VECTOR

MICROSAR RTE

Technical Reference

Version 4.16.0

Author	PES1.3
Status	Released

Document Information

History

Author	Date	Version	Remarks
Bernd Sigle	2005-11-14	2.0.0	Document completely reworked and adapted to AUTOSAR RTE
Bernd Sigle	2006-04-20	2.0.1	API description for Rte_IRead / Rte_IWrite added, description of used OS/COM services added
Bernd Sigle	2006-07-11	2.0.2	API description for Rte_Receive / Rte_Send added; Adaptation to RTE SWS 1.0.0 Final
Martin Schlodder	2006-11-02	2.0.3	Separation of RTE and target package
Martin Schlodder	2006-11-15	2.0.4	Client/Server communication
Martin Schlodder	2006-12-21	2.0.5	Serialized client/server communication
Martin Schlodder	2007-01-17	2.0.6	Array data types
Martin Schlodder	2007-02-14	2.0.7	Added exclusive areas, removed description of TargetPackages
Bernd Sigle	2007-02-19	2.0.8	Added transmission acknowledgement handling and minor rework of the document
Bernd Sigle	2007-04-25	2.0.9	Added Rte_IStatus
Martin Schlodder	2007-04-27	2.0.10	Added IRV and Const/Enum
Martin Schlodder Bernd Sigle	2007-05-01	2.1.0	Completed documentation for Version 2.2
Bernd Sigle	2007-07-27	2.1.1	Added Rte_InitMemory, Rte_IWriteRef Runnable. Added description of runnable activation offset und updated picture of MICROSAR architecture.
Martin Schlodder	2007-08-03	2.1.2	Added description of template update.
Martin Schlodder Bernd Sigle	2007-11-16	2.1.3	Added warning regarding IWrite / IrvIWrite. Added API descriptions of VFB trace hooks. Updated data type info for nested types.
Martin Schlodder Bernd Sigle	2008-02-06	2.1.4	Updated descriptions on template merging and task mapping. Added description of Rte_Pim, Rte_CData, Rte_Calprm and Rte_Result. Added support of string data type. Updated command line argument description. Added NvRAM mapping description. Added chapter about compiler abstraction and memory mapping.
Hannes Futter	2008-03-11	2.1.5	Additional command line switches to support direct generation on xml and dcf files.
Bernd Sigle	2008-03-26	2.2.0	Updated description of NV Memory Mapping and Chapter about limitations added. Chapter about compiler and memory abstraction updated. Support for AUTOSAR Release 3.0 added.

Bernd Sigle	2008-04-16	2.3.0	Added description about A2L file generation and updated command line options and example calls to cover also the AUTOSAR XML input files.
Bernd Sigle	2008-07-16	2.4.0	Removed limitations for multiple instantiation and compatibility mode support.
Bernd Sigle	2008-08-13	2.5.0	Added description of indirect APIs Rte_Port, Rte_Ports and Rte_NPorts. Added description of platform dependent resource calculation.
Bernd Sigle	2008-10-23	2.6.0	Added description of memory protection support.
Bernd Sigle	2009-01-23	2.7.0	Added description of mode management APIs Rte_Mode and Rte_Switch and updated description of Rte_Feedback. Added description of Rte_Invalidate and Rte_IInvalidate and added new Com APIs. Added additional runnable trigger events and removed section for runnables without trigger, which is no longer supported. Deviation for [rte_sws_2648] added. Usage of new document template
Bernd Sigle	2009-03-26	2.8.0	Removed limitations for unconnected ports and for data type generation.
Sascha Sommer Bernd Sigle	2009-08-11	2.9.0	Added description about usage of basic / extended task Added description of command line parameter -v
Sascha Sommer Bernd Sigle	2009-10-22	2.10.0	Added a warning for VFB trace hooks that prevent macro optimizations Explained that the Activation task attribute has to be set for basic tasks Init-Runnables no longer need to have a special suffix Explained the new periodic trigger implementation dialog. Server runnables with CanBeInvokedConcurrently set to false do not need to be mapped to tasks when the calling clients cannot interrupt each other Resource Usage is now listed in a HTML report Updated version of referenced documents and of supported AUTOSAR release. Updated examples with new workspace file extension. Added new defines for memory mapping.
Bernd Sigle	2010-04-09	2.11.0	Added description of user header file Rte_UserTypes.h Updated component history and interface functions to the OS. Added pictures of Rte Interfaces and Rte Include Structure. Updated picture of MICROSAR architecture. Rework of chapter structure.
Bernd Sigle	2010-05-25	2.11.1	Added description of RTE optimization mode

Bernd Sigle Sascha Sommer	2010-05-26	2.12.0	Added new measurement chapter, added description of COM Rx Filter, macros for access of invalid value, initial value, lower and upper limit, added support of minimum start interval and second array passing variant. Support of AUTOSAR Release 3.1 (RTE SWS 2.2.0)
Bernd Sigle	2010-07-22	2.13.0	Added online calibration support. Removed limitation of missing transmission error detection
Bernd Sigle	2010-09-28	2.13.1	Added more detailed description of extended record data type compatibility rule
Bernd Sigle	2010-11-23	2.14.0	Removed obsolete command line parameters –bo, –bc and –bn.
Stephanie Schaaf Bernd Sigle Sascha Sommer	2011-07-25	2.15.0	Added general support of AUTOSAR Release 3.2.1 (RTE SWS 2.4.0). Added support of never received status. Added support of S/R update handling. Mentioned that –g c and –g i ignore service components when –m specifies an ECU project. Explained RTE usage with Non-Trusted BSW Added hint for FUNC_P2CONST() problems Explained measurement of COM signals
Stephanie Schaaf Bernd Sigle Sascha Sommer	2012-01-25	2.16.0	Enhanced command line interface (support for several generation modes in one command line call, optional command line parameter –m) Split of RTE into OS Application specific files Byte arrays no longer need to be mapped to signals groups Allow configuration of Schedule() calls in non-preemptive tasks
Bernd Sigle	2012-05-18	2.17.0	Corrected description how the Rte_IsUpdated API can be enabled
Bernd Sigle	2012-09-18	2.18.0	Added general support of AUTOSAR Release 3.2.2 (RTE SWS 2.5.0). Added support of non-queued N:1 S/R communication
Bernd Sigle	2012-08-28	3.90.0	AUTOSAR 4.0.3 support, DaVinci Configurator 5 support
Bernd Sigle	2012-12-11	4.0.0	Updated API descriptions concerning RTE_E_UNCONNECTED return code Added description of Rte_UserTypes.h file which is now also generated with the template mechanism
Stephanie Schaaf	2013-03-26	4.1.0	Added support of Rte_MemSeg.a2l file Added description of –o sub option for A2L file path
Bernd Sigle	2013-06-14	4.1.1	Added Multi-Core support (S/R communication) Added support of Inter-Runnable Variables with composite data types

Katharina Benkert Stephanie Schaaf Sascha Sommer Bernd Sigle	2013-10-30	4.2.0	Added support for arrays of dynamic data length (Rte_Send/Rte_Receive) Added support for parallel generation for multiple component types Multicore support Added support for SchM Contract Phase Generation Added support for Nv Block SWCs
Katharina Benkert Sascha Sommer Stephanie Schaaf	2014-02-06	4.3.0	Added support of VFB Trace Client Prefixes Optimized Multicore support without IOCs Memory Protection support for Multicore systems Inter-ECU sender/receiver communication, queued sender/receiver communication and mapped client/server calls are no longer limited to the BSW partition Added support of Development Error Reporting Added support of registering XCP Events in the XCP module configuration
Stephanie Schaaf Bernd Sigle	2014-06-17	4.4.0	Support for unconnected client ports for synchronous C/S communication Inter-Ecu C/S communication using SOME/IP Transformer Support for PR-Ports S/R Serialization using SOME/IP Transformer and E2E Transformer Support LdCom
Bernd Sigle	2014-08-13	4.4.1	Described decimal coding of the version defines and the return code of SchM_GetVersionInfo Added chapter about additional copyrights of FOSS
Bernd Sigle	2014-09-12	4.4.2	Minor format changes only
Bernd Sigle	2014-08-13	4.5.0	Support Postbuild-Selectable for variant data mappings and variant COM signals Support E2E Transformer for Inter-Ecu C/S communication Support tasks mappings where multiple runnable or schedulable entities using different cycle times or activation offsets are mapped to a single Basic Task. The realization uses OS Schedule Tables Support Rte_DRead API Enhanced support for PR-Ports Support ServerArgumentImplPolicy = use ArrayBaseType Explicit order of ModeDeclarationGroups

