>.h	Generated file that contains the APIs for one BSW module. It needs to be included into the BSW module code.	-
SchM_<BSWM>_Type.h	Generated file that contains BSW module specific type definitions.	-

Table 3-4 Generated Files of RTE Contract Phase

Example:

```
DVCfgCmd -p "InteriorLight.dpa"
    -m /MICROSAR/Rte
    -g
    --genArg=”Rte: -g c –m SenderComponent”
```

or

```
DVCfgCmd -p "SenderComponent.arxml"
    --generateSwcsOnly
    --genArg=”Rte: -g c –m SenderComponent”
```

The generated header files are located in a component type specific subdirectory. The application header file must be included in each source file of a SWC implementation, where the RTE API for that specific SWC type is used.

The application header file created in the RTE contract phase can be used to compile object code components, which can be linked to a generated RTE in the RTE generation phase. The application header files are generated in RTE compatibility mode.

Caution
During the RTE generation phase an optimized header file is generated. This optimized header file should be used when compiling the source code SWCs during the ECU build process.
The usage of object code SWCs, which are compiled with the application header files of the contract phase, require an “Implementation Code Type” for SWCs set to “object code” in order to tell the RTE generator in the RTE generation phase NOT to create optimized RTE API accesses but compatible ones.

3.3.3 Template Code Generation for Application Software Components

The following file is generated by the implementation template generation: (Option `-g i`)

File	Description	Integration Tasks
<FileName>.c	Generated file with template areas that contains the runnable imlementations. An implementation template is generated if the `-g i` option is selected. The `-f` option specifies the name of the generated c file. If no name is selected the default name `<ComponentTypeName>.c` is used.	Adapt the dedicated code areas within that file. See hints within that file.

Table 3-5 Generated Files of RTE Template Code Generation

Example:

```
DVCfgCmd -p "InteriorLight.dpa"
    -m /MICROSAR/Rte
    -g
    --genArg=”Rte: -g i -m SenderComponent -f Component1.c”
```

The generated template files contain all empty bodies of the runnable entities for the selected component type. It also contains the include directive for the application header file. In addition, the available RTE API calls are included in comments.

Caution

When the destination file of the SWC template code generation is already available, code that is placed within the user code sections marked by “DO NOT CHANGE”-comments is transferred unchanged to the updated template file.

Additionally, a numbered backup of the original file is made before the new file is written.

The preservation of runnable code is done by checking for the runnable symbol. This implies that after a change of the name of a runnable the runnable implementation is preserved, while a change of the symbol results in a new empty function for the runnable.

Code that was removed during an update is kept in the “removed code” section at the bottom of the implementation file and in the numbered backups.

The template update is particularly useful when e.g. access to some interfaces has been added or removed from a runnable, because then the information of available APIs is updated by the generation process without destroying the implementation.

3.3.4 VFB Trace Hook Template Code Generation

The following file is generated by the VFB trace hook template generation: (Option `-g h`)

File	Description	Integration Tasks
<FileName>.c	Generated file with template areas that contains the VFB trace hook imlementations. An implementation template of the VFB trace hooks is generated if the `-g h` option is selected. The `-f` option specifies the name of the generated c file. If no name is selected the default name `VFBTraceHook_< ECUProjectName >.c` is used.	Adapt the dedicated code areas within that file. See hints within that file.

Table 3-6 Generated Files of VFB Trace Hook Code Generation

Example:

```
DVCfgCmd -p "InteriorLight.dpa"
    -m /MICROSAR/Rte
    -g
    --genArg=”Rte: -g h -f VFBTraceHook_myEcu.c”
```

Caution

When the destination file of the VFB trace hook template generation is already available, code that is placed within the user code sections marked by “DO NOT CHANGE” comments is transferred unchanged to the updated template file.

Additionally, a numbered backup of the original file is made before the new file is written.

The preservation of trace hook code is done by checking for the trace hook name. When the name of a hook changes, e.g. because the name of a data element changed, then the code of the previous trace hook is removed.

Code that was removed during an update is kept in the “removed code” section at the bottom of the implementation file and in the numbered backups.

The template update is particularly useful when some trace hooks have been added or removed due to changed interfaces or OS usage.

Info

The generated VFB trace hook template will only include hooks, that have been configured according to AUTOSAR 4.2.

VFB trace client hooks, that have been configured using the RteVfbTraceClient container according to AUTOSAR 20-11 will not be included in the generated VFB trace hook template.

3.4 Include Structure

3.4.1 RTE Include Structure

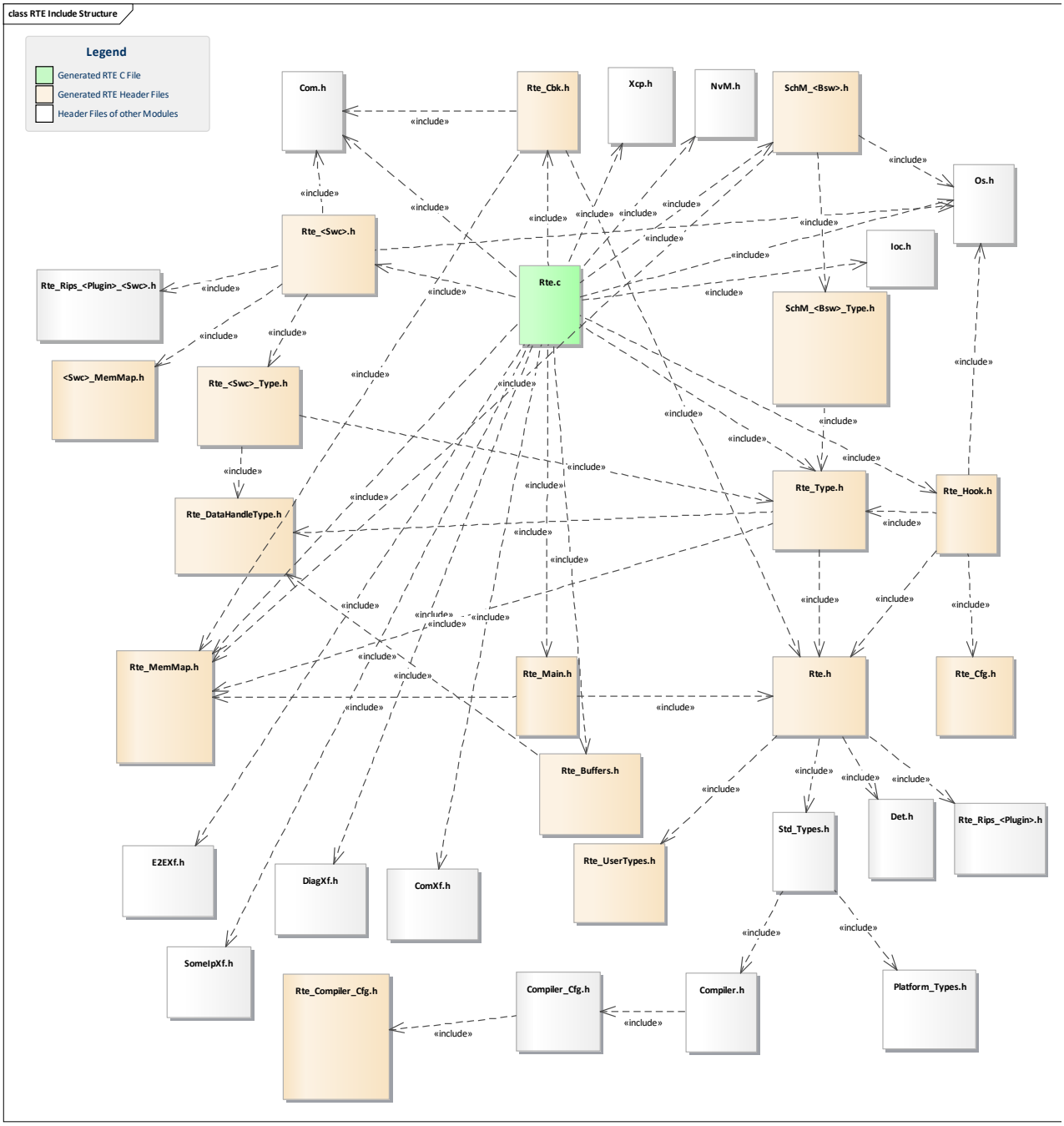


Figure 3-1 RTE Include Structure

3.4.2 SWC Include Structure

The following figure shows the include structure of a SWC with respect to the RTE dependency. All other header files which might be included by the SWC are not shown.

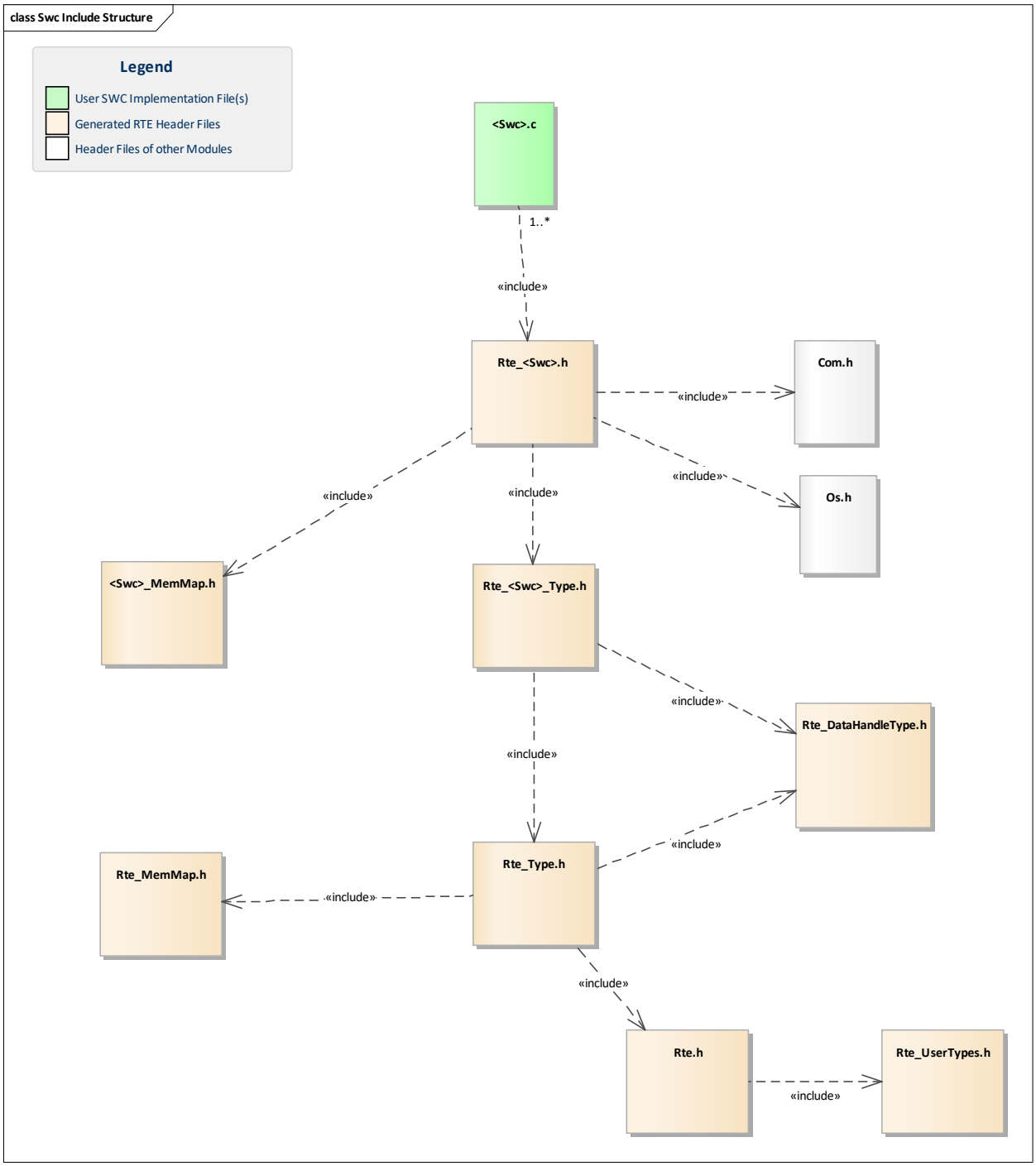


Figure 3-2 SWC Include Structure

3.4.3 BSW Include Structure

The following figure shows the include structure of a BSW module with respect to the SchM dependency. All other header files which might be included by the BSW module are not shown.

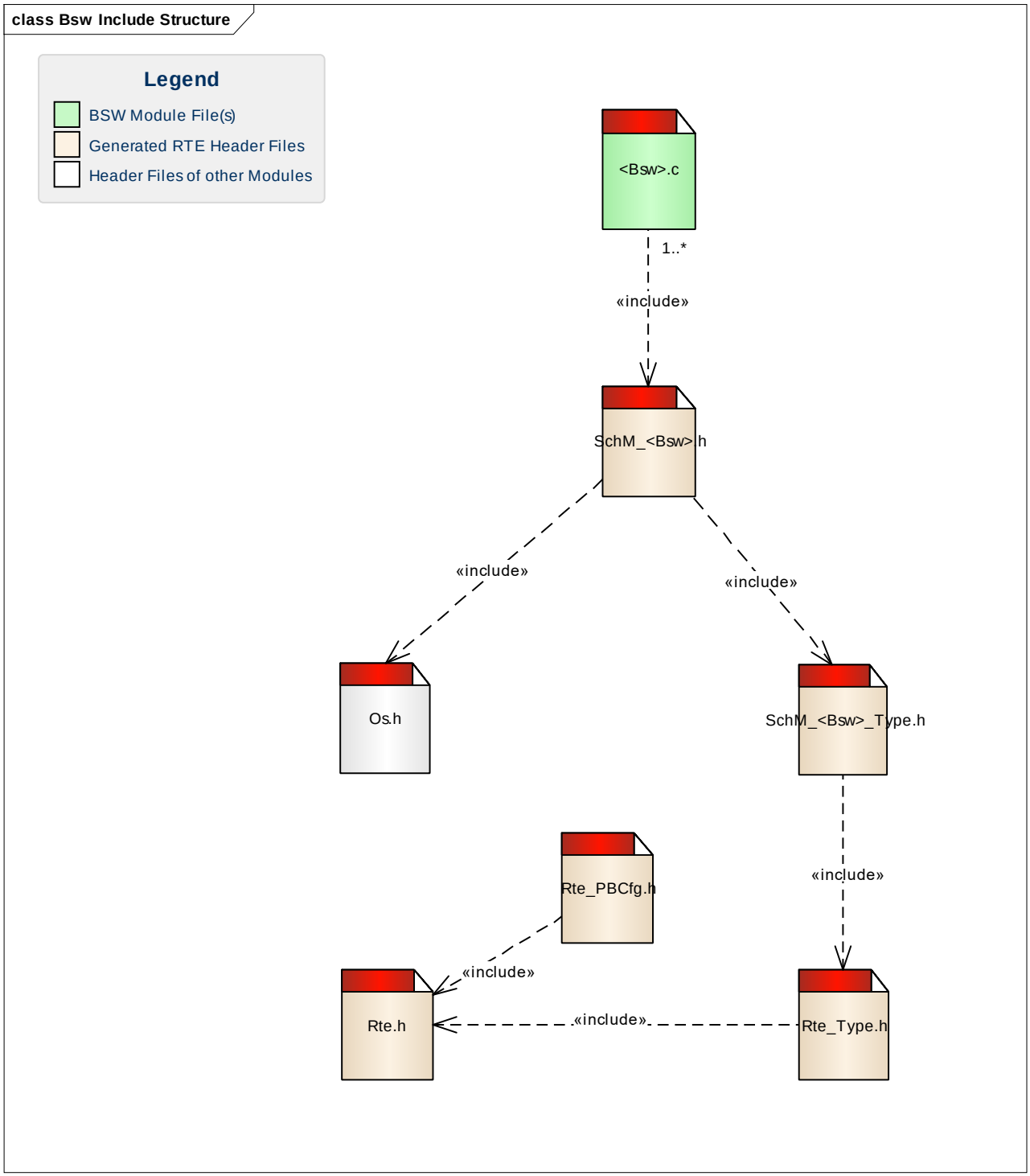


Figure 3-3 BSW Include Structure

3.5 Compiler Abstraction and Memory Mapping

The objects (e.g. variables, functions, constants) are declared by compiler independent definitions – the compiler abstraction definitions. Each compiler abstraction definition is assigned to a memory section.

The following two tables contain the memory section names and the compiler abstraction definitions defined for the RTE and illustrate their assignment among each other.

Memory Mapping Sections / Compiler Abstraction Definitions	RTE_VAR_ZERO_INIT	<Swc>_VAR_ZERO_INIT	RTE_VAR_INIT	<Swc>_VAR_INIT	RTE_VAR_NOINIT	<Swc>_VAR_NOINIT	RTE_VAR_<Pim>	RTE_<NvRamBlock>	RTE_VAR_<Cal>	RTE_CONST	<Swc>_CONST	RTE_CONST_<Cal>	RTE_CODE	<Swc>_CODE	RTE_APPL_CODE	RTE_<SWC>_APPL_CODE	RTE_<SWC>_APPL_VAR	RTE_<SWC>_APPL_DATA	RTE_APPL_VAR	RTE_APPL_DATA
RTE_START_SEC_VAR_ZERO_INIT_8BIT RTE_STOP_SEC_VAR_ZERO_INIT_8BIT	■																			
RTE_START_SEC_VAR_ZERO_INIT_UNSPECIFIED RTE_STOP_SEC_VAR_ZERO_INIT_UNSPECIFIED	■																			
RTE_START_SEC_VAR_<OsAppl>_ZERO_INIT_UNSPECIFIED[1] RTE_STOP_SEC_VAR_<OsAppl>_ZERO_INIT_UNSPECIFIED[1]	■																			
<Swc>_START_SEC_VAR_ZERO_INIT_UNSPECIFIED <Swc>_STOP_SEC_VAR_ZERO_INIT_UNSPECIFIED		■																		
RTE_START_SEC_VAR_INIT_UNSPECIFIED RTE_STOP_SEC_VAR_INIT_UNSPECIFIED			■																	
RTE_START_SEC_VAR_<OsAppl>_INIT_UNSPECIFIED[1] RTE_STOP_SEC_VAR_<OsAppl>_INIT_UNSPECIFIED[1]			■																	
<Swc>_START_SEC_VAR_INIT_UNSPECIFIED <Swc>_STOP_SEC_VAR_INIT_UNSPECIFIED				■																
RTE_START_SEC_VAR_NOINIT_UNSPECIFIED RTE_STOP_SEC_VAR_NOINIT_UNSPECIFIED					■															
RTE_START_SEC_VAR_<OsAppl>_NOINIT_UNSPECIFIED[1] RTE_STOP_SEC_VAR_<OsAppl>_NOINIT_UNSPECIFIED[1]					■															
<Swc>_START_SEC_VAR_NOINIT_UNSPECIFIED <Swc>_STOP_SEC_VAR_NOINIT_UNSPECIFIED						■														
RTE_START_SEC_VAR_<Pim>_UNSPECIFIED RTE_STOP_SEC_VAR_<Pim>_UNSPECIFIED							■													
RTE_START_SEC_<NvRamBlock> RTE_STOP_SEC_<NvRamBlock>								■												
RTE_START_SEC_VAR_<Cal>_UNSPECIFIED RTE_STOP_SEC_VAR_<Cal>_UNSPECIFIED									■											
RTE_START_SEC_CONST_UNSPECIFIED RTE_STOP_SEC_CONST_UNSPECIFIED										■										
<Swc>_START_SEC_CONST_UNSPECIFIED <Swc>_STOP_SEC_CONST_UNSPECIFIED											■									
RTE_START_SEC_CONST_<Cal>_UNSPECIFIED RTE_STOP_SEC_CONST_<Cal>_UNSPECIFIED												■								
RTE_START_SEC_CODE RTE_STOP_SEC_CODE													■							
<Swc>_START_SEC_CODE <Swc>_STOP_SEC_CODE														■						
RTE_START_SEC_APPL_CODE RTE_STOP_SEC_APPL_CODE															■					
RTE_START_SEC_<OSAPPL>_CODE RTE_STOP_SEC_<OSAPPL>_CODE													■							

RTE_START_SEC_<TASKNAME>_CODE RTE_STOP_SEC_<TASKNAME>_CODE													■							

Table 3-7 Compiler abstraction and memory mapping

Memory Mapping Sections / Compiler Abstraction Definitions	RTE_VAR_ZERO_INIT_NOCACHE	RTE_VAR_INIT_NOCACHE	RTE_VAR_NOINIT_NOCACHE
RTE_START_SEC_VAR_NOCACHE_ZERO_INIT_8BIT RTE_STOP_SEC_VAR_NOCACHE_ZERO_INIT_8BIT	■		
RTE_START_SEC_VAR_GLOBALSHARED_NOCACHE_ZERO_INIT_8BIT RTE_STOP_SEC_VAR_GLOBALSHARED_NOCACHE_ZERO_INIT_8BIT	■		
RTE_START_SEC_VAR_NOCACHE_ZERO_INIT_UNSPECIFIED RTE_STOP_SEC_VAR_NOCACHE_ZERO_INIT_UNSPECIFIED	■		
RTE_START_SEC_VAR_<OsAppl>_NOCACHE_ZERO_INIT_UNSPECIFIED RTE_STOP_SEC_VAR_<OsAppl>_NOCACHE_ZERO_INIT_UNSPECIFIED	■		
RTE_START_SEC_VAR_NOCACHE_INIT_UNSPECIFIED RTE_STOP_SEC_VAR_NOCACHE_INIT_UNSPECIFIED		■	
RTE_START_SEC_VAR_<OsAppl>_NOCACHE_INIT_UNSPECIFIED RTE_STOP_SEC_VAR_<OsAppl>_NOCACHE_INIT_UNSPECIFIED		■	
RTE_START_SEC_VAR_NOCACHE_NOINIT_UNSPECIFIED RTE_STOP_SEC_VAR_NOCACHE_NOINIT_UNSPECIFIED			■
RTE_START_SEC_VAR_<OsAppl>_NOCACHE_NOINIT_UNSPECIFIED RTE_STOP_SEC_VAR_<OsAppl>_NOCACHE_NOINIT_UNSPECIFIED			■

Table 3-8 Compiler abstraction and memory mapping for non-cacheable variables

The memory mapping sections and compiler abstraction defines specified in Table 3-8 are only used for variables which are shared between different cores on multicore systems. These variables must be linked into non-cacheable memory.

In case a section has no <OsAppl> extension it needs to be mapped to the OS application that contains the communication stack and that initializes the RTE on that core.

The RTE specific parts of `Compiler_Cfg.h` and `MemMap.h` depend on the configuration of the RTE. Therefore, the MICROSAR Classic RTE generates templates for the following files:

- Rte_Compiler_Cfg.h
- Rte_MemMap.h (only when the memmap generator is not available)

[1] These memory mapping sections are only used if memory protection support is enabled

They can be included into the common files and should be adjusted by the integrator like the common files too.

Note
The task specific code sections <TASKNAME>_CODE may also be useful to insert compiler specific pragmas to disable certain compiler optimizations for this task. E.g. when common subexpression elimination leads to excessive stack usage for implicit accesses.

3.5.1 Memory Sections for Calibration Parameters and Per-Instance Memory

The variable part of the memory abstraction defines for calibration parameters `<Cal>` and Per-Instance Memory `<Pim>` depends on the configuration. In DaVinci Developer 4 and later, the memory sections can be configured directly as SwAddrMethod on the relevant objects.

For DaVinci Developer 3, the following table shows the attributes, which have to be defined in order to assign a calibration parameter or a Per-Instance Memory to one of the configured groups. The groups represented by the enumeration values of the attributes can be configured in the attribute definition dialog in DaVinci Developer 3 without any naming restrictions. Only the name of the attribute itself is predefined as described in the table below.

Object Type	Attribute Name	Attribute Type
Calibration Parameter	PAR_GROUP_CAL	Enumeration
Calibration Element Prototype	PAR_GROUP_EL	Enumeration
Per-Instance Memory	PAR_GROUP_PIM	Enumeration
NvBlockDataPrototype	PAR_GROUP_NVRAM	Enumeration

Details of configuration and usage of User-defined attributes can be found in the DaVinci Online Help [23].

Example for Calibration Parameters:

If the attribute `PAR_GROUP_CAL` contains e.g. the enumeration values `CalGroupA` and `CalGroupB` and calibration parameters are assigned to those groups, the RTE generator will create the following memory mapping defines:

```
RTE_START_SEC_CONST_CalGroupA_UNSPECIFIED
RTE_STOP_SEC_CONST_CalGroupA_UNSPECIFIED
RTE_START_SEC_CONST_CalGroupB_UNSPECIFIED
RTE_STOP_SEC_CONST_CalGroupB_UNSPECIFIED
```

In addition, the following memory mapping defines are generated, if the calibration method Initialized RAM is enabled (see also Chapter 5.7):

```
RTE_START_SEC_VAR_CalGroupA_UNSPECIFIED
RTE_STOP_SEC_VAR_CalGroupA_UNSPECIFIED
RTE_START_SEC_VAR_CalGroupB_UNSPECIFIED
RTE_STOP_SEC_VAR_CalGroupB_UNSPECIFIED
```

Example for Per-Instance Memory:

If the attribute `PAR_GROUP_PIM` contains e.g. the enumeration values `PimGroupA` and `PimGroupB` and Per-Instance Memory is assigned to those group, the RTE generator will create the following memory mapping defines:

```
RTE_START_SEC_VAR_PimGroupA_UNSPECIFIED
RTE_STOP_SEC_VAR_PimGroupA_UNSPECIFIED
RTE_START_SEC_VAR_PimGroupB_UNSPECIFIED
RTE_STOP_SEC_VAR_PimGroupB_UNSPECIFIED
```

3.5.2 Memory Sections for Software Components

The MICROSAR Classic RTE generator generates specific memory mapping defines for each SWC type which allows to locate SWC specific code, constants and variables in different memory segments.

The variable part `<Swc>` is the camel case software component type name. The RTE implementation template code generator (command line option `-g i`) uses the SWC specific sections to locate the runnable entities in the appropriate memory section.

Also, SwAddrMethods can be configured to map the runnable entities to specific memory sections.

The SWC type specific parts of `MemMap.h` depend on the configuration. The MICROSAR Classic RTE generator creates a template for each SWC according to the following naming rule when the SWC has no resource consumption or when the memmap generator is not available:

- <Swc>_MemMap.h

3.5.3 Memory Sections for InterRunnableVariables and Sender-Receiver Variables

Also for inter runnable variables and internal non-queued sender-receiver communication,

SwAddrMethods can be configured to map the variables to specific memory sections.

Moreover, when the SectionInitializationPolicy is configured to NO-INIT, CLEARED or POWER-ON-CLEARED, the initial value can be omitted in this case.

3.5.4 Compiler Abstraction Symbols for Software Components and RTE

The RTE generator uses SWC specific defines for the compiler abstraction.

The following define is used in RTE generated SW-C implementation templates in the runnable entity function definitions.

`<Swc>_CODE`

In addition, the following compiler abstraction defines are available for the SWC developer. They can be used to declare SWC specific function code, constants and variables.

```
<Swc>_CODE
<Swc>_CONST
<Swc>_VAR_NOINIT
<Swc>_VAR_INIT
<Swc>_VAR_ZERO_INIT
```

If the user code contains variable definitions, which are passed to the RTE API by reference in order to be modified by the RTE (e.g. buffers for reading data elements) the RTE uses the following define to specify the distance to the buffer.

`RTE_APPL_VAR` (RTE specific)

If the user code contains variable or constant definitions, which are passed to the RTE API as pure input parameter (e.g. buffers for sending data elements) the RTE uses the following define to specify the distance to the variable or constant.

`RTE_<SWC>_APPL_DATA` (SWC specific)
`RTE_APPL_DATA` (RTE specific)

All these SWC and RTE specific defines for the compiler abstraction might be adapted by the integrator. The configured distances have to fit with the distances of the buffers and the code of the application.

Caution

The template files `<Swc>_MemMap.h`, `Rte_MemMap.h` and `Rte_Compiler_Cfg.h` have to be adapted by the integrator depending on the used compiler and hardware platform especially if memory protection is enabled.

When the files are already available during RTE generation, the code that is placed within the user code sections marked by “DO NOT CHANGE”-comments is transferred unchanged to the updated template files. The behavior is the same as for template generation of other files like SWC template generation.

When the configuration is changed, e.g. an OS Application is renamed, the existing memmap definitions need to be changed manually.

3.6 Memory Protection Support

The MICROSAR Classic RTE supports memory protection as an extension to the AUTOSAR RTE specification. Therefore, the memory protection support of the Operating System provides the base functionality. The support is currently limited to the Vector MICROSAR Classic OS because the RTE requires read access to the data from all partitions which is not required by AUTOSAR. Moreover, when trusted functions are used, the RTE uses wrapper functions that are only generated by the MICROSAR Classic OS. These wrapper functions can be implemented manually by following the Chapter „Providing Trusted Functions“ of the AUTOSAR SWS OS (Version 4.0-4.3).

3.6.1 Partitioning of SWCs

The partitioning of SWCs to memory areas can be done in DaVinci CFG. The partitioning is based on assignment of tasks to OS Applications, which is part of the OS configuration process.

There exists the restriction that all Runnable Entities of a single SWC have to be assigned to the same OS Application. This restriction and the assignment of tasks to OS Applications indirectly assign SWCs to OS Applications. This is the basis for grouping SWCs to different memory partitions. Additional information about memory protection configuration can be found in Chapter 5.4.

3.6.2 OS Applications

The operating system supports different scalability classes (SC). Only in SC3 and SC4 the memory protection mechanism is available. Therefore, the configuration of the required scalability class is the first step to enable memory partitioning. The second step is the assignment of SWCs to partitions (OS Applications) which is done by assigning tasks to OS Applications as described above.

The OS supports two types of OS Applications:

- Non-Trusted
- Trusted

Non-Trusted OS Applications run with enabled memory protection, trusted OS Applications with disabled memory protection. When TrustedApplicationWithProtection for a Trusted application is configured, the RTE generator internally treats it similar to non-trusted applications and no longer assumes the trusted applications can write variables in other partitions.

Caution
Enable the error hook and the protection hook in the OS in order to get informed about MPU violations and misusage of the OS.