Bernd Sigle	2014-12-08	4.6.0	Support of PR Mode Ports Support of PR Nv Ports Support of bit field data types (CompuMethods with category BITFIELD_TEXTTABLE) Runtime optimized copying of large data Support for SW-ADDR-METHOD on RAM blocks of NvRAM SWCs
Bernd Sigle	2015-02-20	4.7.0	Support of background triggers Support of data prototype mappings Support of bit field text table mappings Support of union data types Support of UTF16 data type serialization in the SOME/IP transformer Runtime optimization in the generated RTE code by usage of optimized interrupt locking APIs of the MICROSAR OS Support of further E2E profiles for data transformation with the SOME/IP and E2E transformer Support of OS counters with tick durations smaller than 1µs
Bernd Sigle	2015-07-26	4.8.0	Support of COM based Transformer ComXf Support of different strategies for writing NV data in Nv Block SWCs Support of C/S Interfaces for Nv Block SWCs SWC Template generation provides user sections for documentation of runnable entities Wide character support in paths Improved counter selection for operating systems with multiple OS applications Support of optimized macro implementation for SchM_Enter and SchM_Exit Enhanced OS Spinlock support Enable optimizations in QM partitions
Bernd Sigle	2016-01-04	4.9.0	Support of BSW multiple partition distribution Support of activation reason for runnable entities (Rte_ActivatingEvent) Support for initialization of send buffers for implicit S/R communication Generation of VFB Trace Hook calls only if hooks are configured Support of 64 events per task if supported by the MICROSAR OS Support of subelement mapping for Rx-GroupSignals Support for RteUseComShadowSignalApi Updated CFG5 figures

Bernd Sigle	2016-02-23	4.10.0	AUTOSAR 4.2.2 support Enhanced SomeIpXf support Support of literal prefix Migration to new Vector CI
Bernd Sigle Sascha Sommer	2016-05-13	4.11.0	Support of application data types of category map, curve and axis Selection of COM signal timeout source (Swc / Signal) Support of 1:n Inter-ECU S/R with transmission acknowledgement Support E2EXf for primitive byte arrays without serializer Autonomous error responses for Inter-ECU C/S with SomeIpXf Described mapping of SWCs to OS Applications.
Bernd Sigle	2016-07-14	4.12.0	Support of connections between Nv ports and S/R ports Support of Diagnostic Data Transformation (DiagXf) Support of Data Conversion between integer data types on network signals and floating point data types on SWC ports Support of counters from different partitions that are assigned to the same core Updated RTE and SWC include structure
Sascha Sommer	2016-11-21	4.13.0	Described CompuScale limitation Extended multicore documentation
Bernd Sigle Sascha Sommer	2017-03-28	4.14.0	Support of Transformer Error Handling Updated DET error codes and Service IDs Minor improvements.
Patrick Alschbach Katharina Benkert	2017-06-08	4.15.0	Data conversion for signals of signal groups Minor improvements.
Sascha Sommer	2017-08-15	4.16.0	Metadata support for Inter-ECU client-server communication Support for maps and curves that are mapped to array implementation datatypes Display format on data types is now used for A2L generation

Table 1-1 History of the document

Reference Documents

No.	Title	Version
[1]	AUTOSAR_SWS_RTE.pdf	4.2.2
[2]	AUTOSAR_EXP_VFB.pdf	4.2.2
[3]	AUTOSAR_SWS_COM.pdf	4.2.2
[4]	AUTOSAR_SWS_OS.pdf	4.2.2
[5]	AUTOSAR_SWS_NVRAMManager.pdf	4.2.2
[6]	AUTOSAR_SWS_ECU_StateManager.pdf	4.2.2
[7]	AUTOSAR_SWS_StandardTypes.pdf	4.2.2
[8]	AUTOSAR_SWS_PlatformTypes.pdf	4.2.2
[9]	AUTOSAR_SWS_CompilerAbstraction.pdf	4.2.2
[10]	AUTOSAR_SWS_MemoryMapping.pdf	4.2.2
[11]	AUTOSAR_TPS_SoftwareComponentTemplate.pdf	4.2.2
[12]	AUTOSAR_TPS_SystemTemplate.pdf	4.2.2
[13]	AUTOSAR_TPS_ECUConfiguration.pdf	4.2.2
[14]	AUTOSAR_TR_Glossary.pdf	4.2.2
[15]	AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf	4.2.2
[16]	AUTOSAR_SWS_XCP.pdf	4.2.2
[17]	AUTOSAR_SWS_ DefaultErrorTracer.pdf	4.2.2
[18]	AUTOSAR_SWS_LargeDataCOM.pdf	4.2.2
[19]	AUTOSAR_SWS_SOMEIPTransformer.pdf	4.2.2
[20]	AUTOSAR_SWS_COMBasedTransformer.pdf	4.2.2
[21]	AUTOSAR_SWS_E2ETransformer.pdf	4.2.2
[22]	Vector DaVinci Configurator Online help	
[23]	Vector DaVinci Developer Online help	
[24]	AUTOSAR Calibration User Guide	1.0

Table 1-2 Reference documents

Scope of the Document

This document describes the MICROSAR 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 1-2.

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 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
2.3	▶ Complex hierarchical data types like arrays of records ▶ Optimization: Depending on the configuration the Rte_Read API is generated as macro if possible
2.4	▶ String data type (Encoding ISO-8859-1) ▶ SWC local calibration parameters (Rte_CData) ▶ Optimization: Depending on the configuration the Rte_Write API is generated as macro if possible ▶ Generation of unmapped client runnables enabled ▶ Asynchronous C/S communication (Rte_Result)
2.5	▶ Support of AUTOSAR 3.0 Revision 0001 ▶ Access to calibration element prototypes of calibration components (Rte_Calprm) ▶ Access to Per-Instance Memory (Rte_Pim) ▶ SWC implementation template generation (command line option `-g i`) and Contract Phase generation (command line option `-g c`) for a complete ECU
2.6	▶ Intra-ECU timeout handling for synchronous C/S communication ▶ Parallel access of synchronous and asynchronous server calls to an operation of one server port ▶ Generation of an ASAM MCD 2MC / ASAP2 compatible A2L file fragment for calibration parameters and Per-Instance Memory
2.7	▶ Multiple instantiation of software components ▶ Compatibility mode ▶ Object code software components
2.8	▶ Indirect APIs (Rte_Ports, Rte_NPorts and Rte_Port) ▶ Port API Option 'EnableTakeAddress' ▶ Platform dependent resource calculation.
2.9	▶ Memory protection (OS with scalability class SC3/SC4)
2.10	▶ Mode management including mode switch triggered runnable entities and mode dependent execution of runnable entities. (Rte_Switch, Rte_Mode and Rte_Feedback for mode switch acknowledge) ▶ Data element invalidation (Rte_Invalidate and Rte_IInvalidate) ▶ Data reception error triggered runnable entities for invalidated and outdated data elements ▶ Multiple cyclic triggers per runnable entity ▶ Multiple OperationInvokedEvent triggers for the same runnable entity with compatible operations ▶ Extended A2L file generation for calibration parameters and Per-