Both types are supported by the MICROSAR Classic RTE and can be selected in the OS Application configuration dialog or directly in the OS configuration tool.

Caution
If no assignment of tasks to OS Applications exist memory protection is disabled.

3.6.3 Partitioning Architecture

When memory protection is used, one or more SWCs can be assigned to an OS Application but it is not allowed to split up a SWC between two or more OS Applications. That means that all runnables of a SWC have to be assigned to tasks, which belong to the same OS Application. It is the responsibility of the RTE to transfer the data of S/R and the arguments of C/S port interfaces over the protection boundaries.

Caution
Client-Server invocations implemented as direct function calls should be used inside one OS Application only.

The MICROSAR Classic RTE itself and the BSW can either run in a trusted OS Application or in a non-trusted OS Application. Both architectures are described below.

3.6.3.1 Trusted RTE and BSW

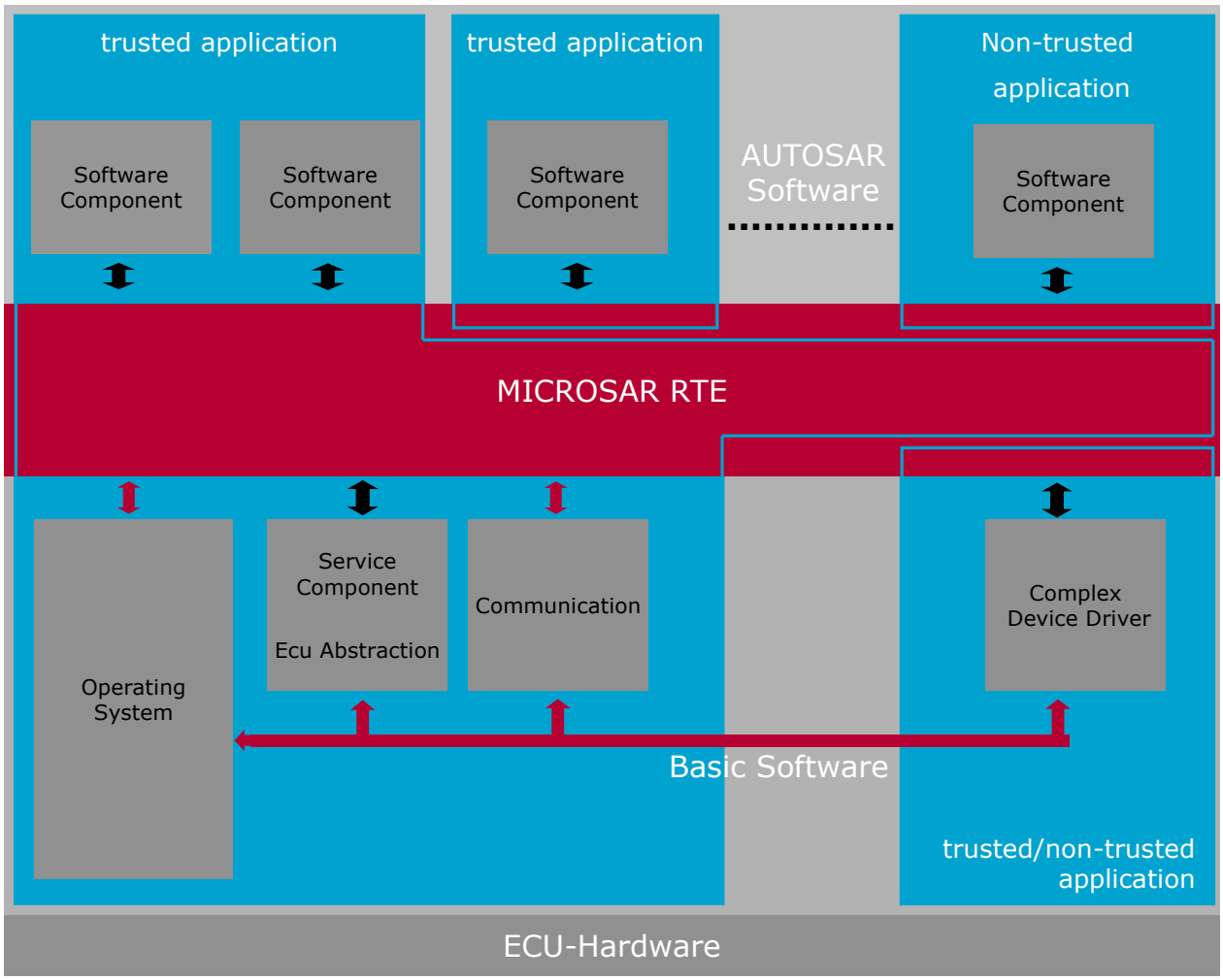


Figure 3-4 Trusted RTE Partitioning example

This architecture overview assumes that the RTE and the BSW modules that are used by the RTE run in one or more trusted OS Applications. Application software components (SWC) above the RTE can either be trusted or non-trusted. When this architecture is used,

the RTE uses trusted functions so that non-trusted SWCs can access the BSW and SWCs in other OS Applications. In this architecture, `Rte_Start()` has to be called from a trusted task and the Com module needs to run in trusted context, too. The RTE will use the same OS Application as the BSW Scheduler tasks.

An architecture where the BSW modules and the RTE are assigned to a non-trusted OS Application is described in the next chapter.

3.6.3.2 Non-Trusted RTE and BSW

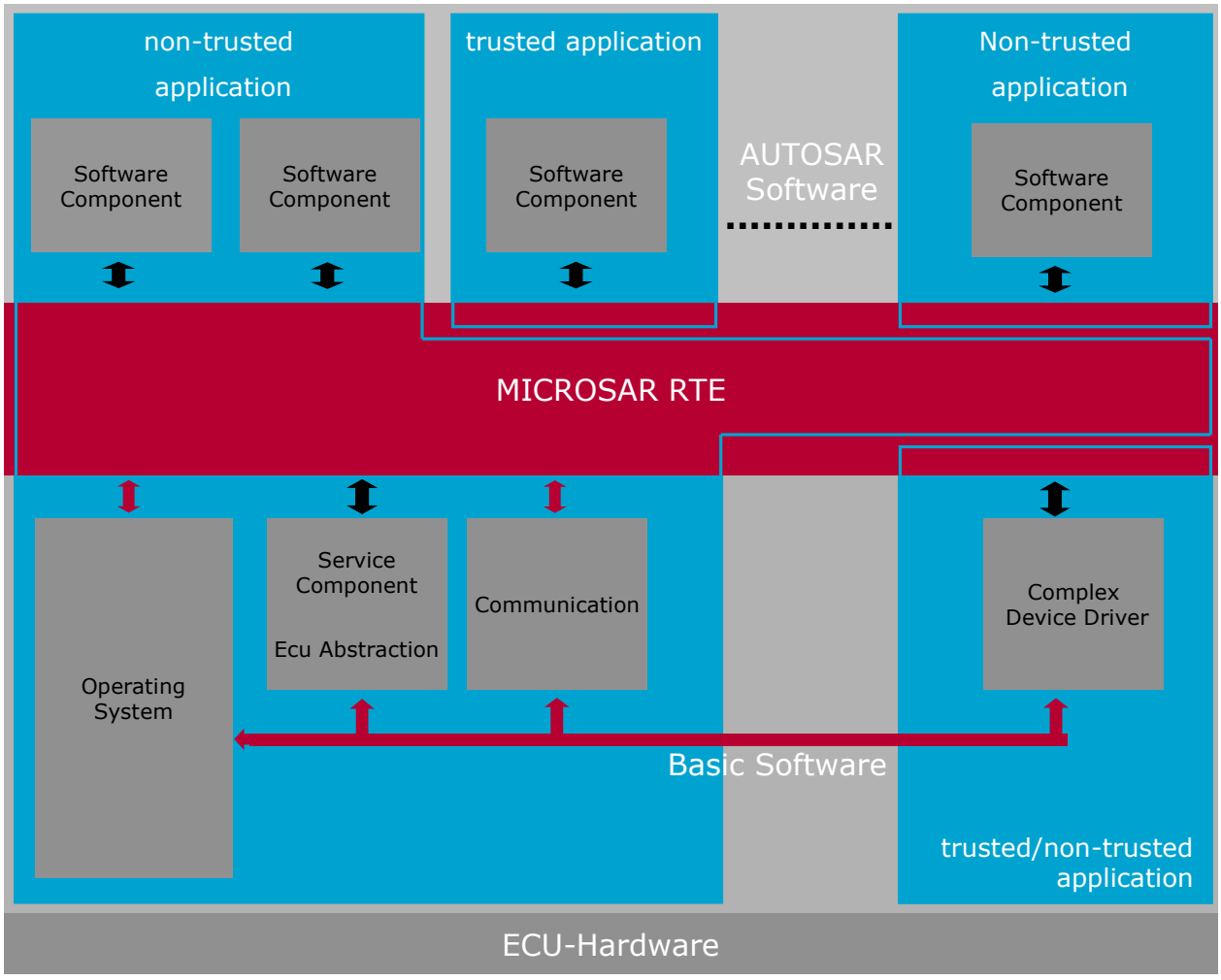


Figure 3-5 Non-trusted RTE Partitioning example

This architecture overview assumes that the BSW modules below the RTE, as well as the RTE itself run in a single non-trusted OS Application. The SWCs above the RTE can either be assigned to the same non-trusted OS Application as the BSW or they can be assigned to one or more other non-trusted or trusted OS Applications. Every OS Application has its own buffers which are only written by runnables that run in the same OS Application. The RTE does not use trusted functions in this architecture. Therefore, it is possible to create a system where all SWCs and the BSW are assigned to non-trusted OS Applications and all runnables/tasks always run with enabled memory protection.

The non-trusted RTE architecture is automatically chosen when the RTE configuration fulfills the following criterions:

- The tasks that contain the BSW modules are known by the RTE. This can be achieved by configuring them as BSW Scheduler tasks (See chapter 5.3).
- All BSW Scheduler tasks are assigned to the same non-trusted OS Application (further referred to as BSW OS Application). It is assumed that this is also the OS Application that initializes the RTE by calling `Rte_Start`. The application tasks can either be assigned to the BSW OS Application or to other non-trusted or trusted OS Applications.
- There are no direct client/server calls across OS Applications

No special limitations apply to SWCs that are assigned to the same OS Application as the BSW. Moreover, the following RTE features can also be used by SWCs in other OS Applications:

- direct or buffered inter-runnable variables
- per-instance memories
- SWC local calibration parameters
- access to calibration software components
- queued and nonqueued intra-ECU sender/receiver communication (when there is only a single sender partition)
- inter-ECU sender/receiver communication (see also chapter 3.8.1)
- direct client/server calls to runnables within the same OS Application
- mapped client/server calls to runnables in the same and different OS Applications (see also chapter 3.8.2)
- reading modes with the Rte_Mode API
- explicit and implicit exclusive areas

3.6.4 Conceptual Aspects

For intra OS Application communication no additional RTE interaction is required. Special memory protection handling is required only if communication between different OS Applications exists. In that case, the RTE has to provide means to transfer data over the protection boundaries. The only possibility is the usage of trusted function calls inside the generated RTE code. Those trusted function calls are expensive concerning code usage and runtime. Therefore, the usage of trusted function calls should be minimized if possible.

The MICROSAR Classic RTE generator uses trusted function calls only if necessary. In some cases, the usage of trusted function calls can be avoided by assigning the RTE buffers to the appropriate OS Application. The Vector MICROSAR Classic OS only provides write access protection but doesn't support read access protection. This behavior is the basis to avoid trusted function calls, because the writing OS Application can be seen as the owner of the memory buffer. Only a simple assignment of the buffer to the appropriate OS Application is necessary. This also makes it possible to completely avoid trusted function calls when the "Non-trusted RTE" architecture (chapter 3.6.3.2) is chosen.

Only if the memory assignment cannot be used, the RTE needs trusted functions to cross the protection boundaries.

For that purpose, the RTE generator uses the OS Application of the BSW Scheduler tasks as its own OS Application, which always needs to be trusted. The trusted functions called by the RTE are assigned to that trusted OS Application.

The RTE generator automatically creates trusted functions to write to RAM Mirrors of NvBlockSWCs when NVM is mapped to a different partition. The RAM mirror is mapped to the NVM partition. When all writers of a NvBlockDescriptor are in the same partition and NVM runs in a trusted partition, the RAM mirror is mapped to the partition of the writers to avoid the overhead of the trusted function calls.

In addition to the communication between SWCs of different OS Applications, there also exists communication between the COM BSW module and the RTE. Especially the notifications of the COM are done in an undefined call context. The MICROSAR Classic RTE assumes that the calls of the COM callbacks are done from the OS Application that contains the BSW Scheduler tasks.

3.6.5 Memory Protection Integration Hints

3.6.5.1 Enabling of Memory Protection support

Please make sure that memory protection is enabled by assigning tasks to OS Applications and by selecting scalability class SC3 or SC4 in the OS configuration.

3.6.5.2 Memory mapping in Linker Command File

If memory protection is enabled, the RTE generator creates additional OS Application specific memory sections for variables: In addition, the user has to split up his Per-Instance Memory (PIM) sections to the different OS Applications. These additional memory sections have to be mapped in the linker command file to the appropriate memory segments. See OS and compiler / linker manual for details.

The individual memory sections are listed in chapter 3.5.

The standard RTE memory section defines need to be mapped to the same segments as the BSW.

OS Application specific parts of the RTE implementation are generated to separate `Rte_<OsApplicationName>.c` files.

3.6.5.3 OS Configuration extension

In addition to the RTE extensions in the OS configuration which are done automatically by the RTE generator, the following steps have to be done by the Integrator.

All OS objects, used by BSW modules e.g. ISRs, BSW-Tasks, OS resources, alarms etc. have to be assigned to an OS Application. COM callbacks have to run in the same OS Application as the RTE/BSW Scheduler tasks. Dependent on the implementation of the COM Stack, the tasks or ISRs, which call the COM callbacks must therefore be assigned to the right OS Application.

In the OS configuration of an SC3 or SC4 OS, the tasks must explicitly allow access by other OS Applications. Due to possible calls of `ActivateTask` or `SetEvent` inside the RTE implemented COM callbacks, the accessing BSW OS Applications for all application tasks, which are affected by these calls, need to be configured. This is automatically done when the RTE configuration contains all BSW Scheduler tasks. Otherwise, the configuration needs to be extended by the integrator.

If the RTE configuration contains not all BSW Scheduler tasks, also the OS Application that sets up the tasks and alarms by calling `Rte_Start or Rte_StartTiming`, needs to be configured for the task and alarm objects in the OS configuration.

This configuration extension has to be done in the OS configuration tool.

3.7 Multicore support

Similar to the mapping of SWCs to partitions with different memory access rights, the MICROSAR Classic RTE also supports the mapping of SWCs to partitions on different cores for parallel execution.

3.7.1 Partitioning of SWCs

The mapping of SWCs to cores happens with the help of OS Applications like in the memory protection use case. The user has to assign runnables to tasks and tasks to OS Applications in order to map SWCs to partitions. The OS Applications can then be assigned to one of the cores of the ECU. SWCs can only be assigned to a single OS Application. This means that all runnables of a SWC need to be mapped to tasks within the same OS Application. If a SWC contains only server runnables that are not mapped to a task, the SWC can be mapped with the help of an ECUC partition. See chapter 5.4.

When two SWCs on different cores communicate with each other, the RTE will automatically apply data consistency mechanisms.

3.7.2 BSW in Multicore Systems

The MICROSAR Classic RTE determines the NVM partition from the task mapping of NvM_MainFunction. The RTE determines the COM/LDCOM partition either from the assigned partition in the ECUC module if present or otherwise the task mapping of the Com_MainFunctionRx and Com_MainFunctionTx. The RTE determines the partitions that call the lifecycle APIs by the task mapping of EcuM_MainFunction and BswM_MainFunction.

Caution

It needs to be assured that the access rights in the OS configuration are configured so that the lifecycle APIs can start and stop the tasks and alarms.

Moreover, the memory mapping, especially for the Rte_InitState<CoreExtension> variables needs to allow write access for the lifecycle APIs and protection from partitions with lower ASIL.

This is especially true, if multiple EcuM_MainFunctions and BswM_MainFunctions are used in multiple partitions on a single core.

In case there are different partitions with calls to BswM_MainFunction on a single core, the lifecycle APIs shall be called from the partition that contains EcuM_MainFunction.

See chapter 3.7.3 for details on multicore BSW.

All RTE Lifecycle APIs (`Rte_Start()`, `Rte_Stop()`, `Rte_InitMemory()`, `SchM_Init()`, `SchM_Deinit()`) have to be called on all cores.

Cyclic triggered runnables will start to run as soon as `Rte_Start()` is called, if `Rte_StartTiming()` is not enabled, on the BSW core. If `Rte_StartTiming()` is enabled, the runnables will only start after a call to `Rte_StartTiming()` on the BSW core.

It is recommended to use only a single BSW OS Application per core. The RTE will then configure the access rights so that `Rte_Start()` can be called from the core specific BSW OS Application.

Caution

The RTE will start the scheduling of cyclic triggered runnable entities as soon as `Rte_Start()` is called on the BSW Core. Therefore, `Rte_Start()` on the BSW core should only be invoked when the `Rte_Start()` calls on all other cores finished execution. This is checked with a DET check. Moreover, initialization runnables on the other cores need to be blocked from execution until the RTE on the BSW core is finished. This can for example be done by calling `Rte_Start()` from a nonpreemptive task and by polling a variable on the BSW code that signals the termination of `Rte_Start()` on the master core. The BSW core can be configured with the EcucPartitionBswModuleExecution parameter.

3.7.3 Service BSW in Multicore Systems

The MICROSAR Classic RTE supports BSW multiple partition distribution. This requires service BSW modules which provide partition specific service SWC descriptions. The BSW main function in such a distributed system can have multiple triggers and each trigger can be mapped to a different task on a different core.

The following example shows a possible configuration for the BSW module WdgM:

Service SWC: WdgMCore0

- Runnable Entity: **WdgM_Mainfunction**
- Periodical Trigger: TriggerCore0 e.g. 5ms
- mapped to TaskCore0 in PartitionBSWCore0 on Core 0
- Service SWC implicitly mapped to Core 0

- Runnable Entity: **WdgM_CheckPointReached**
- OperationInvocation Trigger
- unmapped

Service SWC: WdgMCore1

- Runnable Entity: **WdgM_Mainfunction**
- Periodical Trigger: TriggerCore1 e.g. 1ms
- mapped to TaskCore1 in PartitionBSWCore1 on Core 1
- Service SWC implicitly mapped to Core 1

- Runnable Entity: **WdgM_CheckPointReached**
- OperationInvocation Trigger
- unmapped

Service SWC: WdgMCore1ASIL

- Service SWC explicitly mapped to PartitionCore1ASIL
  because of the missing task mapping for WdgM_Mainfunction

- Runnable Entity: **WdgM_CheckPointReached**
- OperationInvocation Trigger
- unmapped

Application SWCs can call the partition local C/S operation CheckPointReached. If the server runnables are not mapped like in the example above, the RTE can implement the `Rte_Call` API by a direct function call. The BSW function `WdgM_CheckPointReached` needs to be implemented multicore reentrant and therefore requires specific multicore support.

The `WdgM_Mainfunction` also needs to be implemented multicore reentrant because it is called periodically by the RTE from different cores.

Caution
Service BSW modules distributed on different cores requires specific multicore support of the BSW module.

3.7.4 IOC Usage

In multicore systems, the OS provides Inter OS-Application Communicator (IOC) Objects for the communication between the individual cores. However, on many systems the memory of the different cores can also be accessed without IOCs. This is the case when the RTE variables that are used for communication are mapped to non-cacheable RAM and when they can either be accessed atomically or when the accesses are protected with a spinlock. Moreover, in case of memory protection, this is only possible when a variable is only written by a single partition and when the variable can be read by the other partitions.

The MICROSAR Classic RTE Generator tries to avoid IOCs so that it can use the same variables for intra and inter partition communication. Moreover, spinlocks are only used for variables that cannot be accessed atomically.

If the RTE variables cannot be mapped to globally readable, shared, non-cacheable RAM the usage of IOCs can be enforced with the `EnforceIoc` or `EnforceQueuedIoc` parameters in the `RteGeneration` parameters. When an operating system with spinlock free IOC queues is used, the parameters therefore also remove the spinlocks for the queued sender-receiver communication inside the RTE.

Caution
RTE variables that are mapped to `NOCACHE` memory sections need to be mapped to non-cacheable RAM. See also chapter 4.5.

3.8 BSW Access in Partitioned systems

When the SWCs are assigned to different OS Applications, only the SWCs that are assigned to the BSW OS Application can access the BSW directly. SWCs that are assigned to other cores or partitions do not always have the required access rights. The same is true for runnable entities that are directly called by the BSW through client/server interfaces. The RTE can transparently provide proxy code for such BSW accesses but the user needs to map the SendSignal proxy and the server runnables to tasks in which they can be executed.

3.8.1 Inter-ECU Communication

IOCs or additional global RTE variables are automatically inserted by the RTE generator when data needs to be sent from a partition without BSW to another ECU. This is required because the COM APIs cannot be called directly in this case.

Instead, the RTE provides the schedulable entitiy `Rte_ComSendSignalProxyPeriodic` for the partition that contains the BSW COM module, which periodically calls the COM APIs when a partition without BSW transmitted data.

If the COM is instantiated on multiple partitions for parallel use, then such a schedulable entity `Rte_ComSendSignalProxyPeriodic_<PartitionName>` is provided on each partition that contains the COM module. In this case the RteOptimizationMode attribute must be set to RUNTIME as the optimization mode MEMORY would require spinlocks for all bitfield accesses, which would cause too much overhead. In case multiple `Com_MainFunctionTxs` are used for the com signals and signal groups, correspoding schedulable entities `Rte_ComMainFunctionTxProxyPeriodic_<NameOfTheComMainFunctionTx>_<PartitionName>` are provided by the RTE. However, if only one `Com_MainFunctionTx` is configured, the provided schedulable entity will still be named `Rte_ComSendSignalProxyPeriodic`.

Each schedulable entity `Rte_ComSendSignalProxyPeriodic` should be mapped to the same task as the corresponding `Com_MainFunctionTx` with a lower position in task so that it can update the signals before they are transmitted by the COM. `Rte_ComSendSignalProxyPeriodic` will be scheduled with the same cycle time as `Com_MainFunctionTx`. For this, the RTE generator reads the period from the COM configuration. For the reception of signals no special handling is required. The RTE will automatically forward the received data to the appropriate partition in the COM notifications.

Caution
The FanIn / FanOut communication is currently not supported when the BSW COM module is mapped to multiple non trusted partitions.

3.8.2 Client Server Communication

Client server calls between SWCs in different partitions are also possible.

In order to execute the server runnable in another partition, the server runnable needs to be mapped to a task. The RTE will then make the server arguments available in the partition of the server runnable, execute the server runnable in the context of its task and return the results to the calling partition.

Direct client server calls to servers on other cores are not possible because this would enforce that the server is executed in the context of the client core. This would lead to data consistency problems for RTE APIs that only provide buffer pointers like `Rte_Pim()`. The RTE cannot use IOCs for these APIs because the actual buffer update is done by the application code.

You can instruct the RTE to generate a context switch. You can decide this over the task mapping of a function trigger.

If you consider RTE calls which originate from the same partition as the server runnable, a context switch into the task of the server runnable may not be required. Performing a task switch would mean an additional overhead which can be avoided.

Therefore, it is also possible to configure an additional server port prototype for clients which are local to the server partition. The triggers from both server ports can then trigger the same server runnable. However, only the trigger from the port that is connected to foreign partitions needs to be mapped onto a task. As a consequence, the RTE can implement calls from partition local clients as efficient direct function calls.

Please take into account, that this is only allowed when the server runnable is not invoked concurrently or marked as “can be invoked concurrently”. In addition, you can use Exclusive Areas to protect the runnable against concurrent access problems.

3.9 Custom MemCpy Implementations

It is possible to deactivate the compilation of Rte_MemCpy32, Rte_MemClr, Rte_MemCpy and Rte_PartialMemCpy with a preprocessor switch. This allows to manually implement the APIs or directly map them to VStdLib APIs if the signature is identical.

The deactivation is enabled when the following preprocessor defines are specified as compiler flags: RTE_ENABLE_CUSTOM_MEMCPY32, RTE_ENABLE_CUSTOM_MEMCPY, RTE_ENABLE_CUSTOM_MEMCLR, RTE_ENABLE_CUSTOM_PARTIALMEMCPY.

4 API Description

The RTE API functions used inside the runnable entities are accessible by including the SWC application header file `Rte_<ComponentType>.h`.

The names of most RTE APIs depend on the configuration. The RTE generator therefore replaces placeholders that are marked with angle brackets with the actual object names.

Abbreviation	Description
<cts>	ComponentTypeSymbol
<r>	RunnableEntity
<ap>	AccessPoint
<i>	PortInterface
<p>	PortPrototype
<o>	OperationPrototype
<d>	DataElementPrototype
<cp>	CalibrationParameter
<m>	ModeGroupPrototype
<v>	InterRunnableVariable
<n>	PerInstanceMemory
<idt>	ImplementationDataType
<idte>	ImplementationDataTypeElement
<pi>	RipsPluginName
<exe>	ExecutableEntity
<cgi>	CommunicationGraphInstance
<pa>	EcucPartition
<mmi>	ModeMachineInstance
<t>	TaskName
<ctp>	ComponentTypePrototype

Table 4-1 Abbreviations for API Name Placeholders

Info
The following API descriptions contain the direction qualifier IN, OUT and INOUT. They are intended as direction information only and shall not be used inside the application code.

Depending on the configuration, some APIs are efficiently implemented as function-like macros. This implementation introduces restrictions on how the APIs can be used in the software-component. E.g. it is not possible to take the address of a macro in C.

The macro implementation may also lead to unwanted compiler optimizations regarding concurrent accesses of variables. If a variable is accessed multiple times (e.g. by calling the Rte_Read API in a loop), the compiler may not be aware that the value of the variable may change at any time and optimize away the subsequent accesses.

Info
If it is not possible for the implementation of a software-component to use a function-like macro of a port API, the Port API Option `enableTakeAddress` can be used to force the generation of a “C” function. The option however cannot be used in combination with the indirect API.

For an interface overview please see Figure 1-2.

4.1 Data Type Definition

The MICROSAR Classic RTE has special handling for the implementation data types, which are defined in `Std_Types.h` and `Platform_Types.h` (see [7] and [8] for details). The RTE generator assumes that these data types are available and therefore skips the generation of type definitions.

In addition implementation data types where the `typeEmitter` attribute is set to a value different to `RTE` or where the value is not empty the RTE generator also skips generation of the type definition. In this case the user has to adopt the generated template file `Rte_UserTypes.h` which should contain the type definitions for the skipt implementation data types because the RTE itself needs the type definition.

All other primitive or composite application and implementation data types are generated by the RTE generator. This includes the data types which are assigned to a SWC by a definition of an `IncludedDataTypeSet`.

4.1.1 Invalid Value

The MICROSAR Classic RTE provides access to the invalid value of a primitive data type. It can be accessed with the following macro:

```
InvalidValue_<literalPrefix><DataType>
```

Caution
Because the macro does not contain the `Rte_` prefix, care must be taken not to define data types conflicting with any other symbols defined by the RTE or the application code. The optional literalPrefix can be used to resolve conflicts.

4.1.2 Upper and Lower Limit

The range of the primitive application data types is specified by an upper and a lower limit. These limits are accessible from the application by using the following macro if the limits are specified:

```
<literalPrefix><DataType>_LowerLimit
<literalPrefix><DataType>_UpperLimit
```

Caution
Because the macro does not contain the `Rte_` prefix, care must be taken not to define data types conflicting with any other symbols defined by the RTE or the application code. The optional literalPrefix can be used to resolve conflicts.

4.1.3 Initial Value

Like the limits also the initial value of an un-queued data element of an S/R port prototype is accessible from the application:

`Rte_InitValue_<PortPrototype>_<DataElementPrototype>`

Caution
The initial value of an Inter-Ecu S/R communication might be changed by the post-build capabilities of the communication stack. Please note that the macro of the RTE still provides the original initial value defined at pre-compile time. Please don't use the macro if the initial value will be changed in the communication stack at post-build time.

4.1.4 Optional Struct Member

The RTE generator provides macros for each optional struct member for accessing or setting the availability information.

`Rte_IsAvailable_<idt>_<idte>(data)`

`Rte_SetAvailable_<idt>_<idte>(data, available)`

The macros can be used inside the runnable entities.

The `Rte_IsAvailable` API takes a concrete variable as input by reference (e.g. the returned data of `Rte_Read`). The variable must be of type `<ImplementationDataType>`. The return value is used to indicate whether the optional member `<ImplementationDataTypeElement>` is available within the variable of type `<ImplementationDataType>`.

The `Rte_SetAvailable` API takes a concrete variable as input by reference (e.g. a variable which will be passed to an `Rte_Write` call). The variable must be of type `<ImplementationDataType>`. The API sets the availability of the struct member `<ImplementationDataTypeElement>` within the variable to the value defined by the `available` parameter.

4.2 API Error Status

Most of the RTE APIs provide an error status in the API return code. For easier evaluation, the MICROSAR Classic RTE provides the following status access macros:

```
Rte_IsInfrastructureError(status)
Rte_HasOverlayedError(status)
Rte_ApplicationError(status)
```

The macros can be used inside the runnable entities for evaluation of the RTE API return code. The `boolean` return code of the `Rte_IsInfrastructure` and `Rte_HasOverlayedError` macros indicate if either the immediate infrastructure error flag (bit 7) or the overlay error flag (bit 6) is set.

The `Rte_ApplicationError` macro returns the application errors without overlayed errors.

4.3 Runnable Entities

Runnable entities are configured in DaVinci and must be implemented by the user. DaVinci features the generation of a template file containing the empty bodies of all runnable entities that are configured for a specific component type.