Component Version	New Features
	Instance Memory for user defined attributes (A2L-ANNOTATION)
2.11	▶ Signal Fan-In ▶ Unconnected provide ports ▶ Generation of unreferenced data types ▶ Evaluation of COM return codes
2.12	▶ Basic task support (automatically selection) ▶ Several optimizations (e.g. unneeded interrupt locks and Schedule() call removed) ▶ Enhanced error reporting with help messages (-v command line option) ▶ Support of acknowledgement only mode for transmission and mode switch notification ▶ Usage of compiler library functions (e.g. memcpy) removed ▶ Template file update mechanism also introduced for Rte_MemMap.h and Rte_Compiler_Cfg.h
2.13	▶ Unconnected require ports ▶ Basic task support (manual selection) ▶ Init-Runnables no longer have name restrictions ▶ Automatic periodic trigger generation can be disabled e.g. useful for Schedule Table support ▶ HTML Report including resource usage ▶ Explicit selection of task role (Application / BSW Scheduler / Non Rte) ▶ Runnables with CanBeInvokedConcurrently set to false no longer require a mapping, if they are not called concurrently.
2.14	▶ Support composite data types where not all primitive members require an invalid value ▶ Support inclusion of user header file 'Rte_UserTypes.h'
2.15	▶ Optimized runnable scheduling to reduce latency times ▶ Allow implementation template generation for service components, complex device drivers and EcuAbstraction components ▶ Optimization mode (minimize RAM consumption / minimize execution time)
2.16	▶ MinimumStartInterval attribute (runnable de-bouncing) ▶ Measurement support for S/R communication, Interrunnable variables and mode communication. Extended A2L File generation and support of new ASAM MCD 2MC / ASAP2 standard. Measurement with XcpEvents ▶ Com Filter (NewDiffersOld, Always) ▶ Invalid value accessible from application ▶ Support of second array passing variant
2.17	▶ Online calibration support ▶ Support transmission error detection ▶ Support of extended record data type compatibility for S/R communication with different record layout on sender and receiver side
2.18	▶ Enhanced implicit communication support

Component Version	New Features
2.19	▶ Support of AUTOSAR 3.2 Revision 0001 ▶ Support never received status ▶ Support S/R update handling (Rte_IsUpdated based on AUTOSAR 4.0) ▶ Enhanced measurement support (Inter-Ecu S/R communication) ▶ Selective file generation (only if file content is modified) ▶ Support for Non-Trusted BSW
2.20	▶ Enhanced command line interface (support for several generation modes in one call, optional command line parameter `-m`) ▶ Split of generated RTE into OS Application specific files ▶ Byte arrays no longer need to be mapped to signal groups ▶ Allow configuration of Schedule() calls in non-preemptive tasks ▶ Generation of MISRA justification comments
2.21	▶ Support of SystemSignals and SystemSignalGroups using the same name ▶ Support of hexadecimal coded enumeration values
2.22	▶ Support of AUTOSAR 3.2 Revision 0002 ▶ Support S/R update handling according AUTOSAR 3.2.2 ▶ Support N:1 S/R communication ▶ Support unconnected calibration R-Ports ▶ Enhanced initial value handling
3.90	▶ Support of AUTOSAR 4.0 Revision 0003
4.0	▶ Support of pointer implementation data types ▶ Support of 'On Transition' triggered runnable entities ▶ Support of data type symbol attribute ▶ Support of component type symbol attribute ▶ Template generation mechanism added for Rte_UserTypes.h
4.1	▶ Support of Rte_MemSeg.a2l ▶ Enhanced command line interface (path for A2L files selectable)
4.1.1	▶ Multi-Core support (S/R communication) ▶ Support of Inter-Runnable Variables with composite data types

4.2	▶ Support for arrays of dynamic data length (Rte_Send/Rte_Receive)
	▶ Support for parallel generation for multiple component types
	▶ Multi-Core support:
	  ▶ C/S communication
	  ▶ Mode communication without ModeDisablings and ModeTriggers
	  ▶ Inter-ECU S/R communication
	▶ Support mapping of individual OperationInvoked triggers
	▶ Support of SchM Contract Phase Generation
	▶ Support of Nv Block SWCs

4.3	▶ Support of VFB Trace Client Prefixes
	▶ Enhanced Memory Protection support
	  ▶ Memory Protection support for Multi-Core systems
	  ▶ Inter-ECU sender/receiver communication is no longer limited to the

Component Version	New Features
	BSW partition ▶ Mapped client/server calls are no longer limited to the BSW partition ▶ Queued sender/receiver communication is no longer limited to the BSW partition ▶ Optimized Multi-Core support without IOCs ▶ Support of Development Error Reporting ▶ Support of registering XCP Events in the XCP module configuration
4.4	▶ Support for unconnected client ports for synchronous C/S communication ▶ Inter-Ecu C/S communication using SOME/IP Transformer ▶ Support for PR-Ports ▶ S/R Serialization using SOME/IP Transformer and E2E Transformer ▶ Support LdCom ▶ Improved support for 3rd Party OS interoperability especially regarding OS Counter handling
4.5	▶ Support Postbuild-Selectable for variant data mappings and variant COM signals ▶ Support E2E Transformer for Inter-Ecu C/S communication ▶ Support tasks mappings where multiple runnable or schedulable entities using different cycle times or activation offsets are mapped to a single Basic Task. The realization uses OS Schedule Tables ▶ Support Rte_DRead API ▶ Enhanced support for PR-Ports ▶ Support ServerArgumentImplPolicy = use ArrayBaseType ▶ Support for Mode Declaration Groups with Explicit Order
4.6	▶ Support of PR Mode Ports ▶ Support of PR Nv Ports ▶ Support of bit field data types (CompuMethods with category BITFIELD_TEXTTABLE) ▶ Runtime optimized copying of large data ▶ Support for SW-ADDR-METHOD on RAM blocks of NvRAM SWCs
4.7	▶ Support of background triggers ▶ Support of data prototype mappings ▶ Support of bit field text table mappings ▶ Support of union data types ▶ Support of UTF16 data type serialization in the SOME/IP transformer ▶ Runtime optimization in the generated RTE code by usage of optimized interrupt locking APIs of the MICROSAR OS ▶ Support of further E2E profiles for data transformation with the SOME/IP and E2E transformer ▶ Support of OS counters with tick durations smaller than 1µs
4.8	▶ Support of COM based Transformer ComXf ▶ Support of different strategies for writing NV data in Nv Block SWCs ▶ Support of C/S Interfaces for Nv Block SWCs ▶ SWC Template generation provides user sections for documentation of

Component Version	New Features
	runnable entities ▶ Wide character support in paths ▶ Improved counter selection for operating systems with multiple OS applications ▶ Support of optimized macro implementation for SchM_Enter and SchM_Exit ▶ Enhanced OS Spinlock support ▶ Enable optimizations in QM partitions
4.9	▶ Support of BSW multiple partition distribution ▶ Support of activation reason for runnable entities (Rte_ActivatingEvent) ▶ Support for initialization of send buffers for implicit S/R communication ▶ Generation of VFB Trace Hook calls only if hooks are configured ▶ Support of 64 events per task if supported by the MICROSAR OS ▶ Support of subelement mapping for Rx-GroupSignals ▶ Support for RteUseComShadowSignalApi
4.10	▶ AUTOSAR 4.2.2 support ▶ Enhanced SomeIpXf support ▶ Support of literal prefix ▶ Support of VFB Trace Hooks for APIs of unconnected Ports ▶ Support for NvMAutomaticBlockLength parameter ▶ Support of E2E profiles 1 and 2 for SomeIpXf and E2EXf ▶ Support of E2E profiles 4, 5 and 6 for ComXf and E2EXf
4.11	▶ Support of application data types of category map, curve and axis ▶ Selection of COM signal timeout source (Swc / Signal) ▶ Support of 1:n Inter-ECU S/R with transmission acknowledgement ▶ Support E2EXf for primitive byte arrays without serializer ▶ Autonomous error responses for Inter-ECU C/S with SomeIpXf
4.12	▶ Support of connections between Nv ports and S/R ports ▶ Support of Diagnostic Data Transformation (DiagXf) ▶ Support of Data Conversion between integer data types on network signals and floating point data types on SWC ports ▶ Support of counters from different partitions that are assigned to the same core
4.13	▶ Added support for SpinlockLockMethod RES_SCHEDULER ▶ Several improvements and bugfixes
4.14	▶ Support of Transformer Error Handling for S/R communication
4.15	▶ Support of Data conversion for signals of signal groups
4.16	▶ Support of metadata for Inter-ECU C/S communication ▶ Support for maps and curves that are mapped to array implementation datatypes ▶ Display format on data types is now used for A2L generation

Table 1-1 Component history

2 Introduction

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

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

Chapter 3 gives an introduction to the MICROSAR 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 3 describes deviations, extensions and limitations of the MICROSAR RTE compared to the AUTOSAR standard.

The RTE generation process including the command line parameters of the MICROSAR RTE generator is described in chapter 4. This chapter also gives hints for 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 4.6 describes the memory protection support of the RTE including hints for integration with the OS.

The RTE API description in chapter 5 covers the API functionality implemented in the MICROSAR RTE.

The description of the RTE configuration in chapter 6 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 AUTOSAR Release*:	4	
Supported Configuration Variants:	pre-compile	
Vendor ID:	RTE_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	RTE_MODULE_ID	2 decimal
AR Version:	RTE_AR_RELEASE_MAJOR_VERSION RTE_AR_RELEASE_MINOR_VERSION RTE_AR_RELEASE_REVISION_VERSION	AUTOSAR Release version decimal coded
SW Version:	RTE_SW_MAJOR_VERSION RTE_SW_MINOR_VERSION RTE_SW_PATCH_VERSION	MICROSAR RTE version decimal coded

* For the precise AUTOSAR Release 4.x please see the release specific documentation.

2.1 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 RTE is located in the AUTOSAR architecture.

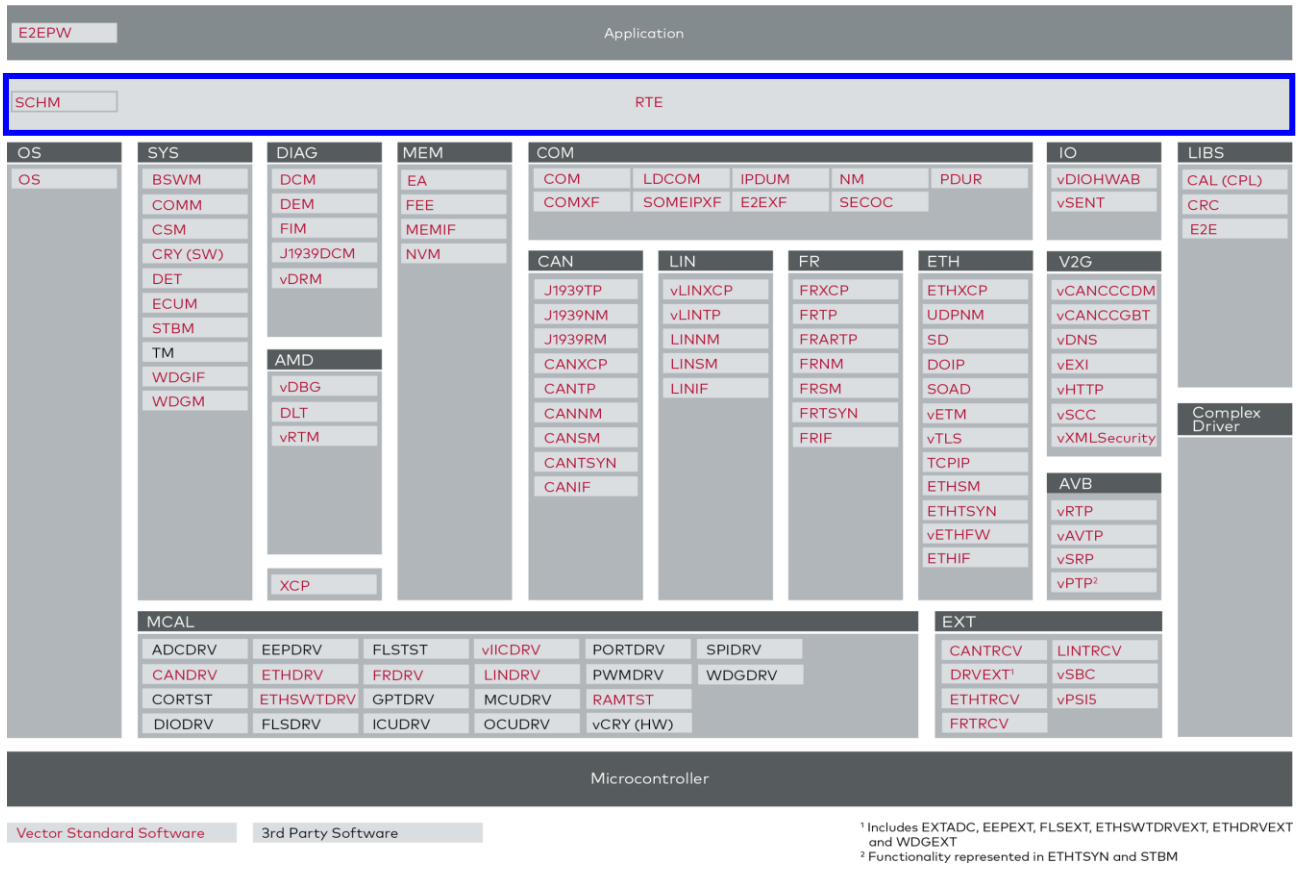


Figure 2-1 AUTOSAR architecture

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

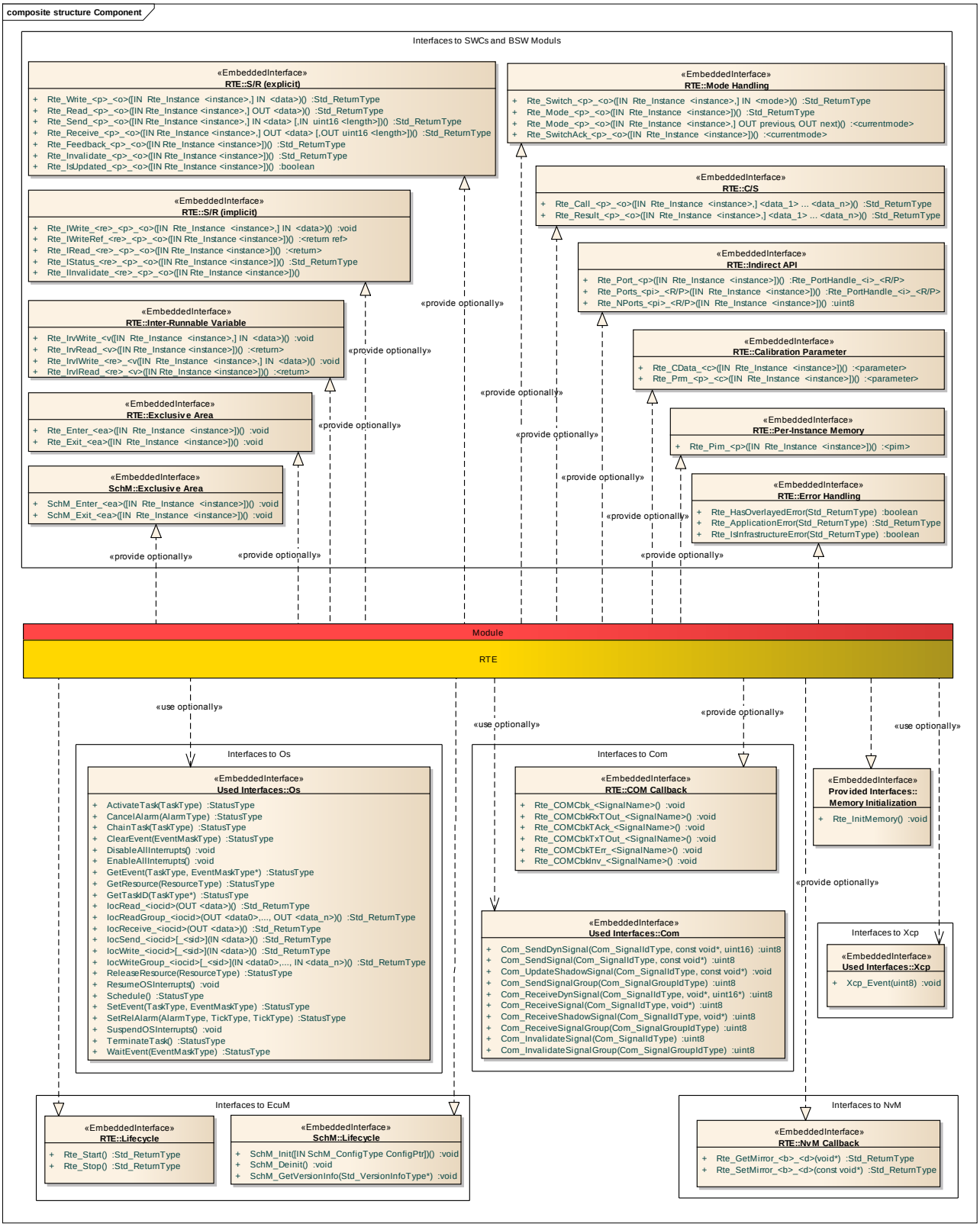


Figure 2-2 Interfaces to adjacent modules of the RTE

3 Functional Description

3.1 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 3-1 Supported AUTOSAR standard conform features
- Table 3-2 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 3-3 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, 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
Explicit exclusive areas [API: Rte_Enter, Rte_Exit]
Implicit exclusive areas
Explicit Inter-Runnable Variables [API: Rte_IrvRead, Rte_IrvWrite]
Implicit Inter-Runnable Variables [API: Rte_IIrvRead, Rte_IIrvWrite]

Supported AUTOSAR Standard Conform Features
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))
Memory protection
Unconnected ports
COM-Filter (NewDiffersOld, Always)
Measurement (S/R-Communication, Mode-Communication, Inter-Runnable Variables)
Runnable de-bouncing (Minimum Start Interval)

Supported AUTOSAR Standard Conform Features
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
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
E2E Transformer Support
Transformer Error Handling for S/R
Initialization of send buffers for implicit S/R communication
Data conversion (limited to S/R communication with integer network signal(s) mapped to floating point data types on SWC ports, compu methods of type LINEAR or IDENTICAL and data type policy LEGACY or OVERRIDE)

Table 3-1 Supported AUTOSAR standard conform features

3.1.1 Deviations

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

Not Supported AUTOSAR Standard Conform Features
COM-Filter (only partially supported)
Measurement (Client-Server arguments)
external Trigger (via port) [API: Rte_Trigger]
Inter-Runnable Trigger (SWC internal) [API: Rte_IrTrigger]
Tx-Ack for implicit communication [API: Rte_IFeedback]
BSW-Scheduler Mode Handling [API: SchM_Mode, SchM_Switch, SchM_SwitchAck]
external Trigger between BSW modules [API: SchM_Trigger]
BSW-Scheduler Trigger [API: SchM_ActMainFunction]
BSW-Scheduler Calibration Parameter Access [API: SchM_CData]
BSW-Scheduler queued S/R communication [API: SchM_Send, SchM_Receive]
BSW-Scheduler C/S communication [API: SchM_Call, SchM_Result]

Not Supported AUTOSAR Standard Conform Features
BSW-Scheduler Per-Instance Memory Access [API: SchM_Pim]
Enhanced Rte Lifecycle API [API: Rte_StartTiming]