4.3.1 <RunnableEntity>

Prototype	
`void <RunnableEntity> ( [IN Rte_Instance instance][, IN Rte_ActivatingEvent_<RunnableEntity> activation])`	
`{Std_ReturnType|void} <ServerRunnable> ( [IN Rte_Instance instance] {, IN type [*]inputparam}* {, OUT type *outputparam}* )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
activation	The activation parameter can be used to get the actual activation reason of the runnable entity if the runnable has multiple possible trigger conditions (e.g. different cyclic triggers or a cyclic trigger and a data reception trigger). Note: This is an optional parameter depending on the configuration of an activation reason at the runnable entity. It is only reasonable if more than one runnable trigger (RTE Event) is configured.
[*]inputparam, *outputparam	Parameters are only present for server runnables, i.e. runnable entities triggered by an OperationInvokedEvent. Input (IN) parameters are passed by value (primitive types) or reference (composite and string types), output (OUT) parameters are always passed by reference.
Return code	
RTE_E_OK	Server runnables return RTE_E_OK for successful operation execution if an application error is referenced by the operation prototype. Application errors are defined at the client/server port interface.
RTE_E_<interf>_<error>	Server runnables may return an application error (in the range of 1 to 63) if the operation execution was not successful. Application errors are defined at the client/server port interface and are referenced by the operation prototype.
Existence	
If configured in DaVinci.	
Functional Description	
The user function `<RunnableEntity>()` is the specific implementation of a runnable entity of a software component and has to be provided by the user. It is called from the MICROSAR Classic RTE. The first prototype form with no return value and the optional instance parameter is valid for the following trigger conditions: ▶ TimingEvent: Triggered on expiration of a configured timer.	

- DataReceivedEvent: Triggered on reception of a data element.
- DataReceiveErrorEvent: Triggered on reception error of a data element.
- DataSendCompletionEvent: Triggered on successful transmission of a data element.
- ModeSwitchEvent: Triggered on entering or exiting of a mode of a mode declaration group.
- ModeSwitchedAckEvent: Triggered on completion of a mode switch of a mode declaration group.
- AsynchronousServerCallReturnsEvent: Triggered on finishing of an asynchronous server call. The `Rte_Result()` API shall be used to get the out parameters of the server call.
- InitEvent: Triggered on startup of the RTE.
- BackgroundEvent: Triggered by the RTE in an endless loop – in the background – when no other runnable runs.
- ExternalTriggerOccurredEvent
- InternalTriggerOccurredEvent: Triggered when the referenced internal trigger has occurred.

The optional activation parameter is valid for all above described trigger conditions except for the InitEvent.

The second prototype form is valid for server runnables:

- OperationInvokedEvent: Triggered on invocation of the operation in a C/S port interface (server runnable). A return value is only present if application errors are referenced by the implemented operation. The parameter list is directly derived from the configured operation prototype with the addition of the optional instance parameter.

The configuration of the trigger conditions can be done in the runnable entities tab of the component type configuration.

Call Context

The call context of server runnables depends on the task mapping. Server runnables mapped to a task are executed in the context of this task, unmapped server runnables are executed in the context of the task that invoked the operation. All other runnables are invoked by the RTE in the context of the task the runnables are mapped to.

Caution
The relative priority of the assigned OS tasks is responsible for the call sequence of Init-Runnables. The RTE ensures that the Init-Runnable is called before any other runnable is mapped to the same task but does not enforce that all Init-Runnables have been executed before any other runnable is called. To make sure that all Init-Runnables are executed before any other runnable is called, all Init-Runnables should be mapped to the task with the highest priority.

4.3.2 Runnable Activation Reason

If the activation reason is configured the actual reason can be evaluated with the following generated define

`Rte_ActivatingEvent_<RunnabaleEntity>_<Reason>`

where `_<RunnabaleEntity>` is the symbol attribute of the Runnable and `<Reason>` is the symbolic name of activation reason. The return type of the macro depends on the highest configured bit position for all trigger conditions of a runnable entity. It is `uint8`, `uint16` or `unit32`.

Caution
The activation reason is not supported if alive timeout monitoring is enabled for multiple ports of a SW-C for the same signal.

4.4 SWC Exclusive Areas

4.4.1 Rte_Enter

Prototype	
`void` **`Rte_Enter_<ExclusiveArea>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
–	
Existence	
This API exists when at least one runnable has configured explicit access (`canEnterExclusiveArea`) to an exclusive area of a component.	
Functional Description	
The function `Rte_Enter_<ea>()` implements explicit access to the exclusive area. The exclusive area is defined in the context of a component type and may be accessed by all runnables of that component, either implicitly or explicitly via this API. This function is the counterpart of `Rte_Exit_<ea>()`. Each call to `Rte_Enter_<ea>()` must be matched by a call to `Rte_Exit_<ea>()` in the same runnable entity. One exclusive area must not be entered more than once at a time, but different exclusive areas may be nested, as long as they are left in reverse order of entering them. For restrictions on using exclusive areas with different implementations, see section 2.6.10.	
Call Context	
This function can be used inside runnable entities. This function must not be called with locked interrupts.	

4.4.2 Rte_Exit

Prototype	
`void` **`Rte_Exit_<ExclusiveArea>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
-	
Existence	
This API exists when at least one runnable has configured explicit access (`canEnterExclusiveArea`) to an exclusive area of a component.	
Functional Description	
The function `Rte_Exit_<ea>()` implements releasing of an explicit entered exclusive area. The exclusive area is defined in the context of a component type and may be accessed by all runnables of that component, either implicitly or explicitly via this API. This function is the counterpart of `Rte_Enter_<ea>()`. Each call to `Rte_Enter_<ea>()` must be matched by a call to `Rte_Exit_<ea>()` in the same runnable entity. One exclusive area must not be entered more than once at a time, but different exclusive areas may be nested, as long as they are left in reverse order of entering them. For restrictions on using exclusive areas with different implementations, see section 2.6.10.	
Call Context	
This function can be used inside runnable entities. This function must not be called with locked interrupts.	

4.5 BSW Exclusive Areas

4.5.1 SchM_Enter

Prototype
`void` **`SchM_Enter_<Bsw>_<ExclusiveArea>`** `( void )`
Parameter
–
Return code
–
Existence
This API exists when at least one schedulable entity has configured access (`canEnterExclusiveArea`) to an exclusive area in the internal behavior of the BSW module description.
Functional Description
The function `SchM_Enter_<bsw>_<ea>()` implements access to the exclusive area. The exclusive area is defined in the context of a BSW module and may be accessed by all schedulable entities of that module via this API. This function is the counterpart of `SchM_Exit_<bsw>_<ea>()`. Each call to `SchM_Enter_<bsw>_<ea>()` must be matched by a call to `SchM_Exit_<bsw>_<ea>()` in the same schedulable entity. One exclusive area must not be entered more than once at a time, but different exclusive areas may be nested, as long as they are left in reverse order of entering them. For restrictions on using exclusive areas with different implementation methods, see section 2.6.10.
Call Context
This function can be used inside a schedulable entity in Task or Interrupt context.

4.5.2 SchM_Exit

Prototype	
`void` **`SchM_Exit_<Bsw>_<ExclusiveArea>`** `( void )`	
Parameter	
–	
Return code	
–	
Existence	
This API exists when at least one schedulable entity has configured access (`canEnterExclusiveArea`) to an exclusive area in the internal behavior of the BSW module description.	
Functional Description	
The function `SchM_Exit_<bsw>_<ea>()` implements releasing of the exclusive area. The exclusive area is defined in the context of a BSW module and may be accessed by all schedulable entities of that module via this API. This function is the counterpart of `SchM_Enter_<bsw>_<ea>()`. Each call to `SchM_Enter_<bsw>_<ea>()` must be matched by a call to `SchM_Exit_<bsw>_<ea>()` in the same schedulable entity. One exclusive area must not be entered more than once at a time, but different exclusive areas may be nested, as long as they are left in reverse order of entering them. For restrictions on using exclusive areas with different implementation methods, see section 2.6.10.	
Call Context	
This function can be used inside a schedulable entity in Task or Interrupt context.	

4.6 Sender-Receiver Communication

4.6.1 Rte_Read

Prototype	
`Std_ReturnType` **`Rte_Read_<p>_<d>`** `( [IN Rte_Instance instance,] INOUT <DataType> *data [, OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
*data	The output <data> is passed by reference. The <DataType> is the type, specified at the data element prototype in the SWC description.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Data read successfully.
RTE_E_UNCONNECTED	Indicates that the receiver port is not connected.
RTE_E_COM_STOPPED	An infrastructure communication error was detected by the RTE. The error is not reported when the COM runs in a different partition.
RTE_E_INVALID	An invalidated signal has been received by the RTE.
RTE_E_MAX_AGE_EXCEEDED	Indicates a timeout, detected by the COM module in case of inter ECU communication, if an `aliveTimeout` is specified.
RTE_E_NEVER_RECEIVED	No data received since system start.
RTE_E_COM_BUSY	The read request is rejected due to a currently ongoing reception. No received data can be provided.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) access in the role `dataReceivePointByArgument` for the data element in the DaVinci configuration and the referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The function `Rte_Read_<p>_<d>()` supplies the current value of the data element. This API can be used for explicit read of S/R data with `isQueued=false`. After startup `Rte_Read` provides the initial value.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

Caution
If group signals are used in combination with data transformation, the implementation may violate [SWS_Rte_06830], if the I-PDU is stopped after the successful reception of the group signal. In this case Rte_Read might return a value different from the last known value (or init value).

Caution
According to the RTE specification, RTE_E_MAX_AGE_EXCEEDED cannot occur when the I-PDU is stopped (RTE_E_COM_STOPPED). However, if a timeout occurred before the I-PDU is stopped, RTE_E_MAX_AGE_EXCEEDED is still reported in addition to RTE_E_COM_STOPPED as the COM does not call the notification callback that clears the status in this scenario.

Caution
In case a hard transformer error occurs, the API might return an inconsistent value.

Caution
The COM specification specifies that the initial value is set on restart of an IPDU group. However, COM does not call the communication callbacks from the RTE during restart. When the signal is received in the communication callbacks, e.g. for queued communication, fan-in or in case of inter-ECU communication, the last value is returned by the RTE APIs instead of the initial value.

Caution
For each ImplementationDataTypeElement of data where an arrayImplPolicy is set to payloadAsPointerToArray, the user has to allocate an array of sufficient size and set the respective pointer to it. In that case, data is treated as an INOUT parameter, otherwise its direction is OUT.

4.6.2 Rte_DRead

Prototype	
`<DataType> Rte_DRead_<p>_<d> ( [IN Rte_Instance instance][, OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
<DataType>	The return value contains the current value of the data element. The <DataType> is the (primitive) type, specified at the data element prototype in the SWC description.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) access in the role `dataReceivePointByValue` for the data element in the DaVinci configuration and the referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The function `Rte_DRead_<p>_<d>()` supplies the current value of the data element. This API can be used for explicit read of S/R data with `isQueued=false`. After startup or if the receiver port is unconnected, `Rte_DRead` provides the initial value. The API is only available for primitive data types.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

Caution
In case a hard transformer error occurs, the API might return an inconsistent value.

4.6.3 Rte_Write

Prototype	
`Std_ReturnType` **`Rte_Write_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> data [, OUT Std_TransformerError transformerError] )`	
`Std_ReturnType` **`Rte_Write_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> *data [, OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
data	The input data <data> for primitive data types without string types is passed by value. The <DataType> is the type, specified at the data element prototype in the SWC description.
*data	The input data <data> for string types and composite data types is passed by reference. The <DataType> is the type, specified at the data element prototype in the SWC description.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Data passed to communication services successfully.
RTE_E_COM_STOPPED	An infrastructure communication error was detected by the RTE. The error is not reported when the COM runs in a different partition.
RTE_E_COM_BUSY	The write request is rejected due to a currently ongoing transmission.
RTE_E_TIMEOUT	The write request is rejected because the TP buffer is locked.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) access to the data element in the DaVinci configuration and the referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The function `Rte_Write_<p>_<d>()` can be used for explicit transmission of S/R data with `isQueued=false`.	

Call Context
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.

Caution
If the length returned by a transformer exceeds the send buffer size and the transformer itself does not return with a hard transformer error, Rte_Write returns RTE_E_HARD_TRANSFORMER_ERROR without setting the transformerError parameter.

4.6.4 Rte_Receive

Prototype	
`Std_ReturnType` **`Rte_Receive_<p>_<d>`** `( [IN Rte_Instance instance,] OUT <DataType> *data [, OUT uint16 *length][, OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
*data	The output <data> is passed by reference. The <DataType> is the type, specified at the data element prototype in the SWC description.
*length	In case of an array with variable number of elements, the dynamic length <length> is returned. Note: The parameter is only applicable for dynamic-size arrays defined according to Chapter 2.8.1.1 [11].
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Data read successfully.
RTE_E_UNCONNECTED	Indicates that the receiver port is not connected.
RTE_E_NO_DATA	A non-blocking call returned no data due to an empty receive queue. No other error occurred.
RTE_E_TIMEOUT	Returned by a blocking call after the timeout has expired. No data returned and no other error occurred. The argument buffer is not changed.
RTE_E_LOST_DATA	Indicates that some incoming data has been lost due to an overflow of the receive queue. This is not an error of the data returned in the out parameter.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured polling or waiting access to the data element in the DaVinci configuration and the referenced data element prototype is configured with queued communication (`isQueued=true`).	
Functional Description	
The function `Rte_Receive_<p>_<d>()` supplies the oldest value stored in the reception queue of the data element. This API can be used for explicit read of S/R data with `isQueued=true`.	

Call Context
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.

Caution
The IOCs in MICROSAR Classic OS can return IOC_E_NOK (1) in case of incorrect usage of IOCs (e.g. when an RTE API and therefore also the IOC API is called from the wrong partition). This error code is then returned to the application, although IOC_E_NOK is not any of the return values specified above.

4.6.5 Rte_Send

Prototype	
`Std_ReturnType` **`Rte_Send_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> data [, OUT Std_TransformerError transformerError] )`	
`Std_ReturnType` **`Rte_Send_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> *data [, IN uint16 length] [, OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
data	The input data <data> for primitive data types without string types is passed by value. The <DataType> is the type, specified at the data element prototype in the SWC description.
*data	The input data <data> for string types and composite data types is passed by reference. The <DataType> is the type, specified at the data element prototype in the SWC description.
length	In case of an array with variable number of elements, the input data <length> specifies the dynamic array length. Note: The parameter is only applicable for dynamic-size arrays defined according to Chapter 2.8.1.1 [11].
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Data passed to communication services successfully.
RTE_E_COM_STOPPED	An infrastructure communication error was detected by the RTE. The error is not reported when the COM runs in a different partition.
RTE_E_INVALID	An invalid length was passed to an array with variable number of elements.
RTE_E_TIMEOUT	The write request is rejected because the TP buffer is locked.
RTE_E_LIMIT	The submitted data has been discarded because the receiver queue is full. Relevant only to intra ECU communication.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the data element in the DaVinci configuration and the referenced data element prototype is configured with queued communication (`isQueued=true`).	

Functional Description
The function `Rte_Send_<p>_<d>()` can be used for explicit transmission of S/R data with `isQueued=true`.
Call Context
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.

Caution
If the length returned by a transformer exceeds the send buffer size and the transformer itself does not return with a hard transformer error, Rte_Send returns RTE_E_HARD_TRANSFORMER_ERROR without setting the transformerError parameter.

Caution
The IOCs in MICROSAR Classic OS can return IOC_E_NOK (1) in case of incorrect usage of IOCs (e.g. when an RTE API and therefore also the IOC API is called from the wrong partition). This error code is then returned to the application, although IOC_E_NOK is not any of the return values specified above.

4.6.6 Rte_IRead

Prototype	
`<DataType> **Rte_IRead_<r>_<p>_<d>** ( [IN Rte_Instance instance] )`	
`<DataType> ***Rte_IRead_<r>_<p>_<d>** ( [IN Rte_Instance instance] )`	
Parameter	
Instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<DataType>	The return value contains the buffered data for primitive data types. <DataType> is the type, specified at the data element prototype in the SWC description
<DataType> *	The return value contains a reference to the buffered data for string types and composite data types. <DataType> is the type, specified at the data element prototype in the SWC description
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration.	
Functional Description	
The function `Rte_IRead_<r>_<p>_<d>()` supplies the value of the data element, stored in a buffer before starting of the runnable entity. This API can be used for buffered (implicit) read of S/R data with `isQueued=false`. After startup `Rte_IRead` provides the initial value.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

Caution
In case a hard transformer error occurs, the API might return an inconsistent value.

4.6.7 Rte_IWrite

Prototype	
`void Rte_IWrite_<r>_<p>_<d> ( [IN Rte_Instance instance,] IN <DataType> data )`	
`void Rte_IWrite_<r>_<p>_<d> ( [IN Rte_Instance instance,] IN <DataType> *data )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
data	The input data <data> for primitive data types without string types is passed by value. The <DataType> is the type, specified at the data element prototype in the SWC description.
*data	The input data <data> for string types and composite data types is passed by reference. The <DataType> is the type, specified at the data element prototype in the SWC description.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration.	
Functional Description	
The function `Rte_IWrite_<r>_<p>_<d>()` can be used for buffered transmission of S/R data with `isQueued=false`. Note, that the actual transmission is performed and therefore visible for other runnable entities after the runnable entity has been terminated.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

Caution
When implicit write access to a data element has been configured for a runnable, the runnable has to update the data element at least once during its execution time using the `Rte_IWrite` API or writing to the location returned by the `Rte_IWriteRef` API. Otherwise, the content of the data element is undefined upon return from the runnable. Only when the parameter RteInitializeImplicitBuffers is set to true, the RTE will send the last sent data again when `Rte_IWrite` or `Rte_IWriteRef` are not called in the runnable.

4.6.8 Rte_IWriteRef

Prototype	
`<DataType> *Rte_IWriteRef_<r>_<p>_<d> ( [IN Rte_Instance instance] )`	
Parameter	
Instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<DataType> *	The return value contains a reference to the buffered data. <DataType> is the type, specified at the data element prototype in the SWC description
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration.	
Functional Description	
The function `Rte_IWriteRef_<r>_<p>_<d>()` can be used for buffered transmission of S/R data with `isQueued=false`. Note, that the actual transmission is performed and therefore visible for other runnable entities after the runnable entity has been terminated. The returned reference can be used by the runnable entity to directly update the corresponding data elements. This is especially useful for data elements of composite types.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

Caution
When implicit write access to a data element has been configured for a runnable, the runnable has to update the data element at least once during its execution time using the `Rte_IWrite` API or writing to the location returned by the `Rte_IWriteRef` API. Otherwise, the content of the data element is undefined upon return from the runnable.

4.6.9 Rte_IStatus

Prototype	
`Std_ReturnType` **`Rte_IStatus_<r>_<p>_<d>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
RTE_E_OK	Data read successfully.
RTE_E_UNCONNECTED	Indicates that the receiver port is not connected.
RTE_E_INVALID	An invalidated signal has been received by the RTE.
RTE_E_MAX_AGE_EXCEEDED	Indicates a timeout, detected by the COM module in case of inter ECU communication, if an `aliveTimeout` is specified.
RTE_E_NEVER_RECEIVED	No data received since system start.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration and if either ▶ data element outdated notification (`aliveTimeout > 0`) or ▶ data element invalidation is activated for this data element or ▶ the attribute `handleNeverReceived` is configured.	
Functional Description	
The function `Rte_IStatus_<r>_<p>_<d>()` returns the status of the data element which can be read with `Rte_IRead`.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

4.6.10 Rte_Feedback

Prototype	
`Std_ReturnType` **`Rte_Feedback_<p>_<d>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
RTE_E_NO_DATA	No data transmitted, when the feedback API was attempted (non-blocking call only).
RTE_E_UNCONNECTED	Indicates that the sender port is not connected.
RTE_E_TIMEOUT	A timeout notification was received from the COM before any error notification (Inter-ECU only).
RTE_E_COM_STOPPED	The last transmission was rejected when either Rte_Send / Rte_Write API was called and the COM was stopped or an error notification from the COM was received before any timeout notification (Inter-ECU only).
RTE_E_TRANSMIT_ACK	A “transmission acknowledgement” has been received.
Existence	
This API exists, if the runnable entity of a SWC has configured explicit access to the data element in the DaVinci configuration of a runnable entity and in addition the transmission acknowledgement is enabled at the communication specification. Furthermore, polling or waiting acknowledgment mode has to be specified for the same data element. If a timeout is specified, timeout monitoring for waiting acknowledgment access is enabled.	
Functional Description	
The function `Rte_Feedback_<p>_<d>()` can be used to read the transmission status for explicit S/R communication. It indicates the status of data, transmitted by `Rte_Write()` and `Rte_Send()` calls. Depending on the configuration, the API can be either blocking or non-blocking.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.6.11 Rte_IsUpdated

Prototype	
`boolean` **`Rte_IsUpdated_<p>_<d>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
TRUE	Data element has been updated since last read.
FALSE	Data element has not been updated since last read.
Existence	
This API exists, if the runnable entity of a SWC has configured explicit access to the data element in the DaVinci configuration of a runnable entity and in addition the `EnableUpdate` attribute is enabled at the communication specification.	
Functional Description	
The function `Rte_IsUpdated_<p>_<d>()` returns if the data element has been updated since the last read or not.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.6.12 Rte_Invalidate

Prototype	
`Std_ReturnType` **`Rte_Invalidate_<p>_<d>`** `( [IN Rte_Instance instance] [, OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	No error occurred.
RTE_E_COM_STOPPED	The RTE could not perform the operation because the COM service is currently not available (inter ECU communication only). The error is not reported when the COM runs in a different partition.
Existence	
This API exists, if the runnable entity of a SWC has configured explicit and non-queued access to the data element in the DaVinci configuration of a runnable entity and in addition the data element invalidation is enabled at the communication specification (`CanInvalidate=true`).	
Functional Description	
The function `Rte_Invalidate_<p>_<d>()` can be used to set the transmission data invalid for explicit non-queued S/R communication.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.6.13 Rte_IInvalidate

Prototype	
`void` **`Rte_IInvalidate_<r>_<p>_<d>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration of a runnable entity and in addition the data element invalidation is enabled at the communication specification (`CanInvalidate=true`).	
Functional Description	
The function `Rte_IInvalidate_<r>_<p>_<d>()` can be used to set the transmission data invalid for implicit (buffered) S/R communication.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

4.6.14 Rte_IFeedback

Prototype	
`Std_ReturnType` **`Rte_IFeedback_<r>_<p>_<d>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
RTE_E_TIMEOUT	A timeout notification was received from the COM before any error notification (Inter-ECU only).
RTE_E_COM_STOPPED	The last transmission was rejected when either Rte_IWrite API was called and the COM was stopped or an error notification from the COM was received before any timeout notification (Inter-ECU only).
RTE_E_TRANSMIT_ACK	A “transmission acknowledgement” has been received.
RTE_E_NO_DATA	No data transmitted, when the ifeedback API was attempted (non-blocking call only).
RTE_E_UNCONNECTED	Indicates that the sender port is not connected.
Existence	
This API exists, if the runnable entity of a SWC has configured (implicit) access to the data element in the DaVinci configuration and if either ▶ data element outdated notification (`aliveTimeout > 0`) or ▶ data element invalidation is activated for this data element or ▶ the attribute `handleNeverReceived` is configured.	
Functional Description	
This API exists, if the runnable entity of a SWC has configured implicit and non-queued access to the data element in the DaVinci configuration of a runnable entity and in addition the data element Tx Acknowledge is activated.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

4.7 External and Internal Trigger Communication

4.7.1 Rte_Trigger

Prototype	
`void` **`Rte_Trigger_<p>_<o>`** `( [IN Rte_Instance instance,] [ OUT Std_TransformerError transformerError] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured access to the trigger element in the DaVinci configuration and the referenced trigger element prototype is configured without queuing support (`isQueued=false`).	
Functional Description	
The function `Rte_Trigger_<p>_<o>()` can be used for direct triggering of runnable entities with `isQueued=false`.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.7.2 Rte_IrTrigger

Prototype	
`[void | Std_ReturnType] Rte_IrTrigger_<re>_<o> ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
-	None in case of signature without queuing support. Note: Std_ReturnType is generated if the SwImplPolicy of the associated InternalTriggeringPoint is set to queued.
RTE_E_OK	Trigger was successfully queued or if no queue is configured
RTE_E_LIMIT	Trigger was not queued because the maximum queue size is already reached
Existence	
This API exists, if the runnable entity of a SWC has configured an internal triggering point.	
Functional Description	
The function `Rte_IrTrigger_<re>_<o>()` can be used for direct triggering of runnable entities of the same software-component instance.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.8 Mode Management

4.8.1 Rte_Switch

Prototype	
`Std_ReturnType` **`Rte_Switch_<p>_<m>`** `( [IN Rte_Instance instance,] IN Rte_ModeType_<ModeDeclarationGroup> mode )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
mode	The next mode. It is of type `Rte_ModeType_<m>`, where <m> is the name of the mode declaration group.
Return code	
RTE_E_OK	Mode switch trigger passed to the RTE successfully.
RTE_E_LIMIT	The submitted mode switch has been discarded because the mode queue is full.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the mode declaration group prototype in the DaVinci configuration.	
Functional Description	
The function `Rte_Switch_<p>_<m>()` can be used to trigger a mode switch of the specified mode declaration group prototype.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.8.2 Rte_Mode

Prototype	
`Rte_ModeType_<ModeDeclarationGroup> Rte_Mode_<p>_<m> ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
RTE_TRANSITION_<mg>	This return code is returned if the mode machine is in a mode transition.
RTE_MODE_<mg>_<m>	This value is returned if the mode machine is not in a transition. <m> indicates the currently active mode.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the mode declaration group prototype in the DaVinci configuration and the enhanced Mode API is not active.	
Functional Description	
The function `Rte_Mode_<p>_<m>()` provides the current mode of a mode declaration group prototype.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.8.3 Enhanced Rte_Mode

Prototype	
`Rte_ModeType_<ModeDeclarationGroup> Rte_Mode_<p>_<m> ( [IN Rte_Instance instance], OUT Rte_ModeType_<ModeDeclarationGroup> previousMode, OUT Rte_ModeType_<ModeDeclarationGroup> nextMode )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
`previousMode`	The previous mode is returned if the mode machine is in a transition.
`nextMode`	The next mode is returned if the mode machine is in a transition.
Return code	
RTE_TRANSITION_<mg>	This return code is returned if the mode machine is in a mode transition.
RTE_MODE_<mg>_<m>	This value is returned if the mode machine is not in a transition. <m> indicates the currently active mode.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the mode declaration group prototype in the DaVinci configuration and the enhanced Mode API is active.	
Functional Description	
The function `Rte_Mode_<p>_<m>()` provides the current mode of a mode declaration group prototype. In addition, it provodes the previous mode and the next mode if the mode machine is in transition. When the mode machine is not in transition, previousMode, nextMode and the return value return the current mode.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.8.4 Rte_SwitchAck

Prototype	
`Std_ReturnType` **`Rte_SwitchAck_<p>_<m>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
RTE_E_NO_DATA	No mode switch triggered, when the switch ack API was attempted (non-blocking call only).
RTE_E_TIMEOUT	No mode switch processed within the specified timeout time, when the switch ack API was attempted (blocking call only).
RTE_E_TRANSMIT_ACK	The mode switch acknowledgement has been received.
RTE_E_UNCONNECTED	Indicates that the mode provide port is not connected.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the mode declaration group prototype in the DaVinci configuration of a runnable entity and in addition the mode switch acknowledgement is enabled at the mode switch communication specification. Furthermore, polling or waiting acknowledgment mode has to be specified for the same mode declaration group prototype. If a timeout is specified, timeout monitoring for waiting acknowledgment access is enabled.	
Functional Description	
The function `Rte_SwitchAck_<p>_<m>()` can be used to read the mode switch status of a specific mode declaration group prototype. It indicates the status of a mode switch, triggered by a `Rte_Switch` call. Depending on the configuration, the API can be either blocking or non-blocking.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.9 Inter-Runnable Variables

4.9.1 Rte_IrvRead

Prototype	
`<DataType> **Rte_IrvRead_<r>_<v>** ( [IN Rte_Instance instance] )`	
`void **Rte_IrvRead_<r>_<v>** ([IN Rte_Instance instance,]  OUT <DataType> *data)`	
Parameter	
Instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
*data	The output <data> is passed by reference for composite data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
Return code	
<DataType>	The return value contains the current content of the Inter-Runnable Variable of primitive data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) read access to the Inter-Runnable Variable in the SWC configuration.	
Functional Description	
The function `Rte_IrvRead_<r>_<v>()` supplies the current value of the Inter-Runnable Variable. This API is used to read direct (explicit) Inter-Runnable Variables. After startup `Rte_IrvRead` provides the configured initial value.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

4.9.2 Rte_IrvWrite

Prototype	
`void` **`Rte_IrvWrite_<r>_<v>`** `( [IN Rte_Instance instance,] IN <DataType> data )`	
`void` **`Rte_IrvWrite_<r>_<v>`** `( [IN Rte_Instance instance,] IN <DataType> *data )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
data	The input data <data> is passed by value for primitive data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
*data	The input data <data> for composite data types is passed by reference. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) write access to the Inter-Runnable Variable in the SWC configuration.	
Functional Description	
The function `Rte_IrvIWrite_<r>_<v>()` can be used for updating direct (explicit) access Inter-Runnable Variables. The update is performed immediately.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

4.9.3 Rte_IrvIRead

Prototype	
`<DataType> **Rte_IrvIRead_<r>_<v>** ( [IN Rte_Instance instance] )`	
`<DataType> ***Rte_IrvIRead_<r>_<v>** ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<DataType>	The return value contains the buffered content of the Inter-Runnable Variable for primitive data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
<DataType> *	The return value contains a reference to the buffered content of the Inter-Runnable Variable for composite data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) read access to the Inter-Runnable Variable in the SWC configuration.	
Functional Description	
The function `Rte_IrvIRead_<r>_<v>()` supplies the value of the Inter-Runnable Variable, stored in a buffer before the runnable entity is started. This API is used to read the buffered (implicit) Inter-Runnable Variable. After startup `Rte_IrvIRead` provides the configured initial value.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

4.9.4 Rte_IrvIWrite

Prototype	
`void Rte_IrvIWrite_<r>_<v> ( [IN Rte_Instance instance,] IN <DataType> data )`	
`void Rte_IrvIWrite_<r>_<v> ( [IN Rte_Instance instance,] IN <DataType> *data)`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
data	The input data <data> is passed by value for primitive data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
*data	The input data <data> is passed by reference for composite data types. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) write access to the Inter-Runnable Variable in the SWC configuration.	
Functional Description	
The function `Rte_IrvIWrite_<r>_<v>()` can be used for updating buffered (implicit) Inter-Runnable Variables. Note, that the actual update is performed and therefore visible for other runnable entities after the runnable entity has been terminated.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

Caution
When buffered (implicit) write access to an Inter-Runnable Variable has been configured for a runnable, the runnable has to update the Inter-Runnable variable at least once during its execution time using the `Rte_IrvIWrite` API. Otherwise, the content of the Inter-Runnable Variable may become undefined upon return from the runnable.