Post Build Variant Sets
Debugging and Logging Support
Variant Handling support (Pre-Compile variability, Postbuild variability for Connectors and ComponentPrototypes)
Multi-Core support (Mode communication with ModeSwitchTriggers or ModeDisablings, Rte_ComSendSignalProxyImmediate, RteIocInteractionReturnValue=RTE_COM)
Activation reason in multicore and memory protection systems
Restarting of partitions
Re-scaling of ports / Data conversion (only partially supported)
Pre-Build data set generation phase
Post-Build data set generation phase
Initialization of PerInstanceMemories
Asynchronous Mode Handling
MC data support
Generated BSWMD
Range checks
RTE memory section initialization strategies
Configuration of coherency groups for implicit communication
Immediate Buffer update for implicit communication
External configuration switch strictConfigurationCheck
ScaleLinear and ScaleLinearTexttable CompuMethods with more than one CompuScale

Table 3-2 Not supported AUTOSAR standard conform features

3.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Rte_InitMemory API function. See Chapter 5.14.3 for details.
Init-Runnables. See Chapter 3.6.9 for details.
VFB Trace Hooks for SchM APIs. See Chapter 5.16.3 and 5.16.4 for details.
Measurement support for mode communication. See Chapter 6.6 for details.
Measurement with XCP Events. See Chapter 6.6 for details.

Table 3-3 Features provided beyond the AUTOSAR standard

3.1.3 Limitations

There are no known limitations.

3.2 Initialization

The RTE is initialized by calling `Rte_Start`. Initialization is done by the ECU State Manager (EcuM).

The Basis Software Scheduler (SchM) is initialized by calling `SchM_Init`. Initialization is done by the ECU State Manager (EcuM).

3.3 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).

3.4 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 between different ECUs (Inter-ECU). 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.

3.5 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 5.3 Runnable Entities.

3.6 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 RTE:

- TimingEvent
- DataReceivedEvent
- DataReceiveErrorEvent
- DataSendCompletedEvent
- OperationInvokedEvent
- AsynchronousServerCallReturnsEvent
- ModeSwitchEvent
- ModeSwitchedAckEvent
- InitEvent
- BackgroundEvent

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. The second is not possible for all RTEEvents and needs an RTE API to setup this waitpoint inside the runnable entity code.

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.

3.6.1 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])
```

3.6.2 Data Received Triggered Runnables

AUTOSAR defines the `DataReceivedEvent` to trigger a runnable entity on data reception (queued or last-is-best) or to continue reception of 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])
```

3.6.3 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])
```

3.6.4 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 waits at a waitpoint for the transmission status or the mode switch in a blocking `Rte_Feedback` call. 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])
```

3.6.5 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])
```

3.6.6 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 waits 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])
```

3.6.7 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 in/output and output parameters. Input parameters for primitive data type are passed by value. Input parameters for composite data types and all in/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.