4.9.5 Rte_IrvIWriteRef

Prototype	
`<DataType> *Rte_IrvIWriteRef_<r>_<v> ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<DataType> *	The return value contains a reference to the buffered content of the Inter-Runnable Variable. The <DataType> is the type of the Inter-Runnable Variable specified in the SWC description.
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) write access to the Inter-Runnable Variable in the SWC configuration.	
Functional Description	
The function `Rte_IrvIWriteRef_<r>_<v>()` returns a reference to the buffered (implicit) Inter-Runnable Variable. This is especially useful for composite data types. Note: the actual update is performed and therefore visible for other runnable entities after the runnable entity has been terminated.	
Call Context	
This function can be used inside the runnable <r> of an AUTOSAR software component (SWC). Usage in other runnables of the same SWC is forbidden! This function must not be called with locked interrupts.	

Caution
When buffered (implicit) write access to an Inter-Runnable Variable has been configured for a runnable, the runnable has to update the Inter-Runnable variable at least once during its execution time using the `Rte_IrvIWrite` API or writing to the location returned by the `Rte_IrvIWriteRef` API. Otherwise, the content of the Inter-Runnable Variable may become undefined upon return from the runnable.

4.10 Per-Instance Memory

4.10.1 Rte_Pim

Prototype	
`<C-type> *Rte_Pim_<n> ( [IN Rte_Instance instance] )`	
`<DataType> *Rte_Pim_<n> ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<C-Type> *	If the configured data type of the Per-Instance Memory is specified by any C type string, a reference to the PIM of the C-type is returned.
<DataType> *	If the configured DataType of the Per-Instance Memory is an AUTOSAR DataType, a reference to the PIM of this AUTOSAR type is returned. If the data type is known and completely described, the RTE generator knows the size of the PIM variable and is able to generate the PIM variables in a specific optimized order.
Existence	
This API exists for each specified Per-Instance Memory specified for an AUTOSAR application SWC.	
Functional Description	
The function `Rte_Pim_<n>()` can be used to access Per-Instance Memory. Note: If several runnable entities have concurrent access to the same Per-Instance Memory, the user has to protect the accesses by using implicit or explicit exclusive areas.	
Call Context	
This function can be used inside all runnable entities of the AUTOSAR software component (SWC) specifying the Per-Instance Memory.	

Caution
When the Per–Instance Memory uses no AUTOSAR data type and is also not based on a standard data type like e.g. `uint8` the RTE generator cannot create the type definition for this type.
In this case the user has to provide a user header file `Rte_UserTypes.h` which should contain the type definitions for the Per-Instance Memory allowing the RTE generator to allocate the Per-Instance memory.

4.11 Calibration Parameters

4.11.1 Rte_CData

Prototype	
`<DataType> **Rte_CData_<cp>** ( [IN Rte_Instance instance] )`	
`<DataType> ***Rte_CData_<cp>** ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<DataType>	For primitive data types the return value contains the content of the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
<DataType> *	For composite data types and string types the return value contains the reference to the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
Existence	
This API exists for each calibration element prototype specified for an AUTOSAR application SWC.	
Functional Description	
The function `Rte_CData_<cp>()` can be used to access SWC local calibration parameters. Depending on the configuration the Rte_CData API returns a SWC type specific (`shared`) or SWC instance specific (`perInstance`) calibration parameter.	
Call Context	
This function can be used inside all runnable entities of the AUTOSAR software component (SWC) specifying the calibration parameters.	

4.11.2 Rte_Prm

Prototype	
`<DataType> **Rte_Prm_<p>_<cp>** ( [IN Rte_Instance instance] )`	
`<DataType> ***Rte_Prm_<p>_<cp>** ( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
<DataType>	For primitive data types the return value contains the content of the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
<DataType> *	For composite data types and string types the return value contains the reference to the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
Existence	
This API exists for each calibration element prototype specified for a calibration software component.	
Functional Description	
The function `Rte_Prm_<p>_<cp>()` can be used to access the instance specific calibration element prototypes of a calibration component.	
Call Context	
This function can be used inside all runnable entities of the AUTOSAR software component (SWC) specifying access to calibration element prototypes of calibration components via calibration ports.	

4.11.3 SchM_CData

Prototype	
`<DataType> **SchM_CData_<Bsw>_<cp>** ( void )`	
`<DataType> ***SchM_CData_<Bsw>_<cp>** ( void )`	
Parameter	
-	
Return code	
<DataType>	For primitive data types the return value contains the content of the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
<DataType> *	For composite data types and string types the return value contains the reference to the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.

Existence
This API exists for each per instance parameter specified in the internal behavior of the BSW module description.
Functional Description
The function `SchM_CData_<Bsw>_<cp>()` can be used to access BSW local calibration parameters.
Call Context
This function can be used inside a schedulable entity in Task or Interrupt context.

4.12 Client-Server Communication

4.12.1 Rte_Call

Prototype	
`Std_ReturnType` **`Rte_Call_<p>_<o>`** `( [IN Rte_Instance instance,] {IN type [*]inputparam,}* {OUT type *outputparam,}* {INOUT type *inoutputparam,}* [, OUT Std_TransformerError transformerError])`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
[*]inputparam, *outputparam, *inoutputparam,	The number and type of parameters is determined by the operation prototype. Input (IN) parameters are passed by value (primitive types) or reference (composite and string types), output (OUT) and input-output (INOUT) parameters are always passed by reference.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Operation executed successfully.
RTE_E_UNCONNECTED	Indicates that the client port is not connected.
RTE_E_LIMIT	The operation is invoked while a previous invocation has not yet terminated. Relevant only for asynchronous calls.
RTE_E_COM_STOPPED	An infrastructure communication error was detected by the RTE. Relevant only to external communication. The error is not reported when the COM runs in a different partition.
RTE_E_TIMEOUT	Returned by a synchronous call after the timeout has expired and no other error occurred. The arguments are not changed. Also returned for synchronous and asynchronous calls when the server queue is full.
RTE_E_<interf>_<error>	Server runnables may return an application error if the operation execution was not successful. Application errors are defined at the client/server port interface and are references by the operation prototype.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the operation prototype in the DaVinci configuration.	

Functional Description
The function `Rte_Call_<p>_<o>()` invokes the server operation <o> with the specified parameters. If `Rte_Call` returns with an error, the INOUT and OUT parameters are unchanged.
Call Context
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.

Caution
In case of multiple faults during a call of Rte_Call in the context of LdComTp the priority of the resulting return value RTE_E_COM_BUSY is higher than the priority of the return value RTE_E_HARD_TRANSFORMER_ERROR.

Caution
It is the responsibility of the integrator of the RTE to assure that the task mapping and server queue lengths allow processing all client-server calls in time.

Especially if the server is mapped to a task with a lower priority than the client task and the server queue size is small (e.g. 1), the first call from the client task may lead to RTE_E_TIMEOUT in all other calls until the server task completely finalized the server call.

The server queue size of the server SWC can be overwritten with the parameter /MICROSAR/Rte/RteSwComponentInstance/RteEventToTaskMapping/RteServerQueueLength.

4.12.2 Rte_Result

Prototype	
`Std_ReturnType` **`Rte_Result_<p>_<o>`** `( [IN Rte_Instance instance,] {OUT type *outputparam,}* {INOUT type *inoutputparam,}* [, OUT Std_TransformerError transformerError])`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
*outputparam, *inoutputparam	The number and type of parameters is determined by the operation prototype. The output (OUT) and input-output (INOUT) parameters are always passed by reference.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Operation executed successfully.
RTE_E_UNCONNECTED	Indicates that the client port is not connected.
RTE_E_NO_DATA	The result of the asynchronous operation invocation is not available. Relevant only for non-blocking call.
RTE_E_COM_STOPPED	An infrastructure communication error was detected by the RTE. Relevant only to external communication. The error is not reported when the COM runs in a different partition.
RTE_E_TIMEOUT	The result of the asynchronous operation invocation is not available in the specified time. Relevant only for blocking call.
RTE_E_<interf>_<error>	Server runnables may return an application error if the operation execution was not successful. Application errors are defined at the client/server port interface and are references by the operation prototype.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured polling or waiting access to an asynchronous invoked operation of a C/S port interface.	
Functional Description	
The function `Rte_Result_<p>_<o>()` provides the result of asynchronous C/S calls. In case of a polling call, the API returns the OUT parameters if the result is already available while for asynchronous calls the API waits until the server runnable has finished the execution or a timeout occurs.	

Call Context
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.

4.13 Indirect API

4.13.1 Rte_Ports

Prototype	
`Rte_PortHandle_<i>_<R/P>` **`Rte_Ports_<i>_<P/R>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
Rte_PortHandle_<i>_<R/P>	The API returns a pointer to the first port data structure of the port data structure array.
Existence	
This API exists, if the indirect API is configured at the Component Type.	
Functional Description	
The function `Rte_Ports_<i>_<R/P>` returns an array containing the port data structures of all require ports indicated by the API extension <R> or provide ports indicated by <P> of the port interface specified by <i> in order to allow indirect access of the port APIs via the port handle (e.g. iteration over all ports of the same interface).	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC).	

4.13.2 Rte_NPorts

Prototype	
`uint8` **`Rte_NPorts_<i>_<P/R>`** `( [IN Rte_Instance instance] )`	
Parameter	
instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
`uint8`	The API returns the size of the port data structure array provided by `Rte_Ports`.
Existence	
This API exists, if the indirect API is configured at the component type.	
Functional Description	
The function `Rte_NPorts_<i>_<R/P>` returns the number of array entries (port data structures) of all require ports indicated by the API extension <R> or provide ports indicated by <P> of the port interface specified by <i>.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC).	

4.13.3 Rte_Port

Prototype	
`Rte_PortHandle_<i>_<R/P> Rte_Port_<p> ( [IN Rte_Instance instance] )`	
Parameter	
Instance	Instance handle, used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
Rte_PortHandle_<i>_<R/P>	The API returns a pointer to a port data structure.
Existence	
This API exists, if the indirect API is configured at the component type.	
Functional Description	
The function `Rte_Port_<p>` returns the port data structure of the port specified by <p>. It allows indirect API access via the port handle.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC).	

4.14 RTE Lifecycle API

The lifecycle API functions are declared in the RTE lifecycle header file `Rte_Main.h`

4.14.1 Rte_Start

Prototype	
`Std_ReturnType` **`Rte_Start`** `( void )`	
Parameter	
-	
Return code	
RTE_E_OK	RTE initialized successfully.
RTE_E_LIMIT	An internal limit has been exceeded.
Functional Description	
The RTE lifecycle API function `Rte_Start` allocates and initializes system resources and communication resources used by the RTE.	
Call Context	
This function has to be called by the ECU state manager after basic software modules used by the RTE have been initialized, especially the OS and the COM. It has to be called on every core that is used by the RTE. The call on the core that contains the BSW will start the triggering of all cyclic and background runnables (if `Rte_StartTiming` is not enabled). Therefore `Rte_Start` on the other cores and SchM_Init on all cores has to be executed first. This function must not be called with locked interrupts.	

Caution
The RTE starts triggering of runnable entities, when the BSW core that contains the BSW modules COM, LDCOM and NVM calls Rte_Start. This means Rte_Start on the other cores needs to be called before Rte_Start is called on the core that contains the BSW. Moreover, Rte_Starts sets the init state as last action. This means that integration code also needs to assure that the alarms and task activations in Rte_Start that are executed before do no trigger any runnable entities before this state is set. This can be done e.g. by configuring activation offsets for the alarms, using non-preemptive init tasks and polling the Rte_InitState from the non-preemptive init tasks.

4.14.2 Rte_StartTiming

Prototype	
`void` **`Rte_StartTiming`** `( void )`	
Parameter	
-	

Return code	
-	
Functional Description	
The RTE lifecycle API function `Rte_StartTiming` activates the triggering of all cyclic and background runnables. Must be explicitly enabled to be available.	
Call Context	
This function has to be called by the ECU state manager after basic software modules used by the RTE have been initialized, especially the OS and the COM. The call on the core that contains the BSW will start the triggering of all cyclic and background runnables. For other cores, this API performs no operations. `Rte_Start` must be executed before calling `Rte_StartTiming`. This function must not be called with locked interrupts.	

4.14.3 Rte_Stop

Prototype	
`Std_ReturnType` **`Rte_Stop`** `( void )`	
Parameter	
-	
Return code	
RTE_E_OK	RTE initialized successfully.
RTE_E_LIMIT	A resource could not be released.
Functional Description	
The RTE lifecycle API function `Rte_Stop` releases system resources and communication resources used by the RTE and shutdowns the RTE. After `Rte_Stop` is called on the core that contains the BSW no runnable entity must be processed. The API only stops cyclic functionality. It does not terminate any tasks, therefore runnables may still be running after Rte_Stop was called.	
Call Context	
This function has to be called by the ECU state manager on every core that is used by the RTE. The call on the core that contains the BSW will stop the triggering of the cyclic runnables. This function must not be called with locked interrupts.	

Caution
The RTE stops triggering of runnable entities, when the BSW core that contains the BSW modules COM, LDCOM and NVM calls Rte_Stop. This means Rte_Stop on the other cores need to be called after Rte_Stop is called on the core that contains the BSW. Rte_Stop does not terminate any tasks. Therefore, runnables that are currently running are not terminated. If needed, scheduling of new tasks and runnables need to be blocked by the caller.

4.14.4 Rte_InitMemory

Prototype
`void Rte_InitMemory ( void )`
Parameter
-
Return code
-
Functional Description
The API function `Rte_InitMemory` is a MICROSAR Classic RTE specific extension and should be used to initialize RTE internal state variables if the compiler does not support initialized variables.
Call Context
This function has to be called before the ECU state manager calls the initialization functions of other BSW modules especially the AUTOSAR COM module. It needs to be called on the core that contains the BSW.

Caution
Rte_InitMemory API is a Vector extension to the AUTOSAR standard and may not be supported by other RTE generators.
The generation of the Rte_InitMemory API is optional and can be deactivated with */MICROSAR/Rte/RteGeneration/RteDisableRteInitMemory*. If the Rte does not generate the Rte_InitMemory, the user must ensure that the global Rte variables are initialized.

4.15 SchM Lifecycle API

The lifecycle API functions are declared in the RTE lifecycle header file `Rte_Main.h`

4.15.1 SchM_Init

Prototype	
`void **SchM_Init** ( [IN SchM_ConfigType ConfigPtr] )`	
Parameter	
ConfigPtr	Pointer to the Rte_Config_<VariantName> data structure that shall be used for the RTE initialization of the active variant in case of a postbuild selectable configuration. The parameter is omitted in case the project contains no postbuild selectable variance.
Return code	
-	
Functional Description	
This function allocates and initializes system resources used by the BSW Scheduler.	
Call Context	
This function has to be called by the ECU state manager from task context. The OS has to be initialized before as well as those BSW modules for which the SchM provides triggering of schedulable entities (main functions). The API has to be called on all cores that are used by the RTE. This function must not be called with locked interrupts.	

4.15.2 SchM_Start

Prototype	
`void **SchM_Start** ( void )`	
Parameter	
-	
Return code	
-	
Functional Description	
This function initializes the BSW Scheduler.	
Call Context	
This function has to be called by the ECU state manager from task context. It shall be called before the BswM is initialized. The API has to be called on all cores that are used by the RTE. This function must not be called with locked interrupts.	

4.15.3 SchM_StartTiming

Prototype	
`void SchM_StartTiming ( void )`	
Parameter	
-	
Return code	
-	
Functional Description	
This function starts the timers for all cyclic triggered schedulable entities (main functions). Note that all main function calls are activated upon return from this function.	
Call Context	
This function has to be called by the ECU state manager from task context after the SchM has been initialized. The API has to be called on all cores that are used by the RTE. This function must not be called with locked interrupts.	

4.15.4 SchM_Deinit

Prototype	
`void SchM_Deinit ( void )`	
Parameter	
-	
Return code	
-	
Functional Description	
This function finalizes the BSW Scheduler and stops the timer which triggers the main functions.	
Call Context	
This function has to be called by the ECU state manager from task context. It has to be called on all cores that are used by the RTE. SchM_Deinit can only be called after Rte_Stop has been called on all cores. This function must not be called with locked interrupts.	

4.15.5 SchM_GetVersionInfo

Prototype	
`void` **`SchM_GetVersionInfo`** `( OUT Std_VersionInfoType *versioninfo )`	
Parameter	
versioninfo	Pointer to where to store the version information of this module.
Return code	
-	
Existence	
This API exists if `RteSchMVersionInfoApi` is enabled.	
Functional Description	
`SchM_GetVersionInfo()` returns version information, vendor ID and AUTOSAR module ID of the component. The versions are decimal-coded.	
Call Context	
The function can be called on interrupt and task level.	

4.16 VFB Trace Hooks

The RTE's "VFB tracing" mechanism allows to trace interactions of the AUTOSAR software components with the VFB. The choice of events resides with the user and can range from none to all. The "VFB tracing" functionality is designed to support multiple clients for each event. If one or multiple clients are specified for an event, the trace function without client prefix will be generated followed by the trace functions with client prefixes in alphabetically ascending order.

4.16.1 Rte_[<client>_]<API>Hook_<cts>_<ap>_Start

Prototype	
`void` **`Rte_[<client>_]<API>Hook_<cts>_<ap>_Start`** `( [IN const Rte_CDS_<cts>* inst,] params )`	
Parameter	
Rte_CDS_<cts>* inst	The instance specific pointer of type Rte_CDS_<cts> is used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
params	The parameters are the same as the parameters of the <API>. See the corresponding API description for details.
Return code	
-	
Existence	
This VFB trace hook exists if the global and the hook specific configuration switches are enabled.	
Functional Description	
This VFB trace hook is called inside the RTE APIs directly after invocation of the API. The user has to provide this hook function if it is enabled in the configuration. The placeholder <API> represents one of the following APIs: Enter, Exit, Write, Read, DRead, Send, Receive, Invalidate, IsUpdated, Feedback, SwitchAck, Switch, Mode, Call, Result, IrvWrite, IrvRead, Trigger, IrTrigger The <AccessPoint> is defined as follows: ▶ Enter, Exit: <ExclusiveArea> ▶ Write, Read, DRead, Send, Receive, Feedback, Invalidate, IsUpdated: <PortPrototype>_<DataElementPrototype> ▶ Switch, SwitchAck, Mode: <PortPrototype>_<ModeDeclarationGroupPrototype> ▶ Call, Result: <PortPrototype>_<OperationPrototype> ▶ IrvWrite, IrvRead: <InterRunnableVariable> ▶ Trigger, IrTrigger: <PortPrototype>_<TriggerElement>	
Call Context	
This function is called inside the RTE API. The call context is the context of the API itself. Since APIs can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component (SWC).	

4.16.2 Rte_[<client>_]<API>Hook_<cts>_<ap>_Return

Prototype	
`void` **`Rte_[<client>_]<API>Hook_<cts>_<ap>_Return`** `( [IN const Rte_CDS_<cts> *inst,] params )`	
Parameter	
Rte_CDS_<cts>* inst	The instance specific pointer of type Rte_CDS_<cts> is used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
params	The parameters are the same as the parameters of the API. See the corresponding API description for details.
Return code	
-	
Existence	
This VFB trace hook exists if the global and the hook specific configuration switches are enabled.	
Functional Description	
This VFB trace hook is called inside the RTE APIs directly before leaving the API. The user has to provide this hook function if it is enabled in the configuration. The placeholder <API> represents one of the following APIs: Enter, Exit, Write, Read, DRead, Send, Receive, Invalidate, IsUpdated, Feedback, SwitchAck, Switch, Mode, Call, Result, IrvWrite, IrvRead, Trigger, IrTrigger The <AccessPoint> is defined as follows: ▶ Enter, Exit: <ExclusiveArea> ▶ Write, Read, DRead, Send, Receive, Feedback, Invalidate, IsUpdated: <PortPrototype>_<DataElementPrototype> ▶ Switch, SwitchAck, Mode: <PortPrototype>_<ModeDeclarationGroupPrototype> ▶ Call, Result: <PortPrototype>_<OperationPrototype> ▶ IrvWrite, IrvRead: <InterRunnableVariable> ▶ Trigger, IrTrigger: <PortPrototype>_<TriggerElement>	
Call Context	
This function is called inside the RTE API. The call context is the context of the API itself. Since APIs can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component (SWC).	

Caution
The RTE generator tries to prevent overhead by sometimes implementing the Rte_Call API as a macro that does a direct runnable invocation. If VFB trace hooks are enabled for such a Rte_Call API or for the called server runnable, these optimizations are no longer possible.
Macro optimizations for Rte_Read, Rte_DRead, Rte_Write, Rte_IrvRead and Rte_IrvWrite APIs are also disabled when VFB tracing for these APIs are enabled.

Caution
The RTE does not call VFB trace hooks for the following APIs because they are intended to be implemented as macros.

- Implicit S/R APIs: Rte_IWrite, Rte_IWriteRef, Rte_IRead, Rte_IStatus, Rte_IInvalidate
- Implicit Inter-Runnable Variables: Rte_IrvIWrite, Rte_IrvIWriteRef, Rte_IrvIRead
- Per-instance Memory and calibration parameter APIs: Rte_Pim, Rte_CData, Rte_Prm
- Indirect APIs: Rte_Ports, Rte_Port, Rte_NPorts
- RTE Life-Cycle APIs: Rte_Start, Rte_Stop

4.16.3 SchM_[<client>_]<API>Hook_<Bsw>_<ap>_Start

Prototype	
`void SchM_[<client>_]<API>Hook_<bsw>_<ap>_Start ( params )`	
Parameter	
params	The parameters are the same as the parameters of the <API>. See the corresponding API description for details.
Return code	
-	
Existence	
This VFB trace hook exists if the global and the hook specific configuration switches are enabled.	
Functional Description	
This VFB trace hook is called inside the RTE APIs directly after invocation of the API. The user has to provide this hook function if it is enabled in the configuration. The placeholder <API> represents one of the following APIs: Enter, Exit The <AccessPoint> is defined as follows: ▶ Enter, Exit: <ExclusiveArea>	
Call Context	
This function is called inside the RTE API. The call context is the context of the API itself. Since APIs can be called from a BSW function, the context of the trace hook depends on the context of the BSW function.	

Caution
The SchM Hook APIs are a Vector extension to the AUTOSAR standard and may not be supported by other RTE generators.

4.16.4 SchM_[<client>_]<API>Hook_<Bsw>_<ap>_Return

Prototype	
`void SchM_[<client>_]<API>Hook_<bsw>_<ap>_Return ( params )`	
Parameter	
params	The parameters are the same as the parameters of the <API>. See the corresponding API description for details.
Return code	
-	
Existence	
This VFB trace hook exists if the global and the hook specific configuration switches are enabled.	
Functional Description	
This VFB trace hook is called inside the RTE APIs directly before leaving the API. The user has to provide this hook function if it is enabled in the configuration. The placeholder <API> represents one of the following APIs: Enter, Exit The <AccessPoint> is defined as follows: ▶ Enter, Exit: <ExclusiveArea>	
Call Context	
This function is called inside the RTE API. The call context is the context of the API itself. Since APIs can be called from a BSW function, the context of the trace hook depends on the context of the BSW function.	

Caution
The SchM Hook APIs are a Vector extension to the AUTOSAR standard and may not be supported by other RTE generators.

4.16.5 Rte_[<client>_]ComHook_<SignalName>_SigTx

Prototype	
`void` **`Rte_[<client>_]ComHook_<SignalName>_SigTx`** `( <DataType> *data )`	
Parameter	
<DataType>* data	Pointer to data to be transmitted via the COM API. Note: <DataType> is the application specific data type of `Rte_Send`, `Rte_Write` or `Rte_IWrite`.
Return code	
-	
Existence	
This VFB trace hook exists, if at least one data element prototype of a port prototype has to be transmitted over a network (Inter-Ecu) and the global and the hook specific configuration switches are enabled.	
Functional Description	
This hook is called just before the RTE invokes `Com_SendSignal, Com_SendDynSignal` or `Com_UpdateShadowSignal`.	
Call Context	
This function is called inside the RTE APIs `Rte_Send` and `Rte_Write`. The call context is the context of the API itself. Since APIs can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component. If buffered communication (`Rte_IWrite`) is used, the call context is the task of the mapped runnable.	

4.16.6 Rte_[<client>_]ComHook_<SignalName>_SigIv

Prototype
`void` **`Rte_[<client>_]ComHook_<SignalName>_SigIv`** `( void )`
Parameter
-
Return code
-
Existence
This VFB trace hook exists, if at least one data element prototype of a port prototype has to be transmitted over a network (Inter-Ecu) and the global and the hook specific configuration switches are enabled. In addition, the `canInvalidate` attribute at the UnqueuedSenderComSpec of the data element prototype must be enabled.
Functional Description
This hook is called just before the RTE invokes `Com_InvalidateSignal`.
Call Context
This function is called inside the RTE APIs `Rte_Invalidate`. The call context is the context of the API itself. Since APIs can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component. If buffered communication (`Rte_IInvalidate`) is used, the call context is the task of the mapped runnable.

4.16.7 Rte_[<client>_]ComHook_<SignalName>_SigGroupIv

Prototype
`void` **`Rte_[<client>_]ComHook_<SignalGroupName>_SigGroupIv`** `( void )`
Parameter
-
Return code
-
Existence
This VFB trace hook exists, if at least one data element prototype of a port prototype is composite and has to be transmitted over a network (Inter-Ecu) and the global and the hook specific configuration switches are enabled. In addition, the `canInvalidate` attribute at the UnqueuedSenderComSpec of the data element prototype must be enabled.
Functional Description
This hook is called just before the RTE invokes `Com_InvalidateSignalGroup`.
Call Context
This function is called inside the RTE APIs `Rte_Invalidate`. The call context is the context of the API itself. Since APIs can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component. If buffered communication (`Rte_IInvalidate`) is used, the call context is the task of the mapped runnable.

4.16.8 Rte_[<client>_]ComHook_<SignalName>_SigRx

Prototype	
`void` **`Rte_[<client>_]ComHook_<SignalName>_SigRx`** `( <DataType> *data )`	
Parameter	
<DataType>* data	Pointer to the data received via the COM API. Note: <DataType> is the application specific data type of `Rte_Receive`, `Rte_Read`, `Rte_DRead` or `Rte_IRead`.
Return code	
-	
Existence	
This VFB trace hook exists, if at least one data element prototype of a port prototype has to be received from a network and the global and hook specific configuration switches are enabled.	
Functional Description	
This VFB Trace Hook is called after the RTE invokes `Com_ReceiveSignal`, `Com_ReceiveDynSignal` or `Com_ReceiveShadowSignal`.	
Call Context	
This function is called inside the RTE API `Rte_Read` or `Rte_DRead`. The call context is the context of the API itself. Since this API can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component. If buffered communication (`Rte_IRead`) is used, the call context is the task of the mapped runnable. If queued communication is configured (`Rte_Receive`), the call of the Com API is called inside the COM callback after reception. In this case, the context of the trace hook is the context of the COM callback. Note: This could be the task context or the interrupt context!	

4.16.9 Rte_[<client>_]ComHook<Event>_<SignalName>

Prototype
`void` **`Rte_[<client>_]ComHook<Event>_<SignalName>`** `( void )`
Parameter
-
Return code
-
Existence
This VFB trace hook is called inside the <Event> specific COM callback, directly after the invocation by the COM and if the global and the hook specific configuration switches are enabled.
Functional Description
This trace hook indicates the start of a COM callback. <Event> depends on the type of the callback. ▶ empty string: Rte_COMCbk_<SignalName> ▶ TxTOut Rte_COMCbkTxTOut_<SignalName> ▶ RxTOut Rte_COMCbkRxTOut_<SignalName> ▶ TAck Rte_COMCbkTAck_<SignalName> ▶ TErr Rte_COMCbkTErr_<SignalName> ▶ Inv Rte_COMCbkInv_<SignalName>
Call Context
This function is called inside the context of the COM callback. Note: This could be the task context or the interrupt context!

4.16.10 Rte_[<client>_]LdComHook_<SignalName>_SigTx

Prototype	
`void` **`Rte_[<client>_]LdComHook_<SignalName>_SigTx`** `( PduInfoType *PduInfoPtr)`	
Parameter	
PduInfoType *PduInfoPtr	Pointer to data to be transmitted via the LDCOM API. Note: `PduInfoType` is a struct containing a pointer to the application specific data type of `Rte_Send`, `Rte_Write` or `Rte_IWrite` and a pointer to the length-buffer.
Return code	
-	
Existence	
This VFB trace hook exists, if at least one data element prototype of a port prototype has to be transmitted over a network (Inter-Ecu) via LDCOM and the global and the hook specific configuration switches are enabled.	
Functional Description	
This hook is called just before the RTE invokes `LdCom_Transmit`.	
Call Context	
This function is called inside the RTE APIs `Rte_Send` and `Rte_Write`. The call context is the context of the API itself. Since APIs can only be called in a runnable context, the context of the trace hook is also the runnable entity of an AUTOSAR software component. If buffered communication (`Rte_IWrite`) is used, the call context is the task of the mapped runnable.	

4.16.11 Rte_[<client>_]Task_Activate

Prototype	
`void` **`Rte_[<client>_]Task_Activate`** `( TaskType task )`	
Parameter	
task	The same parameter is also used to call the OS API `ActivateTask`.
Return code	
-	
Existence	
This VFB trace hook is called by the RTE immediately before the invocation of the OS API `ActivateTask` and if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates the call of `ActivateTask` of the OS.	
Call Context	
This function is called inside `Rte_Start` and in the context RTE API functions which trigger the execution of a runnable entity where the runnable is mapped to a basic task. For API functions, the call context is the runnable context.	

4.16.12 Rte_[<client>_]Task_Terminate

Prototype	
`void` **`Rte_[<client>_]Task_Terminate`** `( TaskType task )`	
Parameter	
task	The same parameter is also used to call the OS API `TerminateTask` or `ChainTask`.
Return code	
-	
Existence	
This VFB trace hook is called by the RTE immediately before the invocation of the OS API `TerminateTask` and `ChainTask` if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates the call of `TerminateTask` or `ChainTask` of the OS.	
Call Context	
This function is called inside `RTE` generated task bodies.	

4.16.13 Rte_[<client>_]Task_Dispatch

Prototype	
`void` **`Rte_[<client>_]Task_Dispatch`** `( TaskType task )`	
Parameter	
task	The parameter indicates the task which was started (dispatched) by the OS.
Return code	
-	
Existence	
This VFB trace hook exists for each configured RTE task and is called directly after the start if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates the call activation of a task by the OS.	
Call Context	
The call context is the task.	

4.16.14 Rte_[<client>_]Task_SetEvent

Prototype	
`void` **`Rte_[<client>_]Task_SetEvent`** `( TaskType task, EventMaskType event )`	
Parameter	
task	The same parameter is also used to call the OS API `SetEvent`.
event	The same parameter is also used to call the OS API `SetEvent`.
Return code	
-	
Existence	
This VFB trace hook is called by the RTE immediately before the invocation of the OS API `SetEvent` and if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates the call of `SetEvent`.	
Call Context	
This function is called inside RTE API functions and in the COM callbacks. For API functions, the call context is the runnable context. Note: For the COM callbacks the context could be the task context or the interrupt context!	

4.16.15 Rte_[<client>_]Task_WaitEvent

Prototype	
`void` **`Rte_[<client>_]Task_WaitEvent`** `( TaskType task, EventMaskType event )`	
Parameter	
task	The same parameter is also used to call the OS API `WaitEvent`.
event	The same parameter is also used to call the OS API `WaitEvent`.
Return code	
-	
Existence	
This VFB trace hook is called by the RTE immediately before the invocation of the OS API `WaitEvent` and if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates the call of `WaitEvent`.	
Call Context	
This function is called inside RTE API functions and in generated task bodies.	

4.16.16 Rte_[<client>_]Task_WaitEventRet

Prototype	
`void` **`Rte_[<client>_]Task_WaitEventRet`** `( TaskType task, EventMaskType event )`	
Parameter	
task	The same parameter is also used to call the OS API `WaitEvent`.
event	The same parameter is also used to call the OS API `WaitEvent`.
Return code	
-	
Existence	
This VFB trace hook is called by the RTE immediately after returning from the OS API `WaitEvent` and if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates leaving the call of `WaitEvent`.	
Call Context	
This function is called inside RTE API functions and in generated task bodies.	

4.16.17 Rte_[<client>_]Runnable_<cts>_<r>_Start

Prototype	
`void` **`Rte_[<client>_]Runnable_<cts>_<r>_Start`** `( [IN const Rte_CDS_<cts> *inst] )`	
Parameter	
Rte_CDS_<cts>* inst	The instance specific pointer of type Rte_CDS_<cts> is used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
-	
Existence	
This VFB trace hook is called for all mapped runnable entities if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates invocation of the runnable entity. It is called just before the call of the runnable entity and allows for example measurement of the execution time of a runnable together with the counterpart `Rte_[<client>_]Runnable_<cts>_<r>_Return`.	
Call Context	
This function is called inside RTE generated task bodies.	

4.16.18 Rte_[<client>_]Runnable_<cts>_<r>_Return

Prototype	
`void` **`Rte_[<client>_]Runnable_<cts>_<r>_Return`** `( [IN const Rte_CDS_<cts> *inst] )`	
Parameter	
Rte_CDS_<cts>* inst	The instance specific pointer of type Rte_CDS_<cts> is used to distinguish between the different instances in case of multiple instantiation. Note: This is an optional parameter depending on the configuration of `supportsMultipleInstantiation` attribute.
Return code	
-	
Existence	
This VFB trace hook is called for all mapped runnable entities if the global and the hook specific configuration switches are enabled.	
Functional Description	
This trace hook indicates invocation of the runnable entity. It is called just after the call of the runnable entity and allows for example measurement of the execution time of a runnable together with the counterpart `Rte_[<client>_]Runnable_<cts>_<r>_Start`.	
Call Context	
This function is called inside RTE generated task bodies.	

4.16.19 SchM_[<client>_]Schedulable_<Bsw>_<entityName>_Start

Prototype	
`void` **`SchM_[<client>_]Schedulable_<BSW>_<entityName>_Start`** `( void )`	
Parameter	
Return code	
-	
Existence	
This VFB trace hook exists if the global and the hook specific configuration switches are enabled.	
Functional Description	
This VFB trace hook is called just before the execution of BSW Schedulable entry starts via its entry point.	
Call Context	
This function is called inside the RTE API. The call context is the context of the API itself. Since APIs can be called from a BSW function, the context of the trace hook depends on the context of the BSW function.	

4.16.20 SchM_[<client>_]Schedulable_<Bsw>_<entityName>_Return

Prototype
`void` **`SchM_[<client>_]Schedulable_<BSW>_<entityName>_Return`** `( void )`
Parameter
Return code
-	
Existence
This VFB trace hook exists if the global and the hook specific configuration switches are enabled.
Functional Description
This VFB trace hook is called immediately after execution returns to BSW Scheduler code from a BSW Schedulable entity.
Call Context
This function is called inside the RTE API. The call context is the context of the API itself. Since APIs can be called from a BSW function, the context of the trace hook depends on the context of the BSW function.

4.17 RTE Implementation Plugins

The RTE Implementation Plugin (RIP) mechanism allows the use of custom implementations instead of APIs generated by the RTE generator. The MICROSAR Classic RTE generator currently only supports a subset of the RIP mechanism. A RIP has to be configured for a Data Communication Graph by providing a FlatMap with a Flat Instance Descriptor and the Data Communication Graph elements. If there is a RIP configured for a Data Communication Graph it is assumed the RIP support is globally enabled. Specifying a RIP through Uri References is not yet supported.

Currently the MICROSAR Classic RTE supports the following RIP APIs:

Supported RIP APIs
Rte_Rips_StartRead
Rte_Rips_StopRead
Rte_Rips_StartWrite
Rte_Rips_StopWrite
Rte_Rips_Write
Rte_Rips_Read
Rte_Rips_DRead
Rte_Rips_IWrite
Rte_Rips_IRead
Rte_Rips_IWBufferRef
Rte_Rips_IRBufferRef
Rte_Rips_Feedback
Rte_Rips_DataIsUpdated
Rte_Rips_FillFlushRoutine

Table 4-2 Supported RIP APIs

Caution
Rte Implementation Plugins in combination with Inter-Runnable Variables are not supported.

4.17.1 Rte_Rips_StartRead

Prototype
`void` **`Rte_Rips_<pi>_StartRead_<ctp>_<cgi>`** `( void )`
`void` **`Rte_Rips_<pi>_StartRead_<ctp>_GetMirror_<cgi>`** `( void )`
Parameter
-
Return code
-
Existence
This API exists, if the runnable entity of a SWC has configured direct (explicit) access in the role `dataReceivePointByArgument` for the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_RTE`. The referenced data element prototype is configured without queued communication (`isQueued=false`).
Functional Description
The implementation of the `Rte_Rips_<pi>_StartRead_<ctp>[_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.
Call Context
This function is called inside the body of the generated RTE APIs.

4.17.2 Rte_Rips_StopRead

Prototype
`void` **`Rte_Rips_<pi>_StopRead_<ctp>_<cgi>`** `( void )`
`void` **`Rte_Rips_<pi>_StopRead_<ctp>_GetMirror_<cgi>`** `( void )`
Parameter
-
Return code
-
Existence
This API exists, if the runnable entity of a SWC has configured direct (explicit) access in the role `dataReceivePointByArgument` for the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_RTE`. The referenced data element prototype is configured without queued communication (`isQueued=false`).
Functional Description
The implementation of the `Rte_Rips_<pi>_StopRead_<ctp>[_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.
Call Context
This function is called inside the body of the generated RTE APIs.

4.17.3 Rte_Rips_StartWrite

Prototype
`void` **`Rte_Rips_<pi>_StartWrite_<ctp>_<cgi>`** `( void )`
`void` **`Rte_Rips_<pi>_StartWrite_<ctp>_SetMirror_<cgi>`** `( void )`
`void` **`Rte_Rips_<pi>_StartWrite_<ctp>_NvMNotifyInitBlock_<cgi>`** `( void )`
Parameter
-
Return code
-
Existence
This API exists, if the runnable entity of a SWC has configured direct (explicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_RTE`. The referenced data element prototype is configured without queued communication (`isQueued=false`).
Functional Description
The implementation of the `Rte_Rips_<pi>_StartWrite_<ctp>[_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.
Call Context
This function is called inside the body of the generated RTE APIs.

4.17.4 Rte_Rips_StopWrite

Prototype
`void` **`Rte_Rips_<pi>_StopWrite_<ctp>_<cgi>`** `( void )`
`void` **`Rte_Rips_<pi>_StopWrite_<ctp>_SetMirror_<cgi>`** `( void )`
`void` **`Rte_Rips_<pi>_StopWrite_<ctp>_NvMNotifyInitBlock_<cgi>`** `( void )`
Parameter
-
Return code
-
Existence
This API exists, if the runnable entity of a SWC has configured direct (explicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_RTE`. The referenced data element prototype is configured without queued communication (`isQueued=false`).

Functional Description
The implementation of the `Rte_Rips_<pi>_StopWrite_<ctp>[_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.
Call Context
This function is called inside the body of the generated RTE APIs.

4.17.5 Rte_Rips_Write

Prototype	
`Std_ReturnType` **`Rte_Rips_<pi>_Write_[<ctp>][<pa>][_<exe>]_<cgi>`** `(IN <DataType> data [, OUT Std_TransformerError transformerError] )`	
Parameter	
data	The input data <data> for primitive, string and composite data data types is passed by value. The <DataType> is the type, specified at the data element prototype in the SWC description.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Data passed to communication services successfully.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
RTE_E_LIMIT	The submitted data has been discarded because the receiver queue is full. Only relevant for intra ECU communication.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local or Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`. This API also exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`.	
Functional Description	
The implementation of the `Rte_Rips_<pi>_Write_[<ctp>][<pa>][_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_Write_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor or in case of implicit access and configured Cross Cluster Plugin the function is called in the preemption area of the OS task where the RunnableEntitiy is mapped to.	

Note
In case of the existence of a Cross Cluster Plugin and the communication is non-queued the Ecuc Partition name <pa> exists instead of the ComponentTypePrototype <ctp> for the function prototype.

Note
In case of the existence of a Cross Cluster Plugin, the expected return value is always RTE_E_OK.

Caution
Rte_Rips_Write in combination with NvBlockComponents is not supported.

4.17.6 Rte_Rips_Read

Prototype	
`Std_ReturnType` **`Rte_Rips_<pi>_Read_[<ctp>][<pa>][_<exe>]_<cgi>`** `(OUT <DataType> *data [, OUT Std_TransformerError transformerError] )`	
Parameter	
*data	The output <data> is passed by reference. The <DataType> is the type, specified at the data element prototype in the SWC description.
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
RTE_E_OK	Data read successfully.
RTE_E_UNCONNECTED	Indicates that the receiver port is not connected.
RTE_E_HARD_TRANSFORMER_ERROR	An error during transformation occurred which produces invalid data as output.
RTE_E_SOFT_TRANSFORMER_ERROR	An error during transformation occurred which shall be notified to the SWC but still produces valid data as output.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) access in the role `dataReceivePointByArgument` to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local or Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`. This API also exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`.	
Functional Description	
The implementation of the `Rte_Rips_<pi>_Read_[<ctp>][<pa>][_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_Read_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor or in case of implicit access and configured Cross Cluster Plugin the function is called in the preemption area of the OS task where the RunnableEntitiy is mapped to.	

Note
In case of the existence of a Cross Cluster Plugin and the communication is non-queued the Ecuc Partition name `<pa>` exists instead of the ComponentTypePrototype `<ctp>` for the function prototype.

Caution
Rte_Rips_Read in combination with NvBlockComponents is not supported.

4.17.7 Rte_Rips_DRead

Prototype	
`<DataType> Rte_Rips_<pi>_DRead_[<ctp>][<pa>][_<exe>]_<cgi> ( [OUT Std_TransformerError transformerError] )`	
Parameter	
transformerError	Optional OUT parameter if `transformerErrorHandling` is enabled.
Return code	
<DataType>	The return value contains the current value of the data element. The <DataType> is the (primitive) type, specified at the data element prototype in the SWC description.
Existence	
This API exists, if the runnable entity of a SWC has configured direct (explicit) access to the data element in the role `dataReceivePointByValue` in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local or Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`. The referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The implementation of the `Rte_Rips_<pi>_DRead_[<ctp>][<pa>][_<exe>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_DRead_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

Note
In case of the existence of a Cross Cluster Plugin the Ecuc Partition name `<pa>` exists instead of the ComponentTypePrototype `<ctp>` for the function prototype.

Caution
Cross Cluster Plugins in combination with Local Cluster Plugins can only be used for P-Ports of Nonqueued Sender-Receiver communication. In other cases, only a single plugin is supported at a time.

4.17.8 Rte_Rips_IWrite

Prototype	
`void` **`Rte_Rips_<pi>_IWrite_<ctp>_<exe>_<cgi>`** `( IN <DataType> data )`	
Parameter	
data	The input data <data> for primitive data types without string types is passed by value. The <DataType> is the type, specified at the data element prototype in the SWC description.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the primitive data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN` and the RtePluginSupportsIReadIWrite is set to `true`. The referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The implementation of the `Rte_Rips_<pi>_IWrite_<ctp>_<exe>_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_IWrite_<r>_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

4.17.9 Rte_Rips_IRead

Prototype	
`<DataType>` **`Rte_Rips_<pi>_IRead_<ctp>_<exe>_<cgi>`** `( void )`	
Parameter	
-	
Return code	
<DataType>	The return value contains the buffered data for primitive data types. <DataType> is the type, specified at the data element prototype in the SWC description
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the primitive data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN` and the RtePluginSupportsIReadIWrite is set to `true`. The referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The implementation of the `Rte_Rips_<pi>_IRead_<ctp>_<exe>_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_IRead_<r>_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

4.17.10 Rte_Rips_IWBufferRef

Prototype	
`<DataType> *Rte_Rips_<pi>_IWBufferRef_<ctp>_<exe>_<cgi> ( void )`	
Parameter	
-	-
Return code	
<DataType> *	The return value contains a reference to the buffered data and optionally the status. <DataType> is the CDS type, specified at the data element prototype in the SWC description
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`. The referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The implementation of the `Rte_Rips_<pi>_IWBufferRef_<ctp>_<exe>_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_IWrite_<r>_<p>_<d>()` or `Rte_IWriteRef_<r>_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

Note
The buffer has to be provided by the plugin. The declaration of the CDS data type can be found in the `Rte_DataHandleType.h` file.

4.17.11 Rte_Rips_IRBufferRef

Prototype	
`<DataType> *Rte_Rips_<pi>_IRBufferRef_<ctp>_<exe>_<cgi> ( void )`	
Parameter	
-	-
Return code	
<DataType> *	The return value contains a reference to the buffered data and optionally the status. <DataType> is the CDS type, specified at the data element prototype in the SWC description
Existence	
This API exists, if the runnable entity of a SWC has configured buffered (implicit) access to the data element in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Local Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`. The referenced data element prototype is configured without queued communication (`isQueued=false`).	
Functional Description	
The implementation of the `Rte_Rips_<pi>_IRBufferRef_<ctp>_<exe>_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_IRead_<r>_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

Note
The buffer has to be provided by the plugin. The declaration of the CDS data type can be found in the `Rte_DataHandleType.h` file.

4.17.12 Rte_Rips_Feedback

Prototype	
`Std_ReturnType` **`Rte_Rips_<pi>_Feedback_[<ctp>][<pa>]_<cgi>`** `( void )`	
Parameter	
-	
Return code	
RTE_E_NO_DATA	No data transmitted, when the feedback API was attempted (non-blocking call only).
RTE_E_UNCONNECTED	Indicates that the sender port is not connected.
RTE_E_TIMEOUT	A timeout notification was received from the COM before any error notification (Inter-ECU only).
RTE_E_COM_STOPPED	The last transmission was rejected when either Rte_Send / Rte_Write API was called and the COM was stopped or an error notification from the COM was received before any timeout notification (Inter-ECU only).
RTE_E_TRANSMIT_ACK	A “transmission acknowledgement” has been received.
Existence	
This API exists, if the runnable entity of a SWC has configured explicit or implicit access to the data element in the DaVinci configuration of a runnable entity and the transmission acknowledgement is enabled at the communication specification and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`.	
Functional Description	
The implementation of the `Rte_Rips_<pi>_Feedback_[<ctp>][<pa>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_Feedback_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

Note
In case of an explicit access the ComponentTypePrototype `<ctp>` exists instead of the Ecuc Partition name `<pa>` for the function prototype.

Note
In case of an implicit access the Ecuc Partition name `<pa>` exists instead of the ComponentTypePrototype `<ctp>` for the function prototype.

4.17.13 Rte_Rips_DataIsUpdated

Prototype	
`boolean` **`Rte_Rips_<pi>_DataIsUpdated_[<ctp>]_<cgi>`** `( void )`	
Parameter	
-	
Return code	
TRUE	Data element has been updated since last read.
FALSE	Data element has not been updated since last read.
Existence	
This API exists, if the runnable entity of a SWC has configured explicit or implicit access to the data element in the DaVinci configuration of a runnable entity and the `EnableUpdate` attribute is enabled at the communication specification and there exists a FlatInstanceDescriptor referencing the data element within a Data Communication Graph and a Cross Cluster Plugin. The Plugin has set the Global Copy Instantiation Policy to `RTE_RIPS_INSTANTIATION_BY_PLUGIN`.	
Functional Description	
The implementation of the `Rte_Rips_<pi>_DataIsUpdated_[<ctp>]_<cgi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_IsUpdated_<p>_<d>()` instead of the body generated by the MICROSAR Classic RTE generator for the data element within the Data Communication Graph referenced by the FlatInstanceDescriptor.	

4.17.14 Rte_Rips_FillFlushRoutine

Prototype	
`void` **`<Fill-Flush-Routine-Symbol>`** `( void )`	
Parameter	
-	
Return code	
-	
Existence	
This API exists, if a Local Cluster Plugin has a Fill-Flush-Routine configured.	
Functional Description	
The implementation of the Fill-Flush-Routine must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
The call context of the Fill-Flush-Routine depends on the task mapping. All Fill-Flush-Routines are invoked by the RTE in the context of the task the routines are mapped to.	

4.17.15 Rte_Rips_Switch

Prototype	
`void` **`Rte_Rips_<pi>_Switch_<ctp>_<mmi>`** `( IN Rte_Rips_SwitchNotificationStatusType switchNotificationStatus, IN uint32 previousmode, IN uint32 nextmode )`	
Parameter	
switchNotificationStatus	Status of the enqueue operation.
previousmode	Value of the ModeDeclaration of the mode being left.
nextmode	Value of the ModeDeclaration of the mode being entered.
Return code	
-	
Existence	
This API exists, if the runnable entity of a SWC has configured access to the mode declaration group prototype in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the mode declaration group within a Mode Communication Graph and a Cross Cluster Plugin. Additionally, the mode declaration group is referenced by a RteModeMachineInstance.	
Functional Description	
The implementation of the `Rte_Rips_<pi>_Switch_<ctp>_<mmi>()` function must be provided and will not be generated by the MICROSAR Classic RTE generator.	
Call Context	
This function is called inside the body of the generated `Rte_Switch_<p>_<d>()` by the MICROSAR Classic RTE generator for the mode declaration group within the Mode Communication Graph referenced by the FlatInstanceDescriptor.	

4.17.16 Rte_Rips_DequeueModeSwitch

Prototype	
`Rte_Rips_SwitchNotificationStatusType` **`Rte_Rips_<pi>_DequeueModeSwitch_<mmi>_<t>`** `( void )`	
Parameter	
-	
Return code	
Rte_Rips_SwitchNotificationStatusType	Indicates the status of the dequeue operation in a mode queue.
Existence	
This API exists, if the runnable entity of a SWC has configured access to the mode declaration group prototype in the DaVinci configuration and there exists a FlatInstanceDescriptor referencing the mode declaration group within a Mode Communication Graph and a Cross Cluster Plugin. Additionally, the mode declaration group is referenced by a RteModeMachineInstance.	
Functional Description	
The function `Rte_Rips_<pi>_DequeueModeSwitch_<mmi>_<t>()` dequeues a mode switch notification from the mode queue when it is called after the last on-entry ExecutableEntity is terminated.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC). This function must not be called with locked interrupts.	

4.18 Logical Execution Time

The Logical Execution Time (LET) is an abstraction of physical – actual time of real-time Program.

LET defines time intervals in which runnables exchange its data within. When a Runnable is mapped to a LET interval all its implicit Sender/Receiver data will be exchanged inside the time interval.

LET is supported for implicit Sender/Receiver communication also for PRPort. However if PRPort configured with LET in an “Object code” implementation code type SWC the PRPort must enable API usage by address. As API usage by address is not allowed for multiple instantiated SWCs, PR ports are not supported in combination with multiple instantiation.

LET is enabled with the parameter /MICROSAR/Rte/RteGeneration/RteEnableLET.

The RTE generates for every defined LET interval two schedulable entities Release and Terminate.

The RTE generator supports two different LET communication semantics. The default is the classic communication semantic in which runnables have separate implicit buffers and updates to the buffers are reflected in the next iteration of the LET interval. By activating the /MICROSAR/Rte/RteGeneration/UseRteSemanticsForIntraLetCommunication it is possible to use the implicit RTE communication semantic in which runnables, which are mapped to the same LET interval and access the same elements, share an implicit buffer and updates to the buffers are reflected immediately. In both cases the RTE generator tries to minimize the number of generated buffers, if possible.

4.18.1 Rte_LetFrame_Release

Prototype	
`void` **`Rte_LetFrame_<IntervalName>_<OsApplicationName>_<CoreName>_Release`** `( void )`	
Parameter	
-	
Return code	
-	
Existence	
This API exists, if a LET is configured.	
Functional Description	
The implementation of the Rte_LetFrame_Release will be generated by the MICROSAR Classic RTE generator. All implicit data of runnables mapped to the let interval will be read in the Rte_LetFrame_Release.	
Call Context	
The call context of the Rte_LetFrame_Release depends on the task mapping.	

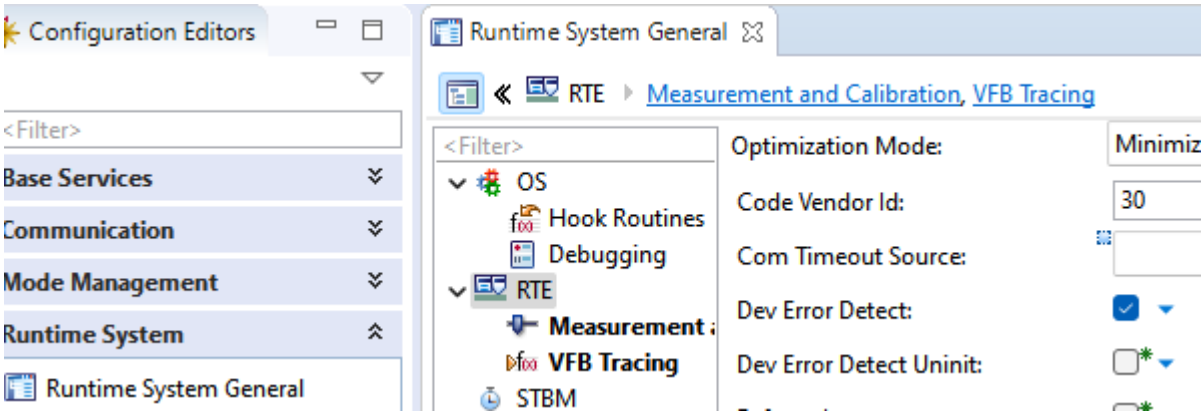
4.18.2 Rte_LetFrame_Terminate

Prototype
`void Rte_LetFrame_<IntervalName>_<OsApplicationName>_<CoreName>_Terminate ( void )`
Parameter
-
Return code
-
Existence
This API exists, if a LET is configured.
Functional Description
The implementation of the Rte_LetFrame_Terminate will be generated by the MICROSAR Classic RTE generator. All implicit data of runnables mapped to the let interval will be written in the Rte_LetFrame_Terminate.
Call Context
The call context of the Rte_LetFrame_Terminate depends on the task mapping.

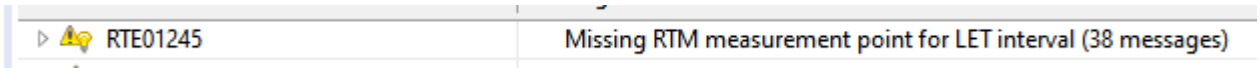
4.18.3 LET Online Monitoring

For monitoring the execution of LET frames, the RTE features online monitoring to detect violations in the execution order Rte_LetFrame_Release, Runnables (running within LET) and Rte_LetFrame_Terminate.

The LET Online Monitoring is activated if LET is configured and RTM module is present along with the parameter /MICROSAR/Rte/RteGeneration/RteDevErrorDetect



The LET online monitoring uses the RTM module to measure the execution time of the interval and the RTE detect possible execution order violations that could happen during execution and passes them on to the RTM module so that they can be reported. For these measurements the RTM needs measurement point containers for the LET intervals. The RTE validates if the measurement point containers for the configured LET frames are available or not.



The validation offers solving actions to create the measurement points that shall be executed in order to detect the violation.

The RTM creates a generation report that contains the measurement result for the LET execution.

In case a violation has occurred, the first error will be shown in the report. The messages in the report are declared as follows:

LET Execution Violation code	
RTE_E_OK	LET interval executed successfully
RTE_LET_E_INTERVAL_EXCEEDED	LET interval exceeded the interval time
RTE_LET_E_UNEXPECTED_RELEASE	Duplicated release call
RTE_LET_E_UNEXPECTED_TERMINATE	Duplicated terminate call
RTE_LET_E_UNEXPECTED_RUNNABLE_EXECUTION	Runnable called to often or not often enough

If online monitoring is enabled, a runnable is only allowed to be executed once within a LET frame. Multiple invocations or no invocations will lead to RTE_LET_E_UNEXPECTED_RUNNABLE_EXECUTION.

Caution

Online monitoring currently does not supervise that the release and terminate entities are activated in time and terminate entities of other intervals are finished before the release starts. This needs to be ensured by an appropriate system design considering task priorities, periods, activation offsets, position in task, worst case execution time etc.

Online monitoring only ensures the sequence and that the time between the start of the release entity and the end of the terminate entity does not exceed the period for the individual LET intervals.

Please note that the detection for the interval end happens inside the terminate entity, not when the interval is supposed to end according to the configuration (Supervision with e.g. alarms would require the activation of a task to handle violations. If the system is so overloaded that terminate is not called in time also the error handling task would also not be triggered). Therefore, other release entities may already have started to read the old output values without considering the new output values.

In case a violation is detected, flushing and filling of this LET interval is skipped until the order is as expected again.

If a custom handler shall be implemented in the application instead of the RTM module one can implement a callback function and define Rte_LET_ReportError to this handler in Rte_UserTypes.h

```
void MyHandler(uint32  measurementPoint, uint8 error);
#define Rte_LET_ReportError MyHandler
```

4.18.4 Scheduling of LET entities and runnables

If LET is used together with a MICROSAR Classic Operating System, the entire handling of the runnable and LET entity calls can be implemented with basic tasks that are triggered by a single schedule table per core.

For this, the task type of the involved tasks needs to be set to basic in the operating system configuration.

In order to synchronize the start of the schedule tables, it is possible to configure a barrier with the name Rte_LET_Barrier in the operating system configuration. The tasks that need to be configured for the barrier need to be the tasks that call the RTE initialization functions that start the schedule tables on the individual cores (SchM_Init/SchM_StartTiming).

Beware that other tasks, interrupt locks or interrupts themselves may delay the LET entities.

These delays need to be considered when configuring the activation offsets of the runnable entities and LET entities.

Thus it is required that the release and terminate entites are mapped to tasks that cannot interrupt each other and that have a higher priority than the tasks with the runnable entities.

If release and terminate entities are mapped to the same task it should be assured that the terminate entities are mapped before release so that flushing always happens before the next release.

4.19 Nv Sub-Element Mapping for NvBlockDataMapping

The RTE generator supports the mapping of nested Sub-Elements in Nv RAM Blocks to multiple Nv Data Prototypes.

The feature is automatically enabled when a NvBlock data mapping references multiple sub elements and DiagXf is configured.

Caution
The feature is active only for connections to NvBlockSWCs that use DiagXf

4.20 RTE Interfaces to BSW

The RTE has standardized Interfaces to the following basic software modules

- COM / LDCOM
- Transformer (COMXF, SOMEIPXF, DIAGXf, E2EXF)
- NVM
- DET
- OS
- XCP
- RTM
- SCHM

The actual used API's of these BSW modules depend on the configuration of the RTE.

4.20.1 Interface to COM / LDCOM

Used COM API
Com_SendSignal
Com_SendDynSignal
Com_SendSignalGroup
Com_SendSignalGroupArray
Com_UpdateShadowSignal
Com_ReceiveSignal
Com_ReceiveDynSignal
Com_ReceiveSignalGroup
Com_ReceiveSignalGroupArray
Com_ReceiveShadowSignal
Com_InvalidateSignal
Com_InvalidateSignalGroup

Used LDCOM API
LdCom_IfTransmit (early versions of MICROSAR Classic LDCOM) LdCom_Transmit

The RTE generator provides COM / LDCOM callback functions for signal notifications. The generated callbacks, which are called inside the COM layer, have to be configured in the COM / LDCOM configuration accordingly. The necessary callbacks are defined in the `Rte_Cbk.h` header file.