3.6.8 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])
```

3.6.9 Init Triggered Runnables

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

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

3.6.10 Background Triggered Runnables

Background triggered runnables have to be mapped to tasks with lowest priority. The runnables are called by the RTE in an endless loop – in the background – when no other runnable runs. The signature of a background triggered runnable is:

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

3.7 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 6.9.

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

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

The MICROSAR RTE analyzes the accesses to the different RTE exclusive areas and eliminates redundant ones if that is possible e.g. if runnable entities accessing the same EA they cannot preempt each other and can therefore be dropped.

Info
For SchM exclusive areas the automatic optimization is currently not supported. Optimization must be done manually by setting the implementation method to `NONE`.
In addition the implementation of the Exclusive Area APIs for the SchM can be set to `CUSTOM`. In that case the RTE generator doesn't generate the `SchM_Enter` and `SchM_Exit` APIs. Instead of the APIs have to be implemented manually by the customer.

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

3.7.1 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. 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.

3.7.2 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.

3.7.3 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.

3.7.4 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.

3.8 Error Handling

3.8.1 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 5. 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
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
0x30	Rte_IsUpdated
0x70	Rte_Start
0x71	Rte_Stop
0x90	Rte_COMCbkTAck_<sn>
0x91	Rte_COMCbkTErr_<sn>
0x92	Rte_COMCbkInv_<sn>

Service ID	Service
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>
0xA6	Rte_LdComCbkTriggerTransmit_<sn>
0xA7	Rte_LdComCbkTxConfirmation_<sn>
0xF0	Rte_Task
0xF1	Rte_IncDisableFlags
0xF2	Rte_DecDisableFlags

Table 3-4 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 RTE and NvM

Table 3-5 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.

4 RTE Generation and Integration

This chapter gives necessary information about the content of the delivery, the RTE generation process including a description about the different RTE generation modes and finally information how to integrate the MICROSAR RTE into an application environment of an ECU.

4.1 Scope of Delivery

The delivery of the RTE contains no static RTE code files. The RTE module is completely generated by the MICROSAR RTE Generator. After the installation, the delivery has the following content:

Files	Description
DVCfgRteGen.exe (including several DLLs and XML files)	Command line MICROSAR RTE generator
MicrosarRteGen.exe	MICROSAR RTE File generator
Rte.jar	DaVinci Configurator 5 adaptation modules
Settings_Rte.xml	
Rte_Bswmd.arxml	BSWMD file for MICROSAR RTE
TechnicalReference_Asr_Rte.pdf	This documentation
ReleaseNotes_MICROSAR_RTE.htm	Release Notes

Table 4-1 Content of Delivery

Info
The RTE Configuration Tool DaVinci Developer is not part of MICROSAR RTE / BSW installation package. It has to be installed separately.

4.2 RTE Generation

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

4.2.1 Command Line Options

Option		Description
`--project <file>`	`-p <file>`	Specifies the absolute path to the DPA project file.
`--generate`	`-g`	Generate the given project specified in `<file>`.
`--modulesToGenerate`	`-m <module>`	Specifies the module definition references, which should be generated by the `-g` switch. Separate multiple modules by a ','. 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 4-3.
`--help`	`-h`	Displays the general help information of `DVCfgCmd.exe`

Table 4-2 DVCfgCmd Command Line Options

4.2.2 RTE Generator Command Line Options

Option	Description
`-m <obj>`	Selects the DaVinci model object, where `<obj>` is either `<ECUProjectName>` or `<ComponentTypeName>`. Note: If `-g i` or `-g c` are selected, which accepts both, `<ComponentTypeName>` or `<ECUProjectName>` and the configuration contains such objects with the same name, the component type object takes preference over the ECU project. 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 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 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.
-v	Enables verbose mode which includes help information for error, warning and info messages.
-h	Displays the general help information.

Table 4-3 RTE Generator Command Line Options

4.2.3 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 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>".

4.3 MICROSAR RTE generation modes

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

4.3.1 RTE Generation Phase

The following files are generated by the RTE generation: (Option -g or -g r)

File	Description
Rte_<ComponentType>.h	Application header file, which has 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_<ComponentType>_Type.h	Application type header file. This header file contains SWC specific type definitions. It is generated to the Components subdirectory by default.
SchM_<BswModule>.h	Module interlink header file, which has to be included into the BSW module code.