Caution
The RTE generator assumes that the context of COM / LDCOM callbacks is either a task context or an interrupt context of category 2.
It is explicitly NOT allowed that the call context of a COM / LDCOM callback is an interrupt of category 1. Moreover COM, LDCOM and Rte_ComSendSignalProxyPeriodic need to be mapped to the same partition. This partition is implicitly configured by mapping the COM mainfunctions to a task of this partition.

In order to access the COM / LDCOM API the generated RTE includes the `Com.h/LdCom.h` header file if necessary.

During export of the ECU configuration description the necessary COM / LDCOM callbacks are exported into the COM / LDCOM section of the ECU configuration description.

4.20.2 Interface to Transformer

Used Transformer API
ComXf_<transformerId>
ComXf_Inv_<transformerId>
SomeIpXf_<transformerId>
SomeIpXf_Inv_<transformerId>
DiagXf_<transformerId>
DiagXf_Inv_<transformerId>
E2EXf_<transformerId>
E2EXf_Inv_<transformerId>

Caution
The RTE generator does not support configurable transformer chains. Only the SomeIpXf and the ComXf are supported as first transformer in the chain. The E2EXf as second transformer is optional dependent on the configuration.
The DiagXf can only be used for Intra-Ecu S/R communication between composite data elements of application or Nv-Ram SWCs and the DCM module which uses a byte array data element.

4.20.3 Interface to OS

In general, the RTE may use all available OS API functions to provide the RTE functionality to the software components. The following table contains a list of used OS APIs of the current RTE implementation.

Used OS API
SetRelAlarm
SetAbsAlarm
CancelAlarm
StartScheduleTableRel
NextScheduleTable
StopScheduleTable
SetEvent
GetEvent
ClearEvent
WaitEvent
GetTaskID
GetCoreID
ActivateTask
Schedule
TerminateTask
ChainTask
GetResource
ReleaseResource
GetSpinlock
ReleaseSpinlock
DisableAllInterrupts
EnableAllInterrupts
SuspendAllInterrupts
ResumeAllInterrupts
SuspendOSInterrupts
ResumeOSInterrupts
CallTrustedFunction (MICROSAR Classic OS specific)
IocWrite
IocRead
IocWriteGroup
IocReadGroup
IocSend
IocReceive
osDisableLevelKM (MICROSAR Classic OS specific)
osEnableLevelKM (MICROSAR Classic OS specific)

Used OS API
osDisableGlobalKM (MICROSAR Classic OS specific)
osEnableGlobalKM (MICROSAR Classic OS specific)
osDisableLevelUM (MICROSAR Classic OS specific)
osEnableLevelUM (MICROSAR Classic OS specific)
osDisableGlobalUM (MICROSAR Classic OS specific)
osEnableGlobalUM (MICROSAR Classic OS specific)
osDisableLevelAM (MICROSAR Classic OS specific)
osEnableLevelAM (MICROSAR Classic OS specific)
osDisableGlobalAM (MICROSAR Classic OS specific)
osEnableGlobalAM (MICROSAR Classic OS specific)

In order to access the OS API the generated RTE includes the `Os.h` header file.

The OS configuration needed by the RTE is stored in the file `Rte_Needs.ecuc.arxml` which is created during the RTE Generation Phase.

For legacy systems the OS configuration is also stored in `Rte.oil`. This file is an incomplete OIL file and contains only the RTE relevant configuration. It should be included in an OIL file used for the OS configuration of the whole ECU.

Caution
The generated files `Rte_Needs.ecuc.arxml` and `Rte.oil` file must not be changed!

Caution
The data consistency APIs from the operating system need to implement a sequentially consistent acquire and release fence/compiler barrier to prevent memory reordering of any read or write which precedes them in program order with any read or write which follows them in program order.

Caution
To detect errors in the OS, the error hook in the operating system needs to be enabled, as the RTE does not provide additional callouts for failing OS API calls.

Be aware, that the RTE needs to start and stop OS alarms in its APIs for timeout monitoring.

If CancelAlarm is called with an expired alarm, the OS enters the error hook.

Rationale: As the OS API CancelAlarm cannot be called with locked interrupts, the AUTOSAR operating system offers no possibility to atomically check if an alarm is still running and to cancel it in this case.

Therefore it cannot be prevented that the OS enters the error hook due to cancelling already stopped alarms.

This happens even if no runtime error occured.

Thus the error E_OS_NOFUNC for cancelling already stopped RTE alarms and schedule tables needs to be ignored in the error hook by user implementation.

4.20.4 Interface to NVM

Used NVM API
NVM_WriteBlock
NVM_SetRamBlockStatus
NVM_GetErrorStatus

The RTE generator provides NVM callback functions for synchronous copying of the mirror buffers to and from the NVM. The generated callbacks, which are called inside the `NVM_MainFunction`, have to be configured in the NVM configuration accordingly. The necessary callbacks are defined in the `Rte_Cbk.h` header file.

Caution
The RTE generator assumes that the call context of NVM callbacks is the task which calls the `NVM_MainFunction`.

During export of the ECU configuration description the necessary NVM callbacks are exported into the NVM section of the ECU configuration description.

4.20.5 Interface to XCP

In addition to the usage of the Com and the OS module as described by AUTOSAR, the MICROSAR Classic RTE generator optionally can also take advantage of the MICROSAR Classic XCP module.

This makes it possible to configure the RTE to trigger XCP Events when certain measurement points are reached.

This for example also allows the measurement of buffers for implicit sender/receiver communication when a runnable entity is terminated.

Measurement is described in detail in chapter 5.7 Measurement and Calibration.

When measurement with XCP Events is enabled, the RTE therefore includes the header `Xcp.h` and calls the Xcp_Event API to trigger the events.

Used Xcp API
Xcp_Event

4.20.6 Interfaces to RTM

The MICROSAR Classic RTE generators supports the RTM modules for monitoring the duration of LET intervals. See chapter 4.18.3 for details.

Used Rtm API
Rtm_Start
Rtm_Stop
Rtm_TriggerReading

4.20.7 Interface to SCHM

In multicore and memory protection systems, the schedulable entity `Rte_ComSendSignalProxyPeriodic` is provided by the RTE and is used to access the COM from OS Applications without BSW. This schedulable entity needs to be called periodically by the SCHM.

See chapter 3.8.1 for details.

Provided Schedulable Entity
Rte_ComSendSignalProxyPeriodic

4.20.8 Interface to DET

The RTE generator reports development errors to the DET, if development error detection is enabled.

See chapter 2.8.1 for details.

Used DET API
Det_ReportError

4.21 Direct connection between Calibration and Sender-Receiver Ports

The RTE supports direct communication between a calibration sender port and receiver port. The used calibration parameter DataType shall be the same as the DataElement DataType in the connected receiver port. Accesses for the receiver port also need to be configured.

5 RTE Configuration

The RTE specific configuration in DaVinci Configurator encompasses the following parts:

- assignment of runnables to OS tasks
- assignment of OS tasks to OS Applications (memory protection/multicore support)
- assignment of Per-Instance Memory to NV memory blocks
- selection of the exclusive area implementation method
- configuration of the periodic triggers
- configuration of measurement and calibration
- selection of the optimization mode
- selection of required VFB tracing callback functions
- configuration of the built-in call to the RTE generator
- platform dependent resource calculation

5.1 Configuration Variants

The RTE supports the configuration variants

- `VARIANT-PRE-COMPILE`
- `VARIANT-POST-BUILD-SELECTABLE`

The configuration classes of the RTE parameters depend on the supported configuration variants. For their definitions please see the `Rte_bswmd.arxml` file.

5.2 Lifecycle Configuration

In the standard configuration, the RTE activates the triggering of cyclic and background runnables in `Rte_Start`. The `Rte_StartTiming` API is disabled and not emitted into the generated code. If it is required to postpone the activation of cyclic and background runnables you can enable the `Rte_StartTiming` API. The API is enabled when at least one `RteInitializationRunnableBatch` exists. Note that (at this point) a `RteInitializationRunnableBatch` only serves as a proxy for enabling the `Rte_StartTiming` API, and cannot be used to trigger `RunnableEntitiy`s for initialization.

5.3 Task Configuration

Runnable Entities triggered by any kind of RTE Event e.g. `TimingEvent` have to be mapped to tasks. Only server runnables (triggered by an `OperationInvokedEvent`) that either have their `CanBeInvokedConcurrently` flag enabled or that are called from tasks that cannot interrupt each other do not need to be mapped. For optimization purposes they can be called directly and are then executed in the context of the calling runnable (client).

The task configuration within DaVinci Configurator also contains some attributes which are part of the OS configuration. The parameters are required to control the RTE generation. The creation of tasks is done in the OS Configuration Editor in the DaVinci Configurator. The **Task Mapping Assistant** has to be used to assign the triggered functions (runnables and schedulable entities) to the tasks.

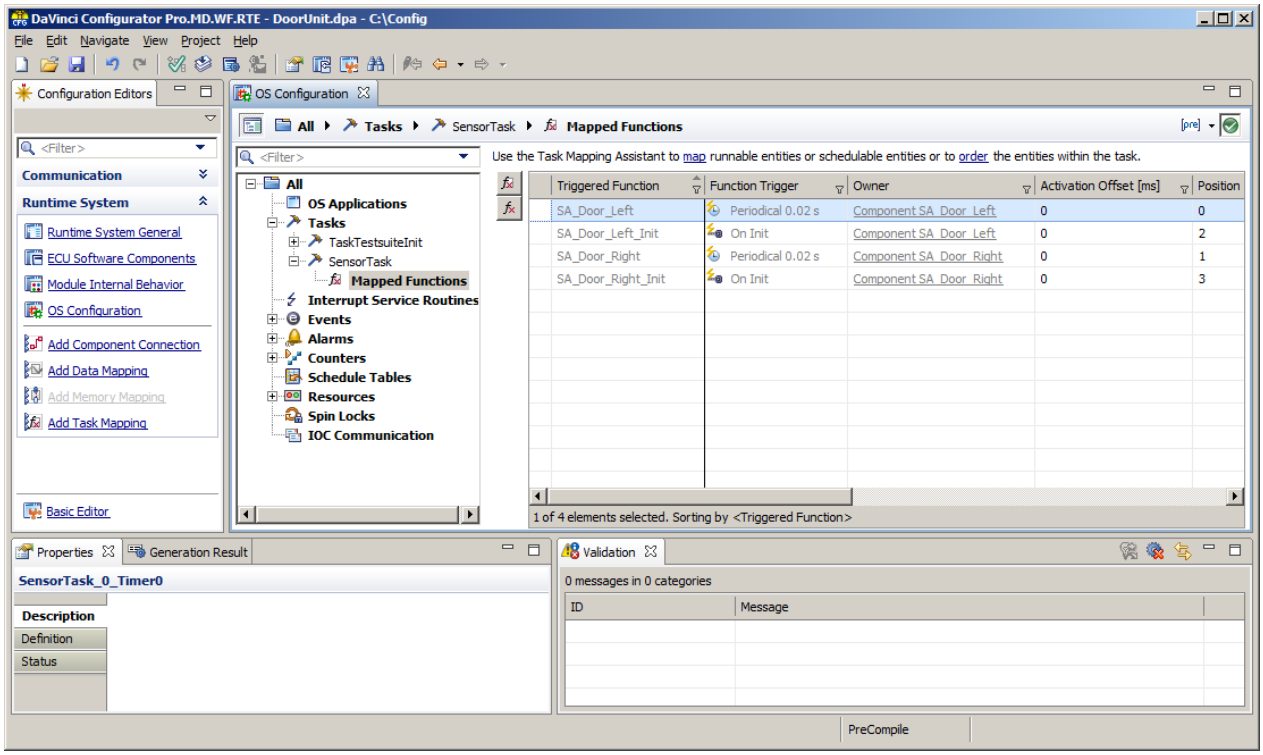


Figure 5-1 Mapping of Runnables to Tasks

The MICROSAR Classic RTE supports the generation of both `BASIC` and `EXTENDED` tasks. The Task Type can either be selected or the selection is done automatically if `AUTO` is configured.

A basic task can be used when all runnables of the task are triggered by one or more identical triggers.

A typical example for this might be several cyclic triggered runnables that share the same activation offset and cycle time.

There is also the possibility to select Task Type `BASIC` if all runnables of a task are triggered cyclically but have different cycle times or different activation offsets. The RTE realizes the

basic task with the help of the OS Schedule Tables. When the number of RTE events exceeds the number of OS events that are possible for a single extended or basic task, the RTE provides an internal event handling mechanism that maps multiple RTE events to a single OS event. The internal event handling is only possible for RTE events that are not triggered by alarms.

Moreover another prerequisite for basic task usage is that the mapped runnables do not use APIs that require a waitpoint, like a blocking `Rte_Feedback()`.

If the above described conditions are not fulfilled an extended task has to be used. The extended task can wait for different runnable trigger conditions e.g. data reception trigger, cyclic triggers or mode switch trigger.

For mode management the tasks with the mode switch triggers perform the actual mode switch. When there are no mode switch triggers and all runnables with mode disablings are triggered by timing events and are mapped to basic tasks, the actual mode switch is delayed until the entities with the disablings were triggered to avoid the generation of an additional mode switch event that would prevent the usage of basic tasks in this scenario.

Caution
When RTE events that trigger a runnable are fired multiple times before the actual runnable invocation happens and when the runnable is mapped to an extended task, the runnable is invoked only once.

However, if the runnable is mapped to a basic task, the same circumstances will cause multiple task activations and runnable invocations. Therefore, for basic tasks, the task attribute Activation in the OS configuration has to be set to the maximum number of queued task activations. If Activation is too small, additional task activations may result in runtime OS errors. To avoid the runtime error the number of possible Task Activation should be increased.

5.4 Memory Protection and Multicore Configuration

For memory protection or multicore support the tasks have to be assigned to OS Applications. The following figures show the configuration of OS Applications and the assignment of OS tasks. For multicore support also the Core ID has to be configured for the OS Application. When a runnable/trigger of a SWC is mapped to a task, the SWC is automatically assigned to the same OS Application as the task. In case the SWC only contains runnables that are not mapped to a task, the SWC can be assigned to an ECUC partition with the parameter `EcuC/EcucPartitionCollection/EcucPartition/EcucPartitionSoftwareComponentInstanceRef`. For every OS Application, an ECUC partition can be created. It then needs to be referenced by the OS Application with the `Os/OsApplication/OsAppEcucPartitionRef` parameter. Besides the assignment of SWCs to OS Applications, the ECUC partition provides a parameter to configure the safety level of the partition (QM or ASIL_A to ASIL_D). The RTE generator uses this parameter to enable additional task priority based optimizations for QM partitions. The same optimizations can also be enabled for ASIL partitions with the parameter *`Rte/RteGeneration/RteOptimizeLocksInASILPartitions`*.

Caution
RTE APIs are only allowed to be used from the runnables for which they are configured. If RTE APIs are called from the wrong tasks or are even called from ISRs, data consistency problems and OS errors can occur.

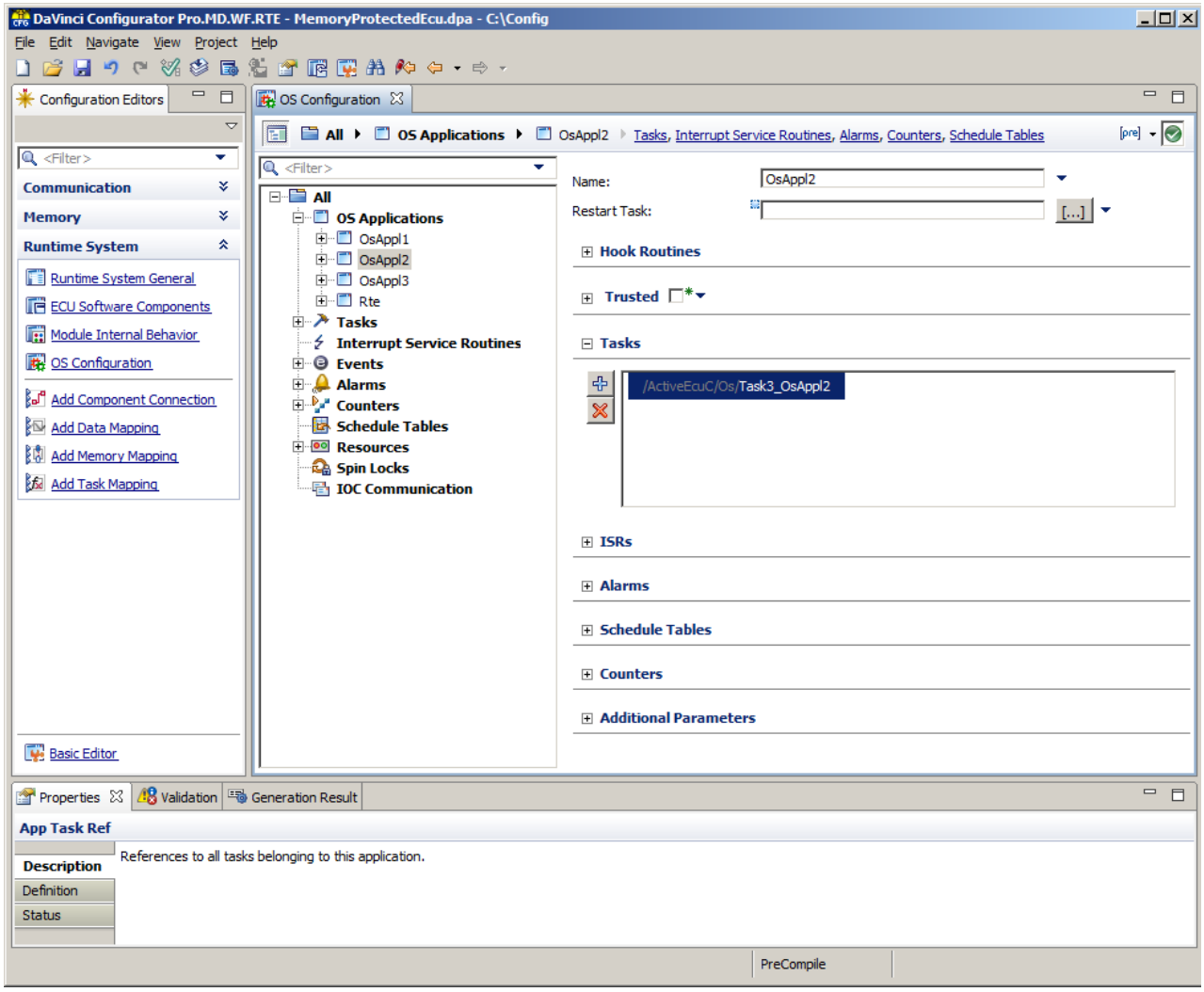


Figure 5-2 Assignment of a Task to an OS Application

Caution
Make sure that the operating system is configured with scalability class SC3 or SC4.

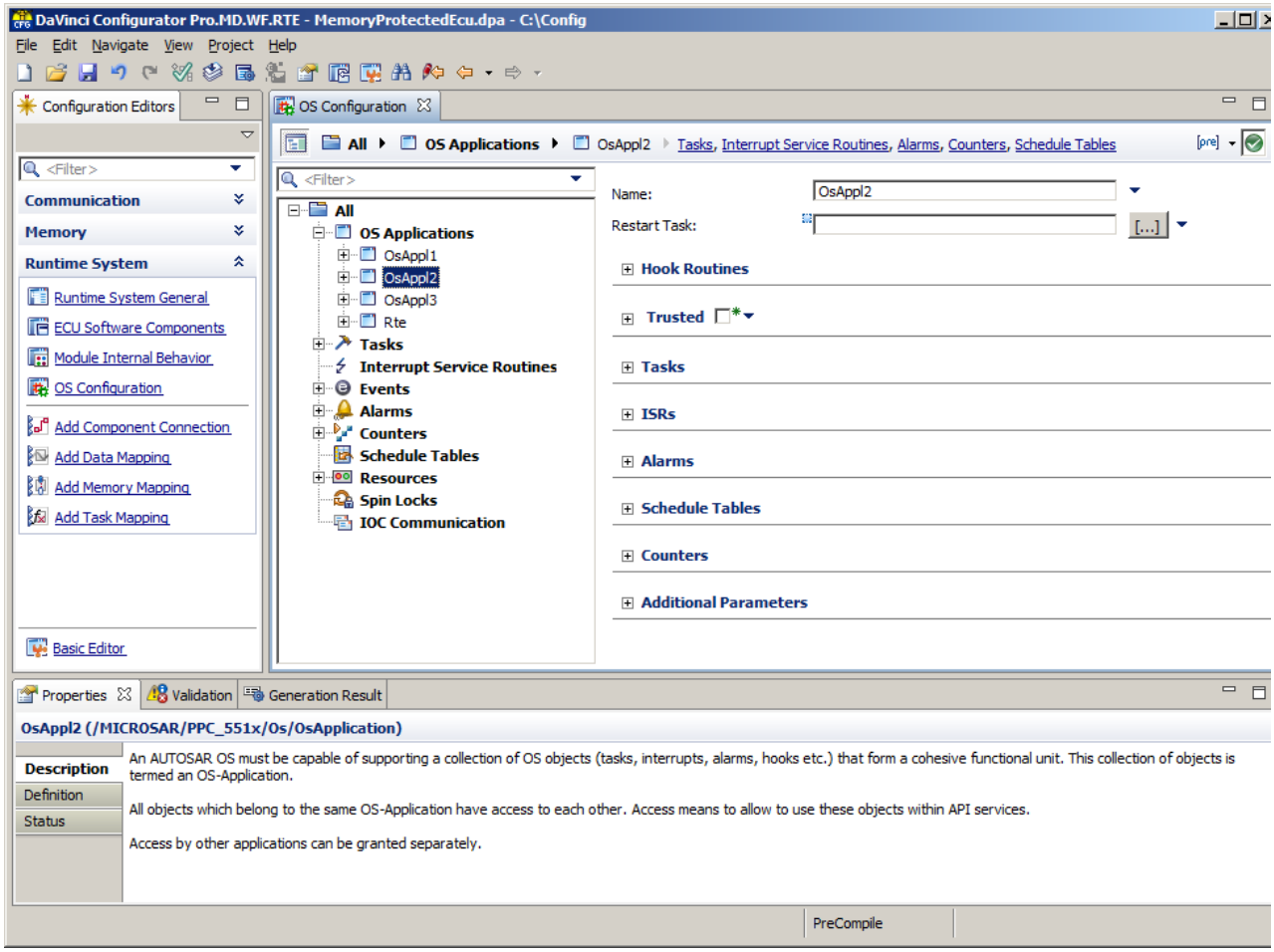


Figure 5-3 OS Application Configuration

5.4.1 Multicore NvM

The RTE supports multiple NvMs and forwards core local NvM requests to the core local NvM. The NvBlockDescriptors can be assigned to an ECUC partition with the Bswmd parameter `NvM/NvMBlockDescriptor/NvMBlockEcucPartitionRef`. The NvBlockSwc and its NvBlockDescriptors must be mapped to the same partition. The NvM can be accessed from each partition a NvM_MainFunction is mapped to.

5.5 NV Memory Mapping

Each instance of a Per-Instance Memory, which has configured **Needs memory mapping** can be mapped to an NV memory block of the NVM.

The Per-Instance Memory (PIM) is used as mirror buffer for the NV memory block. During startup, the EcuM calls `NvM_ReadAll`, which initializes the configured PIM with the value of the assigned NV memory block. During shutdown, `NvM_WriteAll` stores the current value of the PIM buffer in the corresponding NV memory block.

The RTE configurator provides support for manual mapping of already existing NV memory blocks or automatically generation of NV memory blocks and mapping for all PIMs.

The RTE has no direct Interface to the NVM in the source code. There exists only an Interface on configuration level. The RTE configurator will configure the following parts of the NVM configuration:

- Address of the PIM representing the RAM mirror of the NV memory block.
- Optionally: The address of calibration parameter for default values.
- Optionally, if available during configuration time: The size of the PIM in bytes.

The following figure shows the **Memory Mapping** in DaVinci Configurator where assignment of Per-Instance Memory to NV memory blocks can be configured.

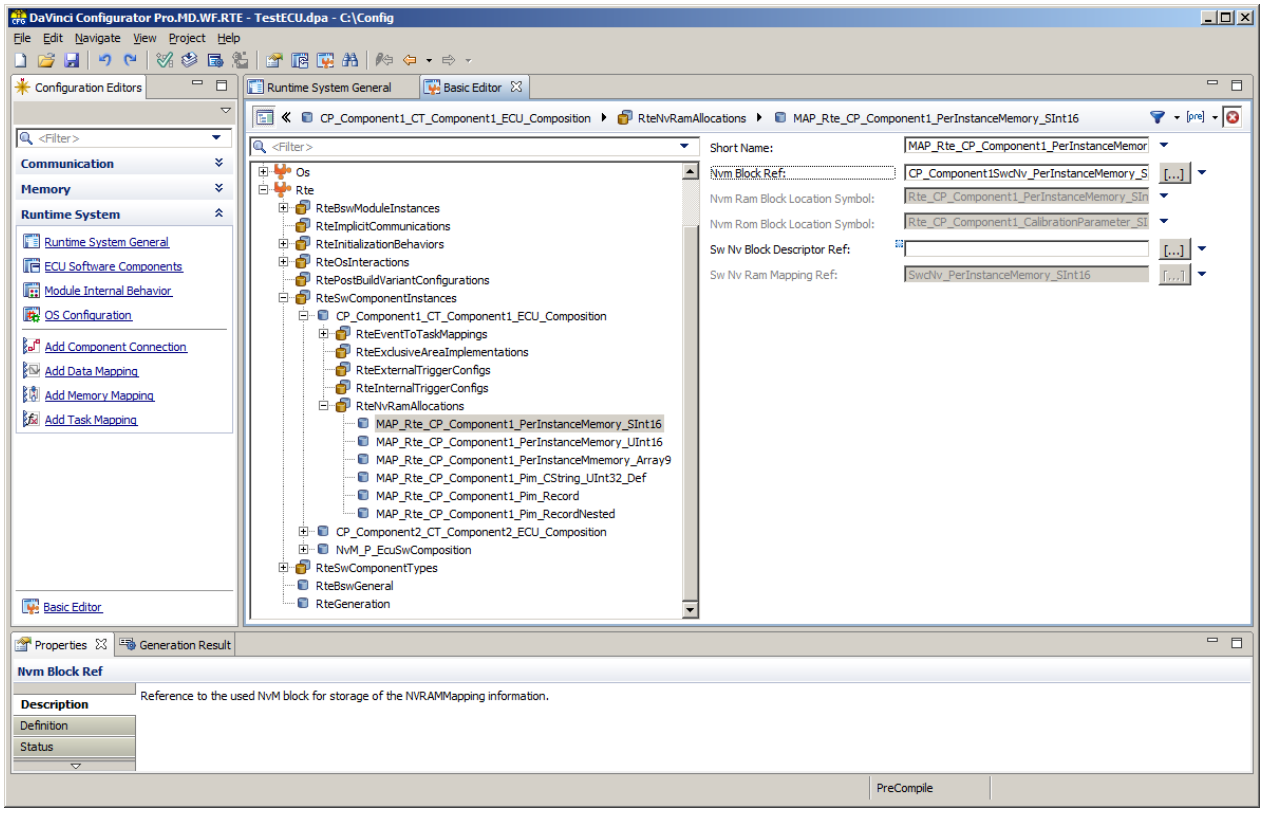


Figure 5-4 Mapping of Per-Instance Memory to NV Memory Blocks

5.6 RTE Generator Settings

The following figure shows how the MICROSAR Classic RTE Generator has to be enabled for code generation within the DaVinci Configurator.

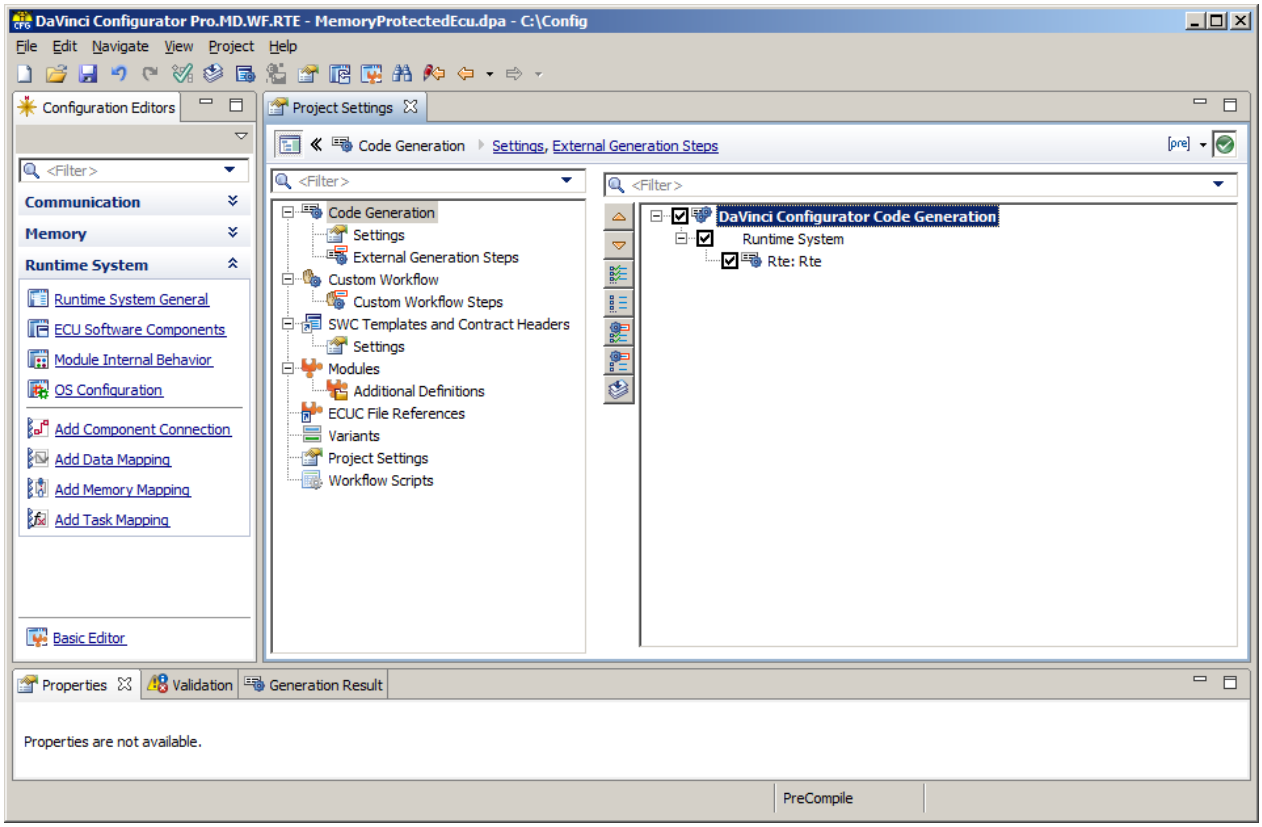


Figure 5-5 RTE Generator Settings

For RTE generation, the parameter /MICROSAR/Rte/RteGeneration/RteBasicRuntimeEnvironmentMode needs to be set to false.

5.6.1 RTE Submodules RteChecks and RteCalculation

In the validation window of each project an information that two RTE submodules are not configured will appear as shown in the figure below.

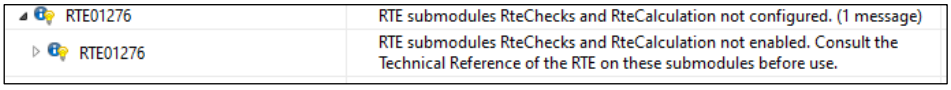


Figure 5-6 Validation message to enable the RTE submodules RteChecks and RteCalculcation.

Applying the solve action will enable the two submodules in the project. The two submodules will then appear as two new Generation Steps when generating the project.

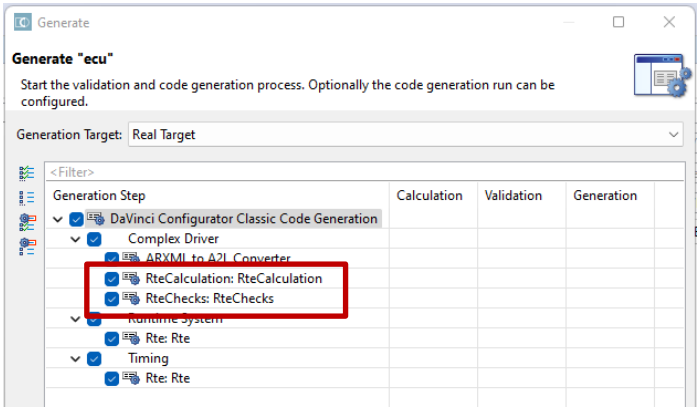


Figure 5-7 Generate menu with the two enabled submodules RteCalculation and RteChecks.

These two modules allow skipping certain phases of the RTE generation, potentially shortening the time needed to generate.

By deselecting

- RteCalculcation, the calculation phase of the RTE
- RteChecks, the developer checks of the RTE

will be skipped during the generation of the RTE.

Caution
These options can potentially break the generation alltogether.
Be aware that you fully need to understand if the model has not changed to skip generation steps.

5.7 Measurement and Calibration

The MICROSAR Classic RTE generator supports the generation of an ASAM MCD-2MC compatible description of the generated RTE that can be used for measurement and calibration purposes. When measurement or calibration is enabled the RTE generator generates a file `Rte.a2l` that contains measurement objects for sender/receiver ports, per-instance memory and inter-runnable variables. Calibration parameters are represented as characteristic objects. The RTE generator also generates A2L entries for constant and static memories.

Additionally, these objects are stored as McSupportData instances in the internal behavior of the RTE BSW module description file to allow A2L generation with the more advanced McDataConverter that is provided as part of CFG5.

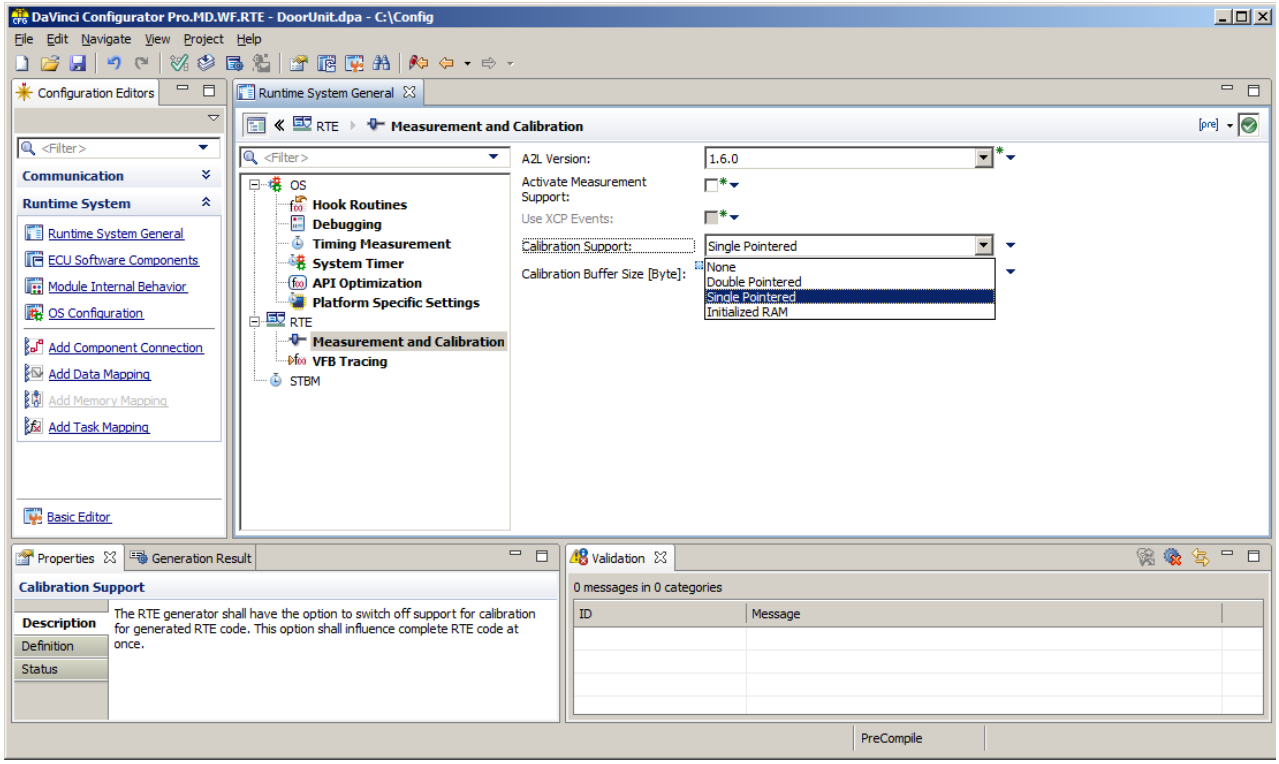


Figure 5-8 Measurement and Calibration Generation Parameters

The switch A2L Version controls the ASAM MCD-2MC standard to which the `Rte.a2l` file is compliant. Version 1.6.0 is recommended as it supports a symbol link attribute that can be used by the measurement and calibration tools to automatically obtain the address of a characteristic or measurement object in the compiled and linked RTE code.

What measurements and characteristics are listed in the `Rte.a2l` file and respectively the McSupportData section within the internal behavior file depends on the measurement and calibration settings of the individual port interfaces, per-instance memories, inter-runnable variables and calibration parameters and if the variable can be measured in general. For example, measurement is not possible for queued communication as described in the RTE

specification. When “Calibration Access” is set to “NotAccessible”, an object will not be listed in the `Rte.a2l` file or the McSupportData.

Within the `Rte.a2l` file, the measurement objects are grouped by SWCs. By default, the A2L groups are not generated to the McSupportData but an additional script RteMcGroups is provided with the generator and can be added as external generation step in CFG5.

Furthermore, the generated Rte.a2l is only a partial A2L file. It is meant to be included in the MODULE block of a skeleton A2L file with the ASAM MCD-2MC /include command.

This makes it possible to specify additional measurement objects, for example from the COM, and IF_DATA blocks directly in the surrounding A2L file.

In order to also allow the measurement of implicit buffers for inter-ECU communication, the MICROSAR Classic RTE generator supports measurement with the help of XCP Events. This is controlled by the flag “Use XCPEvents”. When XCP Events are enabled, the RTE generator triggers an XCP Event that measures the implicit buffer after a runnable with implicit inter-ECU communication is terminated and before the data is sent. “Use XCPEvents” also enables the generation of one XCP Event at the end of every task that can be used to trigger the measurement of other objects. If the XCP module does not provide an XCP Event API that is usable from multiple cores/partitions, the XCP event support in the RTE cannot be used for such configurations.

The RTE generator automatically adds the XCP Events to the configuration of the XCP module. The Event IDs are then automatically calculated by the XCP module.

The definitions for the Events are generated by the XCP module into the file `XCP_events.a2l`. This file can be included in the DAQ section of the IF_DATA XCP section in the skeleton A2L file.

The MICROSAR Classic RTE supports three different online calibration methods, which can be selected globally for the whole ECU. They differ in their kind how the APIs `Rte_CData` and `Rte_Prm` access the calibration parameter. By default, the online calibration is switched off. The following configuration values can be selected:

- None
- Single Pointered
- Double Pointered
- Initialized RAM

Online calibration is only possible for calibration parameters where “Calibration Access” is configured to ReadWrite.

In addition to the ECU global selection of the method the online calibration has to be activated for each component individually by setting the **Calibration Support** switch.

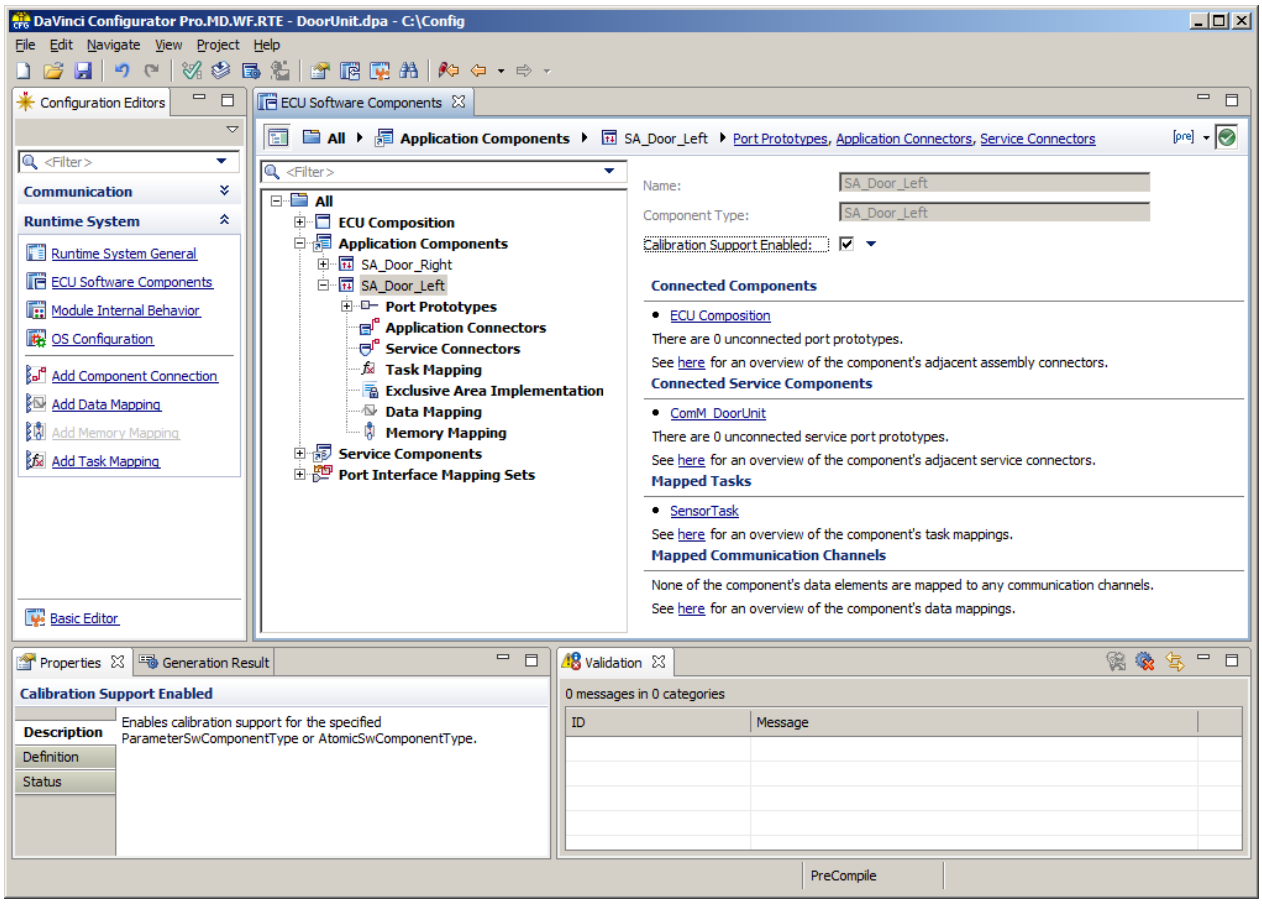


Figure 5-9 SWC Calibration Support Parameters

For each component with activated Calibration Support memory segments are generated into the file `Rte_MemSeg.a2l`. This file can be included in the MOD_PAR section in the skeleton A2L file. This makes it possible to specify additional memory segments in the surrounding A2L file.

If the method Initialized RAM is selected, segments for the Flash data section and the RAM data section of each calibration parameter are generated. The Flash sections are mapped to the corresponding RAM sections.

If the Single Pointered or Double Pointered method is enabled, only memory segments for the Flash data sections are listed in the `Rte_MemSeg.a2l`. In addition, a segment for a RAM buffer is generated, when the Single Pointered method is used and a `CalibrationBufferSize` is set. This parameter specifies the size of the RAM buffer in byte. If it is set to 0, no RAM buffer will be created.

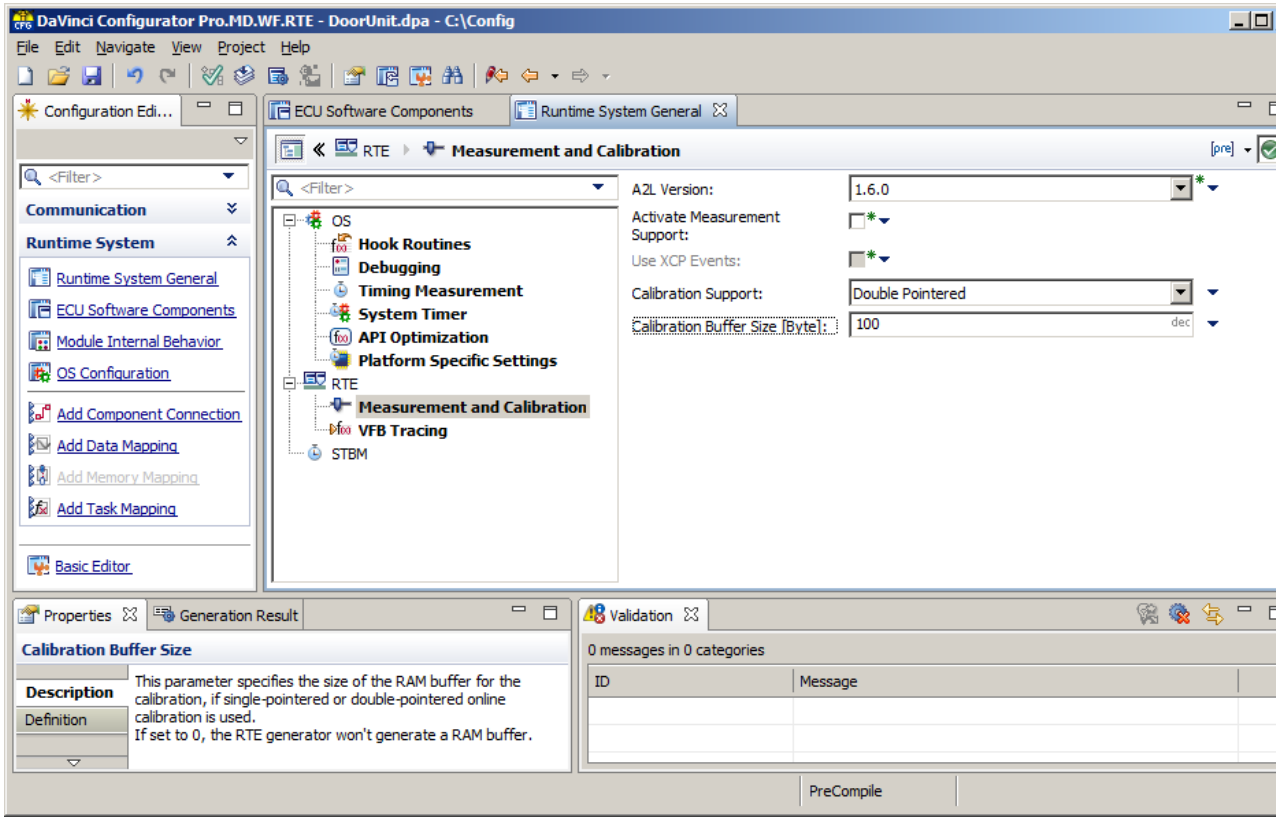


Figure 5-10 CalibrationBufferSize Parameter

The following figure shows a possible include structure of an A2L file. In addition to the fragment A2L files that are generated by the RTE generator other parts (e.g. generated by the BSW) can be included in the skeleton A2L file.

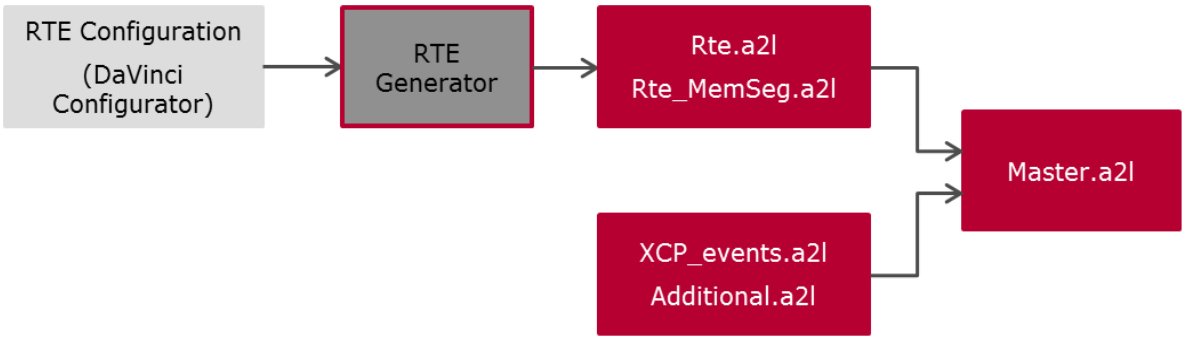


Figure 5-11 A2L Include Structure

If instead of Rte.a2l, McSupportData shall be used for A2L generation the inclusion of the RTE specific files can be skipped. McSupportData contains all the relevant measurement, calibration objects, xcp events and the configured OnlineCalibrationMethod.

For more details about the creation of a complete A2L file see [24].

Caution
A McSupportData instance for the measurement or calibration element is only generated if it is referenced by an existing FlatInstanceDescriptor. The RTE generator provides a live validator to create missing FlatInstanceDescriptors. The names of the FlatInstanceDescriptors are then used as names for the McDataInstances.

5.8 Optimization Mode Configuration

A general requirement to the RTE generator is production of optimized RTE code. If possible the MICROSAR Classic RTE Generator optimizes in different optimization directions at the same time. Nevertheless, sometimes it isn't possible to do that. In that case the default optimization direction is "Minimum RAM Consumption". The user can change this behavior by manually selection of the optimization mode.

- Minimum RAM Consumption (`MEMORY`)
- Minimum Execution Time (`RUNTIME`)

The following figure shows the **Optimization Mode** Configuration in DaVinci Configurator.

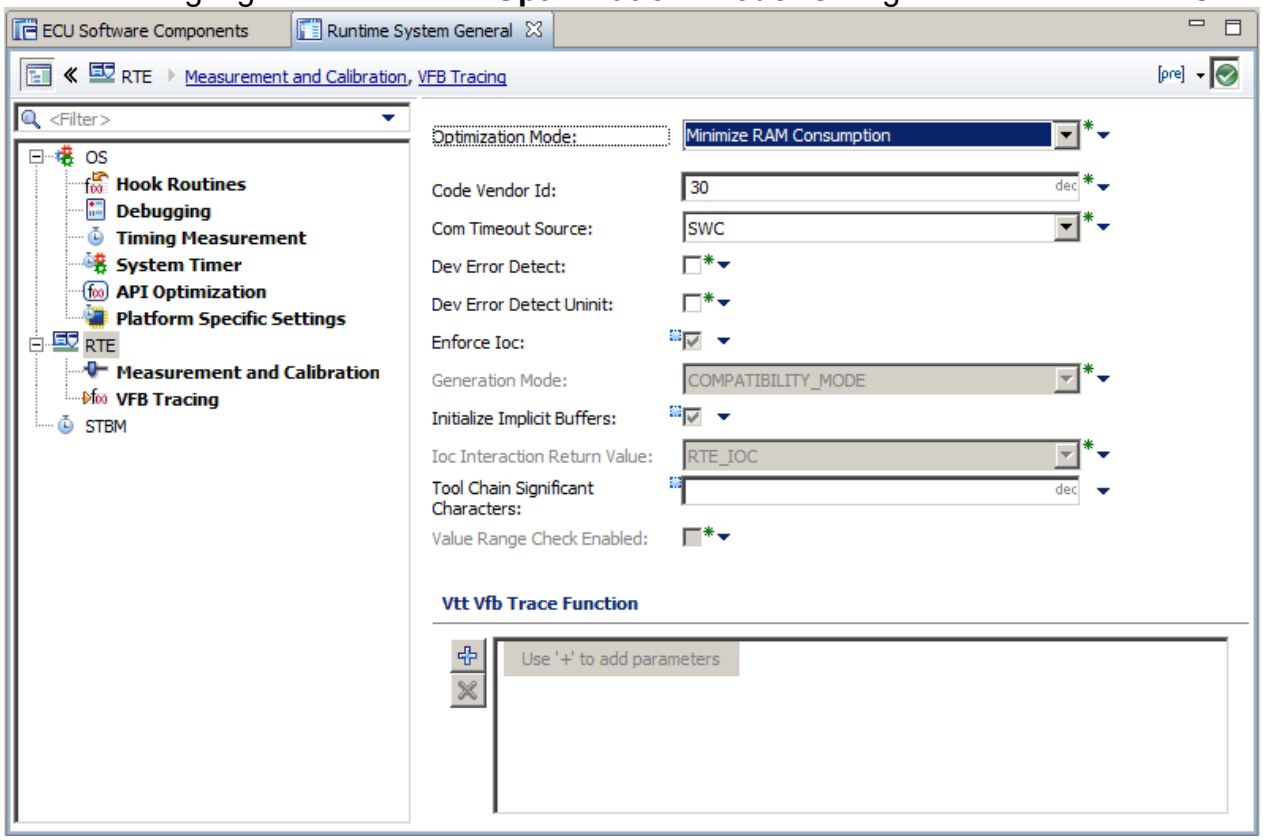


Figure 5-12 Optimization Mode Configuration

5.9 VFB Tracing Configuration

The MICROSAR Classic RTE supports the VFB Tracing feature for both the deprecated specification up until AUTOSAR 4.4 and the current specification from AUTOSAR 20-11 onwards.

5.9.1 AUTOSAR 4.4

The VFB Tracing feature of the MICROSAR Classic RTE may be enabled in the DaVinci Configurator as shown in the following picture.

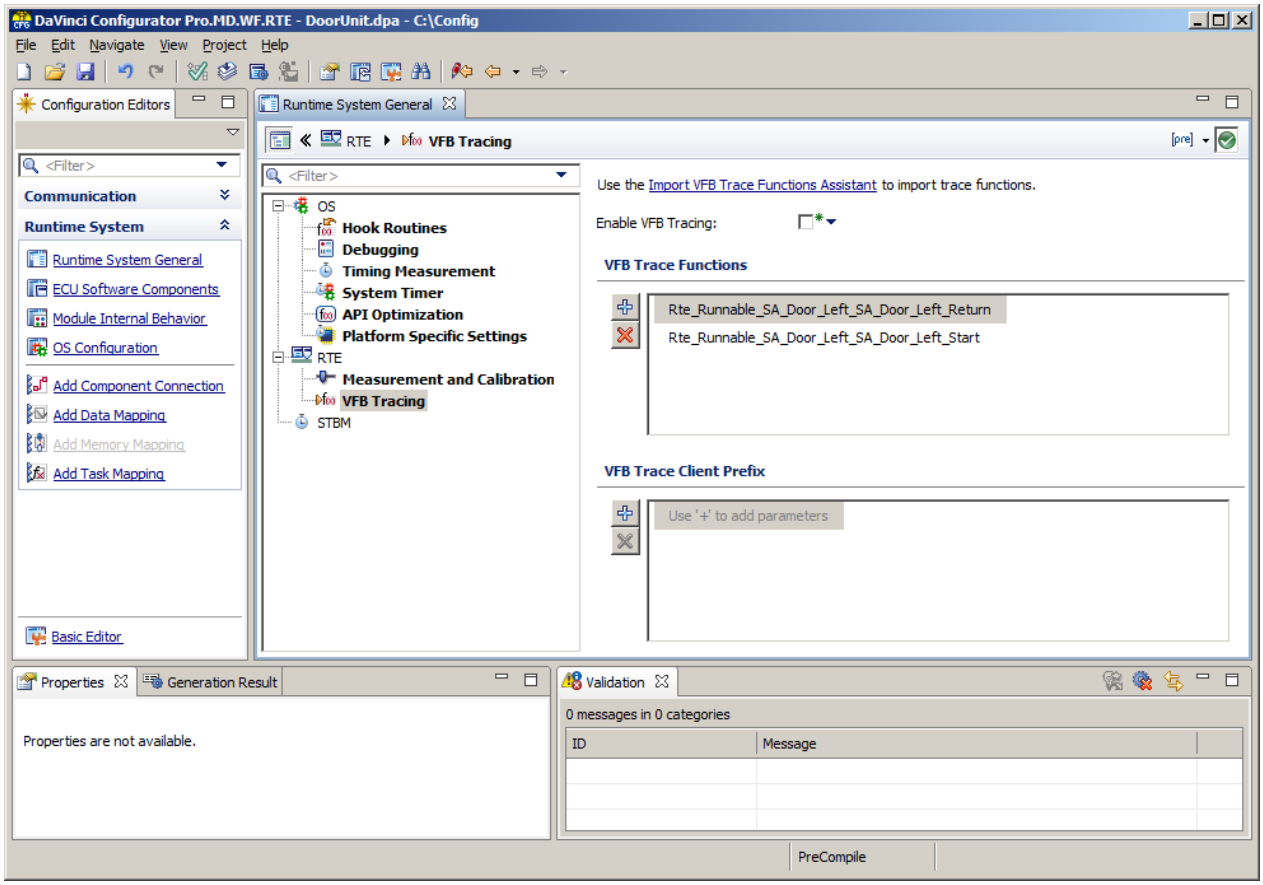


Figure 5-13 VFB Tracing Configuration

You may open an already generated `Rte_Hook.h` header file from within this dialog. This header file contains the complete list of all available trace hook functions, which can be activated independently. You can select and copy the names and insert these names into the trace function list of this dialog manually or you can import a complete list from a file. If you want to enable all trace functions, you can import the trace functions from an already generated `Rte_Hook.h`. The VFB Trace Client Prefix defines an additional prefix for all VFB trace functions to be generated. With this approach it is for example possible to additionally enable trace functions for debugging (Dbg) and diagnostic log and trace (Dlt) at the same time.

Info
All enabled trace functions have to be provided by the user. Section 3.3.4 describes how a template for VFB trace hooks can be generated initially or updated after configuration changes.

5.9.2 AUTOSAR 20-11

The VFB Tracing feature of the MICROSAR Classic RTE may be enabled in the DaVinci Configurator as shown in the following picture.

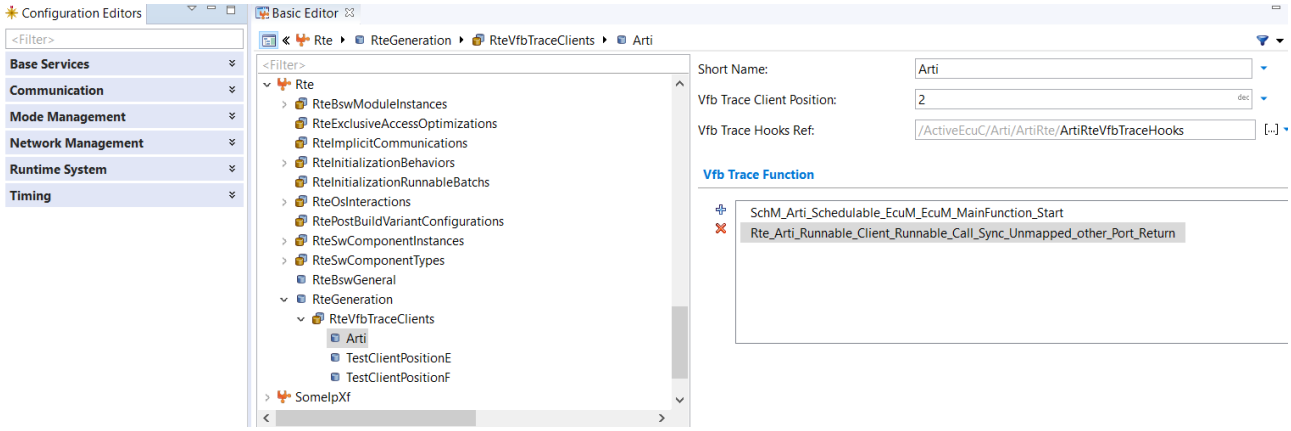


Figure 5-14 VFB Trace Client Configuration

To configure VFB trace client hooks you need to create a RteVfbTraceClient container. There you may add the hooks that shall be called by the RTE. In this container you may also configure a position for the VFB trace client, which will determine the order of the hooks, if there is more than one client container. Also, you may add a reference to a container, where the RTE shall place extended information about the created hooks.

Note
The VFB Trace Functions need to match a valid hook function name exactly to be generated. Auto matching of hooks to the beginning of a configured string is not performed.