SchM_<BswModule>_Type.h	Module interlink types header file. This header file contains BSW module specific type definitions.
<ComponentType>_MemMap.h	Template contains SWC specific part of the memory mapping. It is generated to the Components subdirectory by default.
Rte.c	Generated RTE
Rte_<OsApplication>.c	OsApplication specific part of the generated RTE (only generated when OsApplications are configured)
Rte_PBCfg.c	The RTE post build data set configuration file contains the data structures for the postbuild RTE initialization.
Rte.h	RTE internal declarations

Rte_Main.h	Header file for RTE lifecycle API
Rte_Cfg.h	Configuration file for the RTE
Rte_Cbk.h	Contains prototypes for COM callbacks
Rte_Hook.h	Contains relevant information for VFB tracing
Rte_Type.h	Contains the application defined data type definitions and RTE internal data types
Rte_DataHandleType.h	The RTE data handle types header file contains the data handle type declarations required for the component data structures.
Rte_PBCfg.h	The RTE post build data set configuration header contains the declarations for the data structures that are used for the postbuild RTE initialization.
Rte_UserTypes.h	Template 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.
Rte_MemMap.h	Template contains RTE specific part of the memory mapping
Rte_Compiler_Cfg.h	Template contains RTE specific part of the compiler abstraction
usrostyp.h	Template which is only generated if memory protection support is enabled. In that case this file is included by the MICROSAR OS.
Rte.oil	OS configuration for the RTE
Rte_Needs.ecuc.arxml	Contains the RTE requirements on BSW module configuration for Os, Com, LdCom, Xcp and NvM.
Rte.a2l	A2L file containing CHARACTERISTIC and MEASUREMENT objects for the generated RTE
Rte_MemSeg.a2l	A2L file containing MEMORY_SEGMENT objects for the generated RTE
Rte_rules.mak, Rte_defs.mak, Rte_check.mak, Rte_cfg.mak	Make files according to the AUTOSAR make environment proposal are generated into the `mak` subdirectory.
Rte.html	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 4-4 Generated Files of RTE Generation Phase

Example:

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

4.3.2 RTE Contract Phase Generation

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

File	Description
Rte_<ComponentType>.h	Application header file, which must be included into the SWC code. This header file is the only file to be included in the component code.
Rte_<ComponentType>_Type.h	Application type header file. This header file contains SWC specific type definitions.
<ComponentType>_MemMap.h	Template contains SWC specific part of the memory mapping.
Rte.h	RTE internal declarations
Rte_Type.h	Contains the application defined data type definitions and RTE internal data types
Rte_DataHandleType.h	The RTE data handle types header file contains the data handle type declarations required for the component data structures.
Rte_UserTypes.h	Template 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.
Rte_MemMap.h	Template contains RTE specific part of the memory mapping
Rte_Compiler_Cfg.h	Template contains RTE specific part of the compiler abstraction
SchM_<BswModule>.h	Module interlink header file, which has to be included into the BSW module code.
SchM_<BswModule>_Type.h	Module interlink types header file. This header file contains BSW module specific type definitions.

Table 4-5 Generated Files of RTE Contract Phase

Example:

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

4.3.3 Template Code Generation for Application Software Components

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

File	Description
<FileName>.c	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.

Table 4-6 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.

4.3.4 VFB Trace Hook Template Code Generation

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

File	Description
<FileName>.c	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.

Table 4-7 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.

4.4 Include Structure

4.4.1 RTE Include Structure

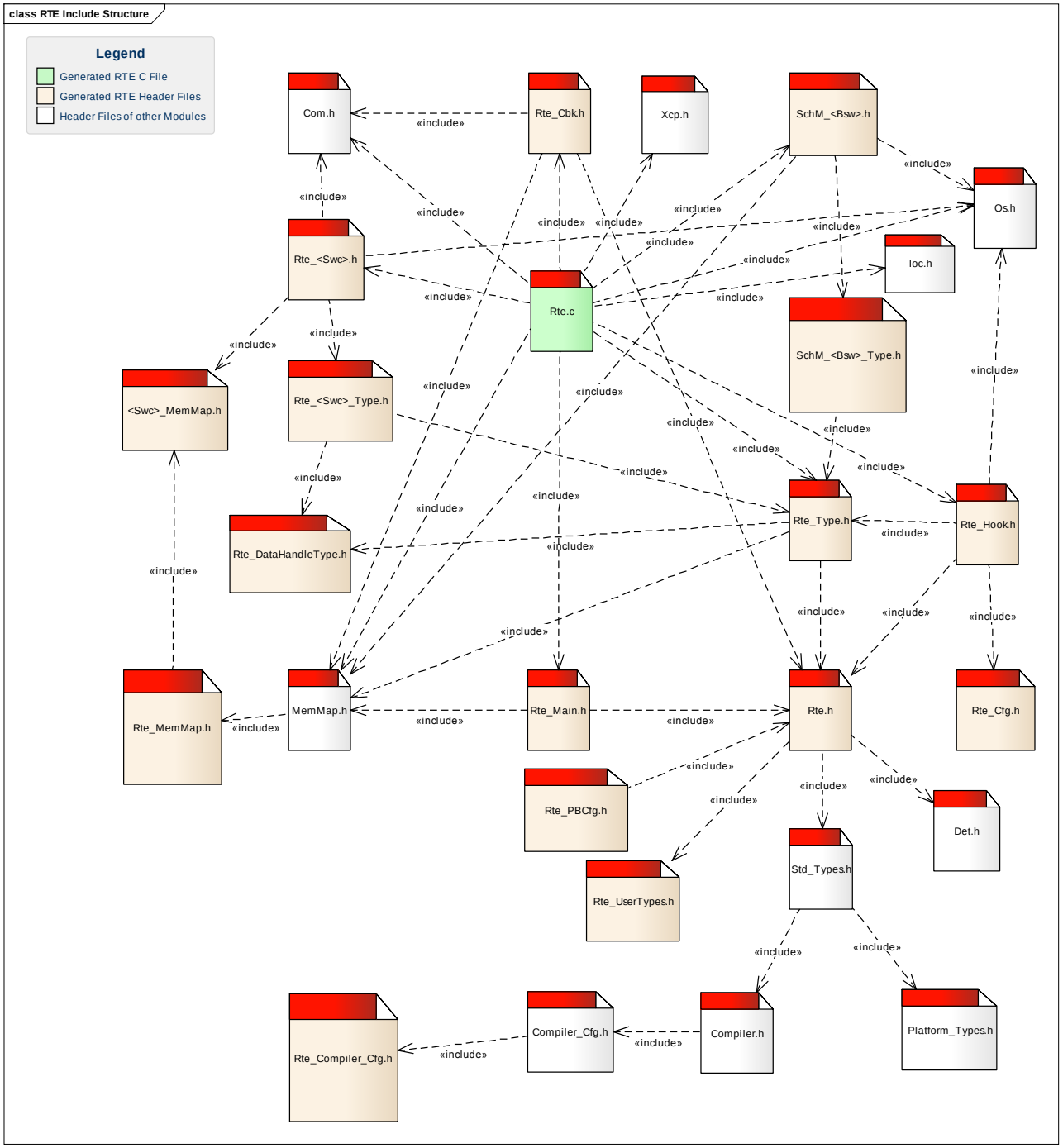


Figure 4-1 RTE Include Structure

4.4.2 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.

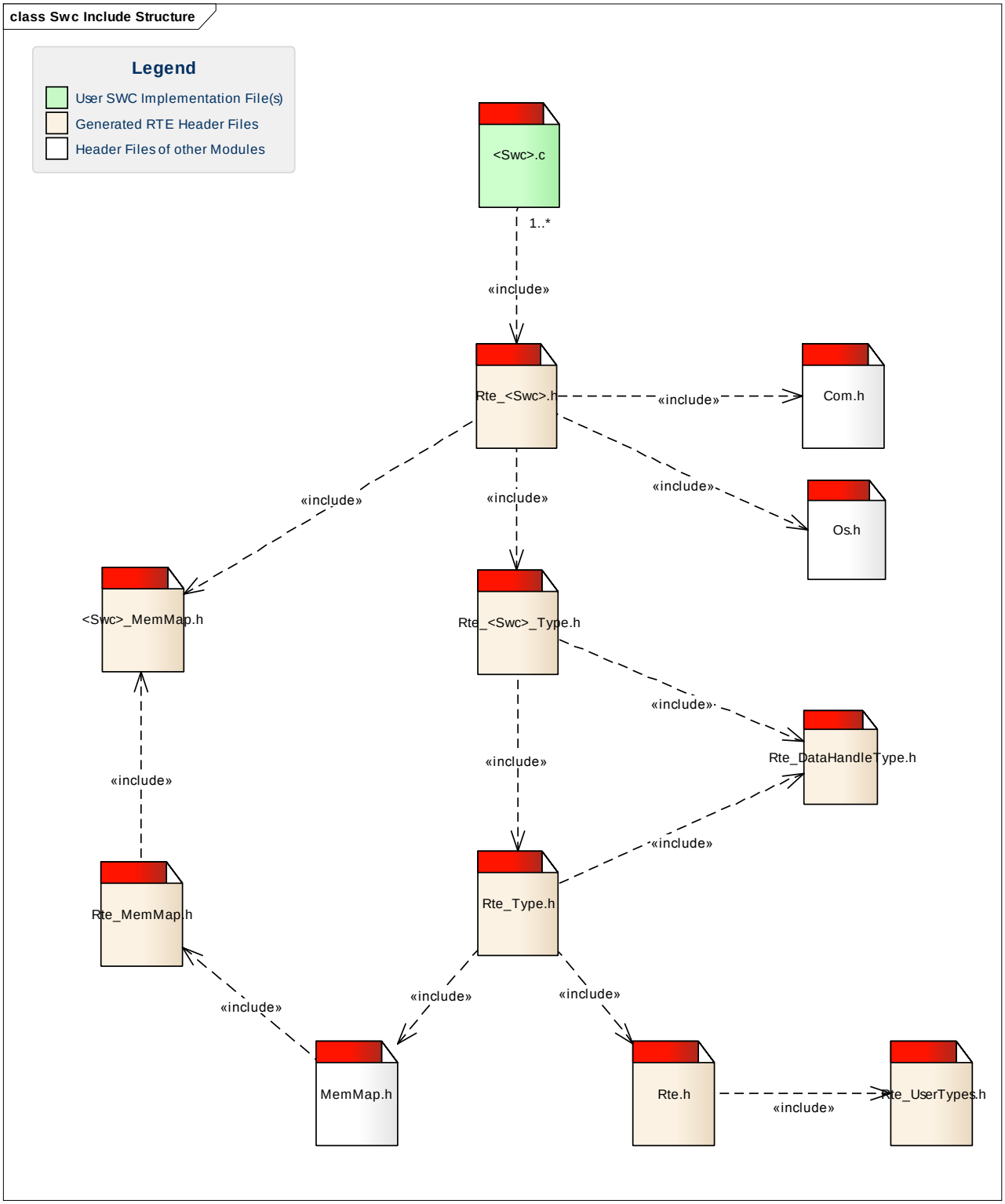


Figure 4-2 SWC Include Structure

4.4.3 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.

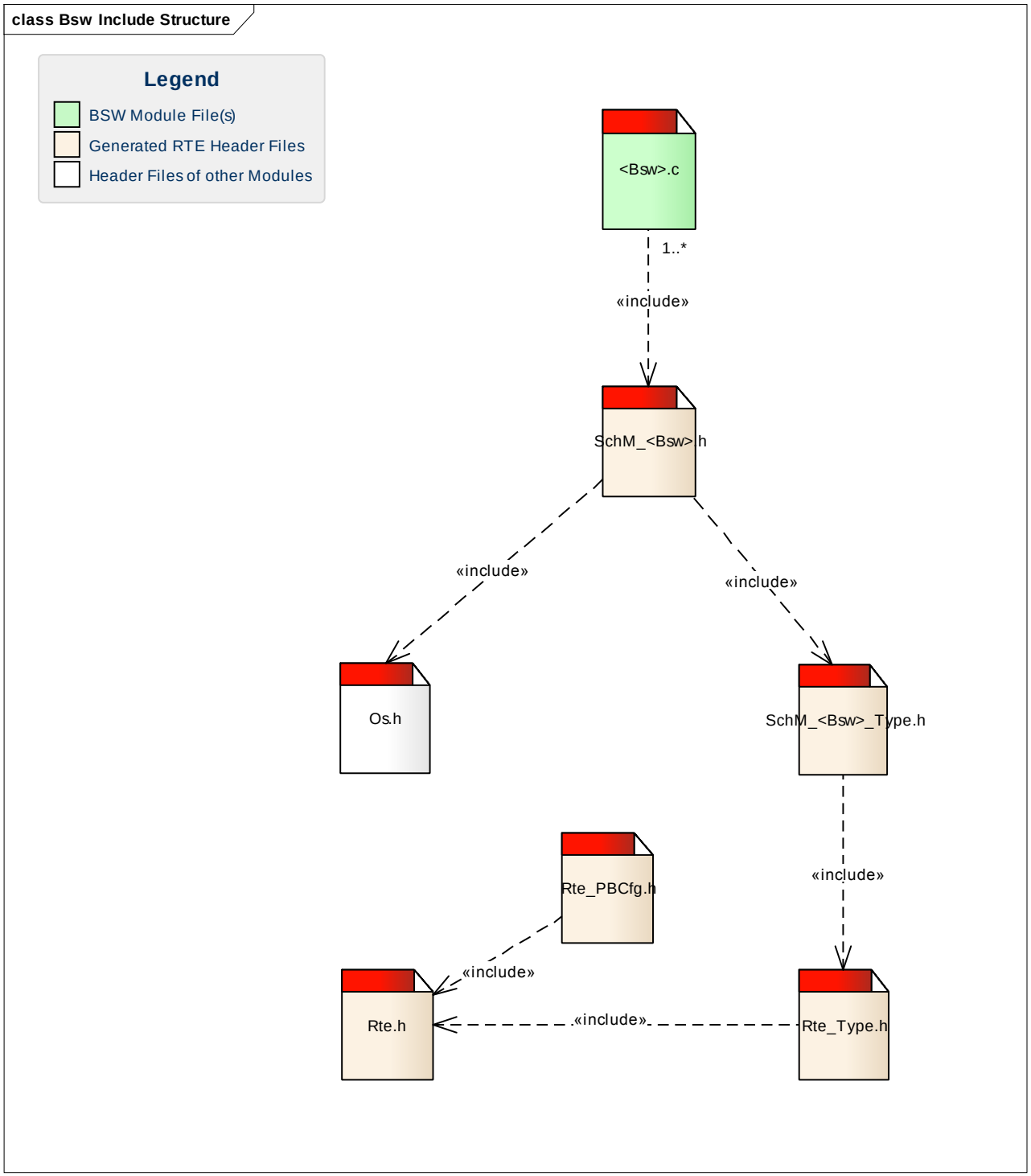


Figure 4-3 BSW Include Structure

4.5 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											■									

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

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															■							

Table 4-8 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 4-9 Compiler abstraction and memory mapping for non-cacheable variables

The memory mapping sections and compiler abstraction defines specified in Table 4-9 are only used for variables which are shared between different cores on multicore systems. These variables must be linked into non-cacheable memory.

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

- Rte_Compiler_Cfg.h
- Rte_MemMap.h

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

4.5.1 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. The following table shows the attributes, which have to be defined in DaVinci Developer 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 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 6.6):