Note
Only the extended information of hooks, that are generated and called by the RTE, will be placed in the referenced container, if such a reference is defined. Invalid hook definitions for the specific configuration will not cause the placement of extended information in the referenced container.

5.10 Exclusive Area Implementation

The implementation method for exclusive areas can be set in the DaVinci Configurator as shown in the following picture.

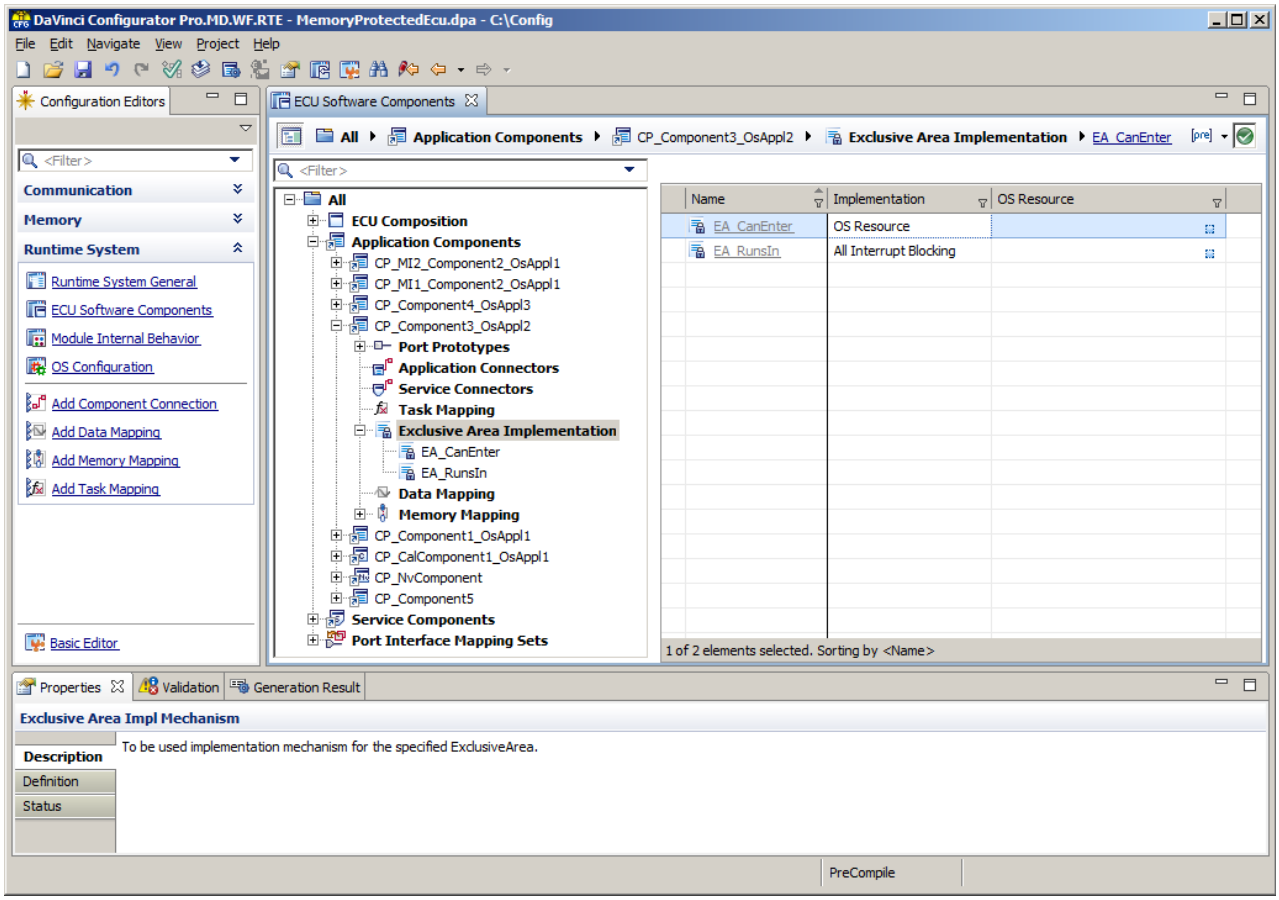


Figure 5-15 Exclusive Area Implementation Configuration

5.11 Periodic Trigger Implementation

The runnable activation offset and the trigger implementation for cyclic runnable entities may be set in the ECU project editor as shown in the following picture.

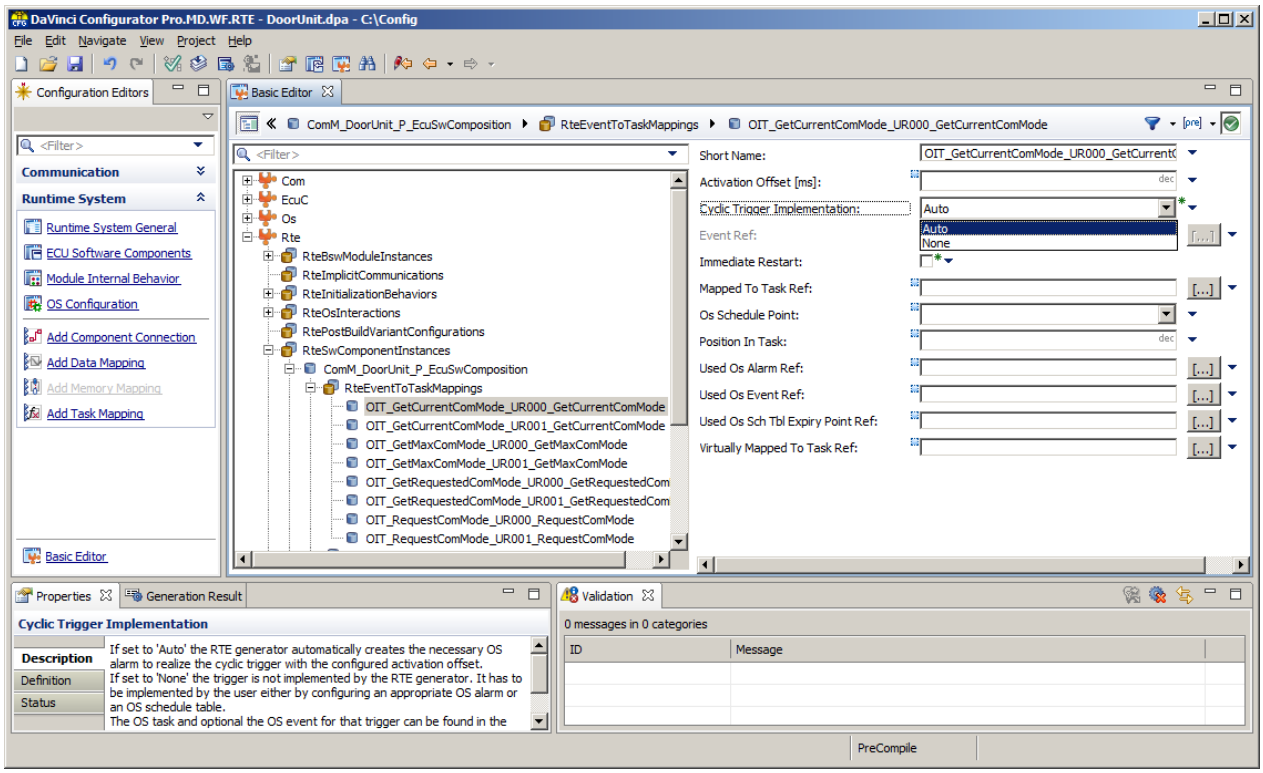


Figure 5-16 Periodic Trigger Implementation Configuration

Caution
Currently it is not supported to define an activation offset and a trigger implementation per trigger. The settings can only be made for the complete runnable with potential several cyclic triggers.

The activation offset specifies at what time relative to the start of the RTE the runnable / main function is triggered for the first time.

Caution
If a SchedulableEntity exists in the configuration, all activation offsets of cyclic Runnables triggered by a ScheduleTable are relative to SchM_Init, instead of Rte_Start or Rte_StartTiming.

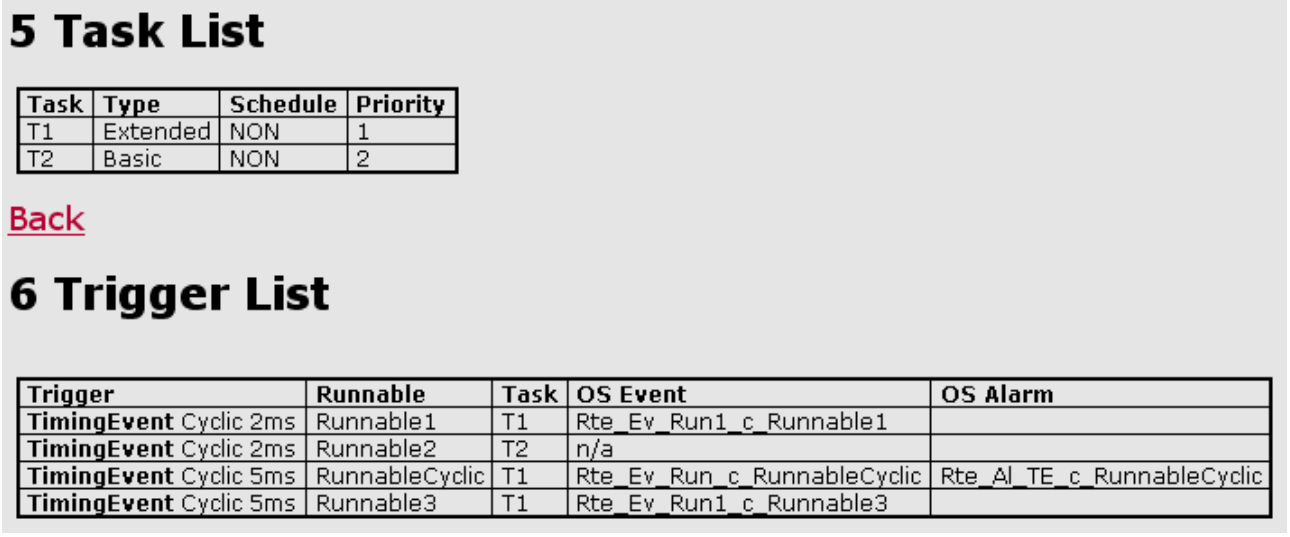
Trigger implementation can either be set to `Auto` or `None`. When it is set to the default setting `Auto`, the RTE generator will automatically generate and set OS alarms that will then trigger the runnables / main functions. Additionally, when all of the cyclically triggered runnables are mapped to a basic task, the RTE generator will configure an OS ScheduleTable, which is then responsible for triggering the task following a set of expiry points. The RTE will then control the triggering of runnables / main functions accordingly.

The same behavior of the RTE can also be achieved without an OS ScheduleTable. When the attribute OsTaskPeriod is set for a task, the RTE expects the task to be periodically triggered by the defined value and will behave the same way as described above. However, in this case the cycle times and activation offsets of the runnables / main functions have to be multiples of the OsTaskPeriod. To deactivate the creation of OS ScheduleTables and OS Alarms the attribute DisableOsScheduleTableAndAlarmGeneration has to be activated.

When trigger implementation is set to `None`, the RTE generator only creates the tasks and events for triggering the runnables / main functions. It is then the responsibility of the user to periodically activate the basic task to which a runnable / main function is mapped or to send an event when the runnable / main function is mapped to an extended task.

This feature can also be used to trigger cyclic runnable entities / main functions with a schedule table. This allows the synchronization with FlexRay.

To ease the creation of such a schedule table, the generated report `Rte.html` contains a trigger listing. The listing contains the triggered runnables / main functions, their tasks and the used events and alarms.



5 Task List

Task	Type	Schedule	Priority
T1	Extended	NON	1
T2	Basic	NON	2

Back

6 Trigger List

Trigger	Runnable	Task	OS Event	OS Alarm
TimingEvent Cyclic 2ms	Runnable1	T1	Rte_Ev_Run1_c_Runnable1	
TimingEvent Cyclic 2ms	Runnable2	T2	n/a	
TimingEvent Cyclic 5ms	RunnableCyclic	T1	Rte_Ev_Run_c_RunnableCyclic	Rte_Al_TE_c_RunnableCyclic
TimingEvent Cyclic 5ms	Runnable3	T1	Rte_Ev_Run1_c_Runnable3	

Figure 5-17 HTML Report

If the OS alarm column for a trigger is empty, the runnable / main function needs to be triggered manually. In the example above, this is the case for all runnables except for RunnableCyclic.

The row for Runnable2 does not contain an event because this runnable is mapped to a basic task.

To manually implement the cyclic triggers, one could for example create a repeating schedule table in the OS configuration with duration 10 that uses a counter with a tick time of one millisecond. An expiry point at offset 0 would then need to contain `SETEVENT` actions for the runnables Runnable1 and Runnable3 and an `ACTIVATETASK` action for Runnable2.

Moreover, further expiry points with the offsets 2, 4, 6, 8 are needed to activate Runnable1 and Runnable2 and another expiry point with offset 5 is needed to activate Runnable3.

Caution
When the trigger implementation is set to none, the settings for the cycle time and the activation offset are no longer taken into account by the RTE. It is then the responsibility of the user to periodically trigger the runnables / main functions at the configured times. Moreover, the user also has to make sure that this triggering does not happen before the RTE is completely started.

5.12 Resource Calculation

The RTE generator generates the file Rte.html containing the RAM and CONST usage of the generated RTE. The RTE generator makes the following assumptions.

- Size of a pointer: 2 bytes. The default value of the RTE generator can be changed with the parameter `Size Of RAM Pointer` in the EcuC module.
- Size of the OS dependent data type `TaskType`: 1 byte
- Size of the OS dependent data type `EventMaskType`: 1 byte
- Padding bytes in structures and arrays are considered according to the configured parameters `Struct Alignment` and `Struct In Array Alignment` in the EcuC module for NVM blocks.
- Size of a `boolean` data type: 1 byte (defined in `PlatformTypes.h`)

The pointer size and the alignment parameters can be found in the container EcuC/EcucGeneral in the Basic Editor of DaVinci Configurator.

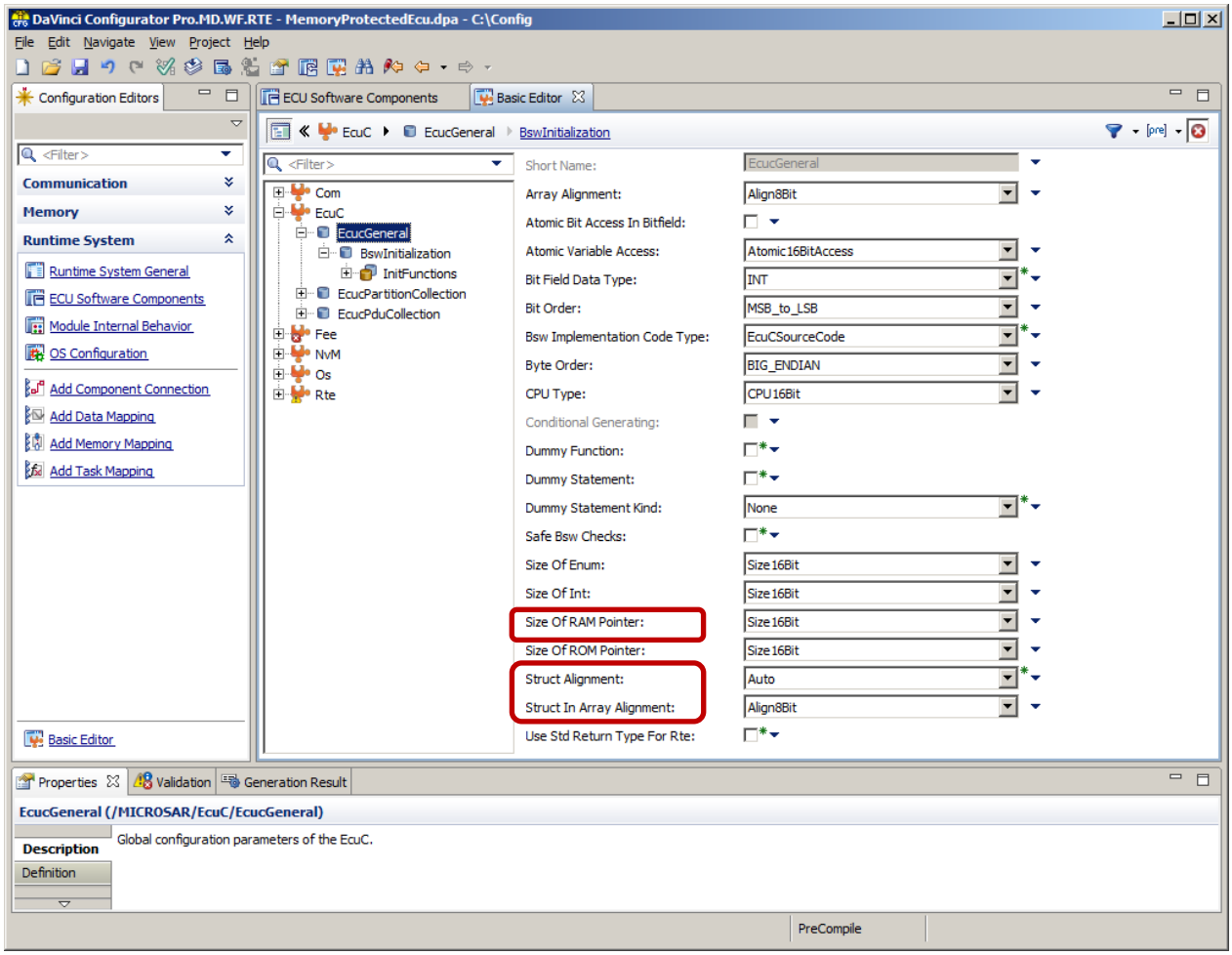


Figure 5-18 Configuration of platform settings

5.13 SWC MemMap Headerfile Generation

The generation of SWC specific MemMap header files can be enabled or disabled in the container Rte/RteSwComponentTypes in the Basic Editor of DaVinci Configurator separately for each component type. By default, the generation is enabled unless the SWC has a resource consumption and the memmap generator is available.

5.14 Implicit Communication

With the help of implicit communication, the runtime of runnable entities can be kept constant. The runnables operate on task local buffers that do not require any locking mechanism in the implicit APIs. All accesses to the global buffers are done before and after the runnable execution.

By default, the RTE generator tries to summarize multiple read accesses to the same data, multiple write accesses to the same data and write accesses followed by read accesses to reduce RAM consumption of the local buffers and copy times.

Moreover, accesses from non-preemptive tasks can directly access the global buffers without any additional buffering in the RTE.

The remaining task local buffers are stored in a union to share memory between different runnables unless the parameter RteInitializeImplicitBuffers is configured that requires persisting of the last written value.

Implicit communication can also be used to access the RAM mirror of NvBlock software components. In most cases, the RAM mirror is updated with NV data during startup before the first SWC accesses it. Moreover, often the data is stored to NV RAM only in special coding sessions. In this case the implicit buffers and copy operations in the SWC would cause unneeded overhead. Therefore, an RteExclusiveAccessOptimization container with a reference to the access and NvM_MainFunction can be configured that tells the RTE generator that the specified implicit access cannot be interrupted by any other access. When such a container is configured, the RTE accesses the RAM mirror without additional buffering. Beware that this optimization does not work with multiple storage strategies and NV block fan-out.

Caution
RteExclusiveAccessOptimization disables the RTE internal data consistency mechanisms for a specific access. It must only be configured for accesses where no concurrent accesses to the data can occur.

If a set of runnables with the same trigger conditions shall access data of the same age, so called coherency groups can be configured.

The RTE generator currently supports coherency groups with RteCoherentAccess and RteImmediateBufferUpdate configured to true. All data elements of the group will be read immediately before the first runnable of the group. The data elements will be written immediately after the last runnable of the group. In between all accesses are done to task local buffers. Please refer to the AUTOSAR specification for more details.

5.15 Rte Implementation Plugins

There need to be two prerequisites to activate a Rte Implementation Plugin for a Communication Graph. An ECUC Container defining the properties for a RIP and an ECU Flat Map containing Flat Instance Descriptors pointing to a Communication Graph and the ECUC Container.

5.15.1 RIP ECUC Container

For a plugin there needs to be an instantiated ECUC container with RteRipsPluginProps defining the identity and properties of a RIP, see the following figure:

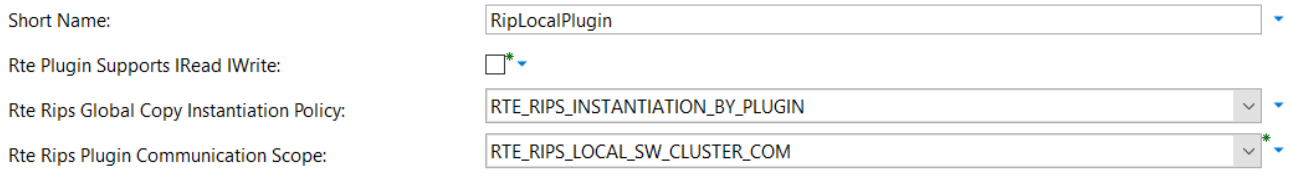


Figure 5-19 Example RteRipsPluginProps ECUC container for a Local Cluster RIP

For the FillFlushRoutines there needs to be an instantiated ECUC container with RteRipsPluginFillFlushRoutineFncs, see the following figure:

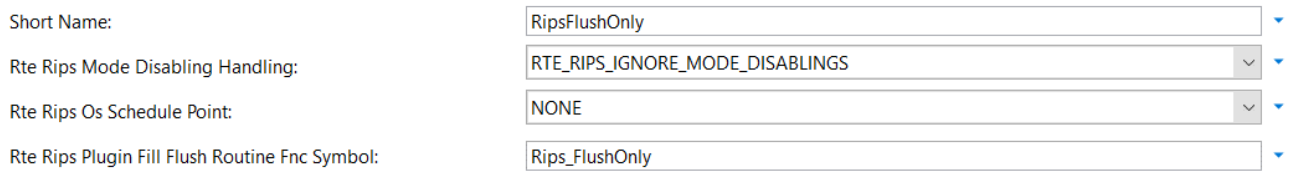


Figure 5-20 Example RteRipsPluginFillFlushRoutineFncs ECUC container for a RipsFillFlushRoutine

Note
The RIP ECUC container definition is not part of the RTE BSWMD file, it must be provided as part of a separate module.

5.15.2 RIP Flat Instance Descriptor

A RIP is activated for a communication graph if there is a Flat Instance Descriptor pointing to one specific communication graph.

The DaVinci Configurator and the RTE generator expects the ECU FlatMaps containing the FlatInstanceDescriptors to define a RIP for a specific Communication Graph to be in the ApplicationComponents directory in the DaVinci project.

```
FLAT-INSTANCE-DESCRIPTOR Graph_ctSender_TxFull_ReadWritePrimitiveCrossAndLocalByPlugin
  RTE-PLUGIN-PROPS
    ASSOCIATED-CROSS-SW-CLUSTER-COM-RTE-PLUGIN-REF /ActiveEcuC/Rip/RipCross
    ASSOCIATED-RTE-PLUGIN-REF ..... /ActiveEcuC/Rip/RipLocalPlugin
  ECU-EXTRACT-REFERENCE-IREF
    CONTEXT-ELEMENT-REF /COM/VECTOR/CFG/WORKFLOW/SYSDESC/SYNC/SYSTEM/COMPOSITIONTYPE
    CONTEXT-ELEMENT-REF /COM/VECTOR/CFG/WORKFLOW/SYSDESC/SYNC/COMPOSITIONTYPE/ctSender
    CONTEXT-ELEMENT-REF /ComponentType/ctSender/TxFull
    TARGET-REF ..... /PortInterface/piFullInternal/ReadWritePrimitiveCrossAndLocalByPlugin
```

Figure 5-21 Flat Instance Descriptor pointing to a Communication Graph

Note
For the definition of a FlatInstanceDescriptor refer to the chapter “Assigning communication graphs to RTE Implementation Plug-Ins” in [12]. Uri references are currently not supported.

5.16 Data Conversion

Data conversion can be configured for Inter- and Intra-ECU S/R communication.
To configure data conversion for Intra ECU communication a port interface mapping for sender and receiver data element is needed. This allows to connect float and integer data elements. For Intra-ECU conversion the RTE Generator supports compu methods of category IDENTICAL, LINEAR and (BITFIELD-)TEXTTABLE.
Inter-ECU data conversion can be applied to float or integer data elements that are mapped to integer signals with data type policy LEGACY or OVERWRITE. The compu method can be configured at the data element and on the signal side. For Inter-ECU conversion the RTE Generator supports compu methods of category IDENTICAL, LINEAR, TEXTTABLE and SCALE_LINEAR_AND_TEXTTABLE.
Integer to integer conversion can be disabled with the parameter /MICROSAR/Rte/RteGeneration/RteDisableInterEcuIntToIntDataConversion.

Caution
The RTE does not handle NaN values or check input values. The user must assure that only valid values are passed to the API and that the data conversion with the configured factors and offsets cannot lead to invalid output data. If invalid values cause exceptions, the handling needs to be implemented in the application.

Note
This feature needs autosar schema version r4.0 AUTOSAR_00049 or higher.

5.17 Illegal Invocation Detection

Illegal invocation detection can be configured for any combination of APIs, that are supported by this check (see 2.10).

To activate this check, at least one supported API must configure a VFB Trace Hook with the prefix “CallContext”. The code needed to detect wrong API calls will then be generated. Every API, that has a hook of the form `Rte_CallContext_<API>Hook_<cts>_<ap>_Start` configured, will be included for the check. Every other API remains unchecked, until a corresponding hook is configured.

Any other VFB Trace Hooks, that are not referring to an RTE API, are excluded from this check. Therefore, configuring `Rte_Runnable_`-, `Rte_Task_`- or `SchM_`-Hooks with the prefix “CallContext” will have no effect.

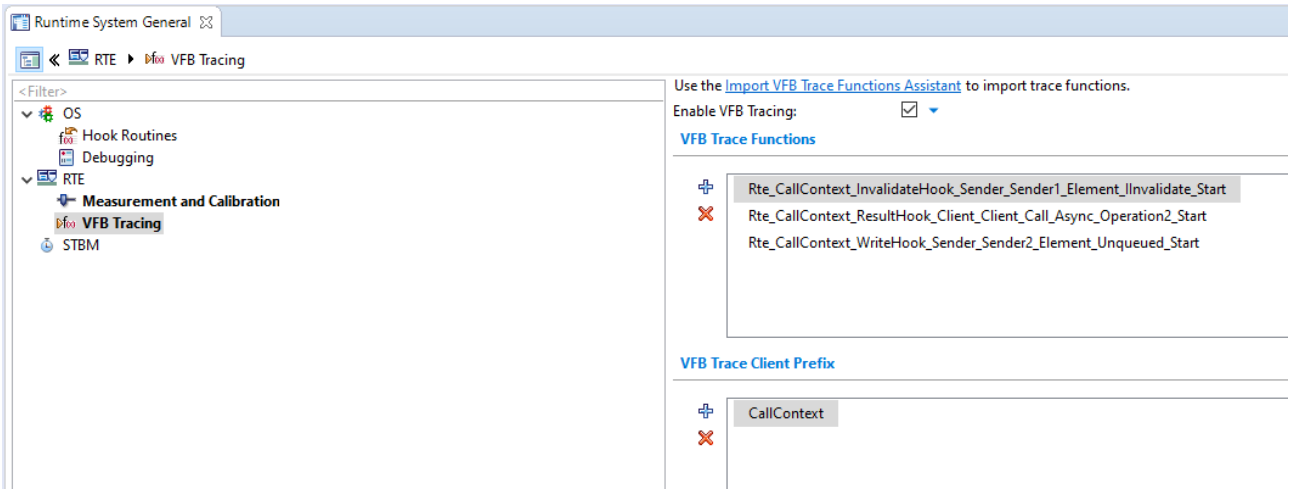


Figure 5-19 Configuration of Illegal Invocation Detection

6 Glossary and Abbreviations

6.1 Glossary

Term	Description
DaVinci DEV	DaVinci Developer: The SWC Configuration Editor.
DaVinci CFG	DaVinci Configurator: The BSW and RTE Configuration Editor.

Table 6-1 Glossary

The AUTOSAR Glossary [14] also describes a lot of important terms, which are used in this document.

6.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
BSWM	Basis Software Module
Com	Communication Layer
ComXf	Com based Transformer
CT	Component Type
C/S	Client-Server
DCM	Diagnostic Communication Manager
DiagXf	Diagnostic Transformer
E2E	End-to-End Communication Protection
E2EXf	End-to-End Transformer
EA	Exclusive Area
ECU	Electronic Control Unit
EcuM	ECU State Manager
FOSS	Free and Open Source Software
HIS	Hersteller Initiative Software
IOC	Inter OS-Application Communicator
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (Vector’s AUTOSAR solution)
NVM	Non-volatile Memory Manager
PIM	Per-Instance Memory
OIL	OSEK Implementation Language
OSEK	Open Systems and their corresponding Interfaces for Electronics in Automotive
RE	Runnable Entity

RIP	Rte Implementation Plugin
SE	Schedulable Entity
RTE	Runtime Environment
SchM	Schedule Manager
SOME/IP	Scalable service-oriented middleware over IP
SomeIpXf	SOME/IP Transformer
S/R	Sender-Receiver
SWC	Software Component
SWS	Software Specification
VFB	Virtual Functional Bus

Table 6-2 Abbreviations

7 Additional Copyrights

The MICROSAR Classic RTE Generator contains *Free and Open Source Software* (FOSS). The following table lists the files which contain this software, the kind and version of the FOSS, the license under which this FOSS is distributed and a reference to a license file which contains the original text of the license terms and conditions. The referenced license files can be found in the directory of the RTE Generator.

File	FOSS	License	License Reference
MicrosarRteGen64.exe MicrosarRteGen64 perl530.dll Folder auto	Perl 5.30	Artistic License	License_Artistic.txt
Newtonsoft.Json.dll	Json.NET 6.0.4	MIT License	License_JamesNewton-King.txt
Rte.jar	flexjson 2.1	Apache License V2.0	License_Apache-2.0.txt

Table 7-1 Free and Open Source Software Licenses

8 Contact

Visit our website for more information on

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

www.vector.com