```
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
```

4.5.2 Memory Sections for Software Components

The MICROSAR 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.

The SWC type specific parts of `MemMap.h` depend on the configuration. The MICROSAR RTE generator creates a template for each SWC according the following naming rule:

- <Swc>_MemMap.h

4.5.3 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.

4.6 Memory Protection Support

The MICROSAR 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 OS because the RTE requires read access to the data from all partitions what is not required by AUTOSAR. Moreover when trusted functions are used, the RTE uses wrapper functions that are only generated by MICROSAR 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).

4.6.1 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 6.3.

4.6.2 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.

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 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.

4.6.3 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 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.

4.6.3.1 Trusted RTE and BSW

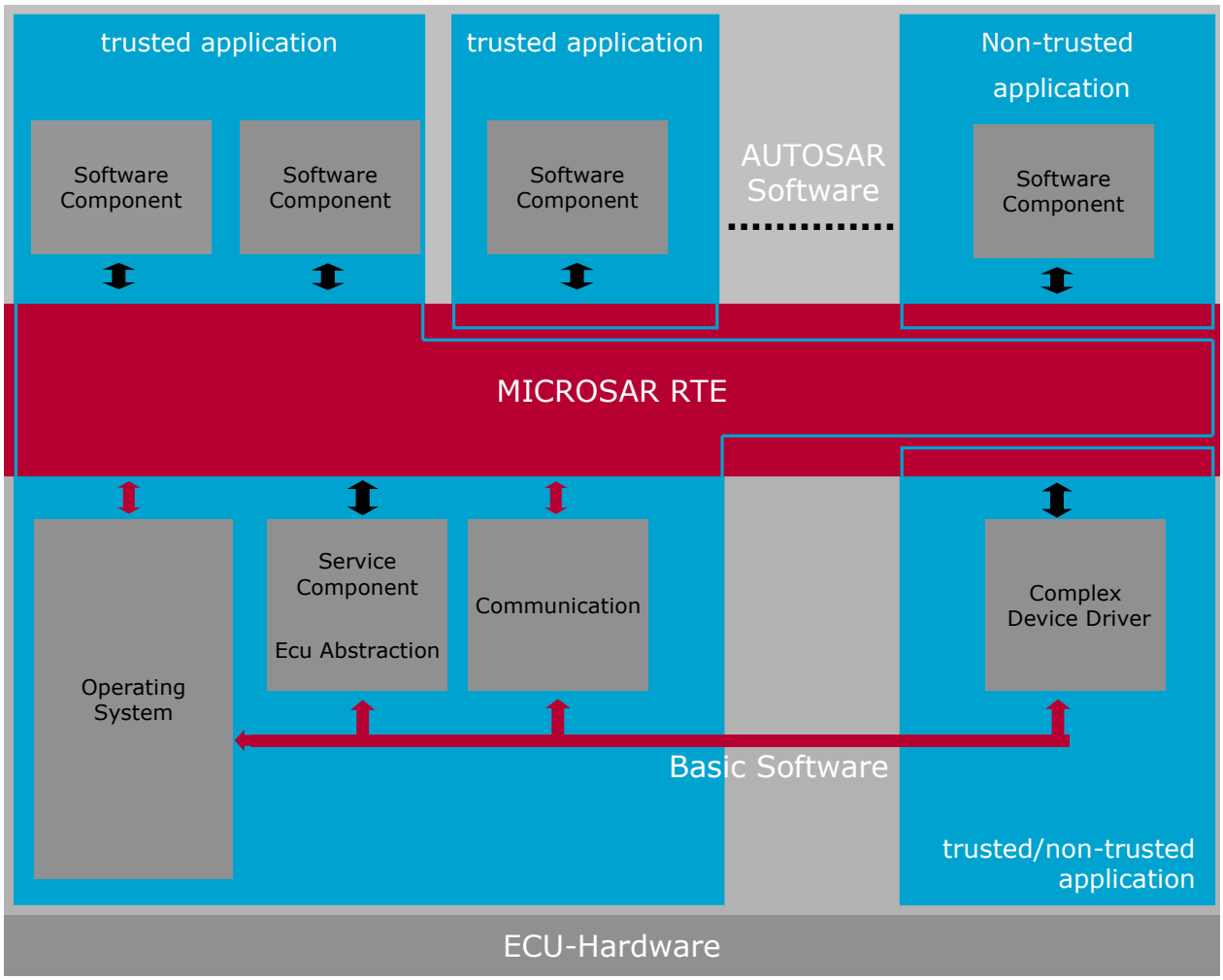


Figure 4-4 Trusted RTE Partitioning example

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.

4.6.3.2 Non-Trusted RTE and BSW

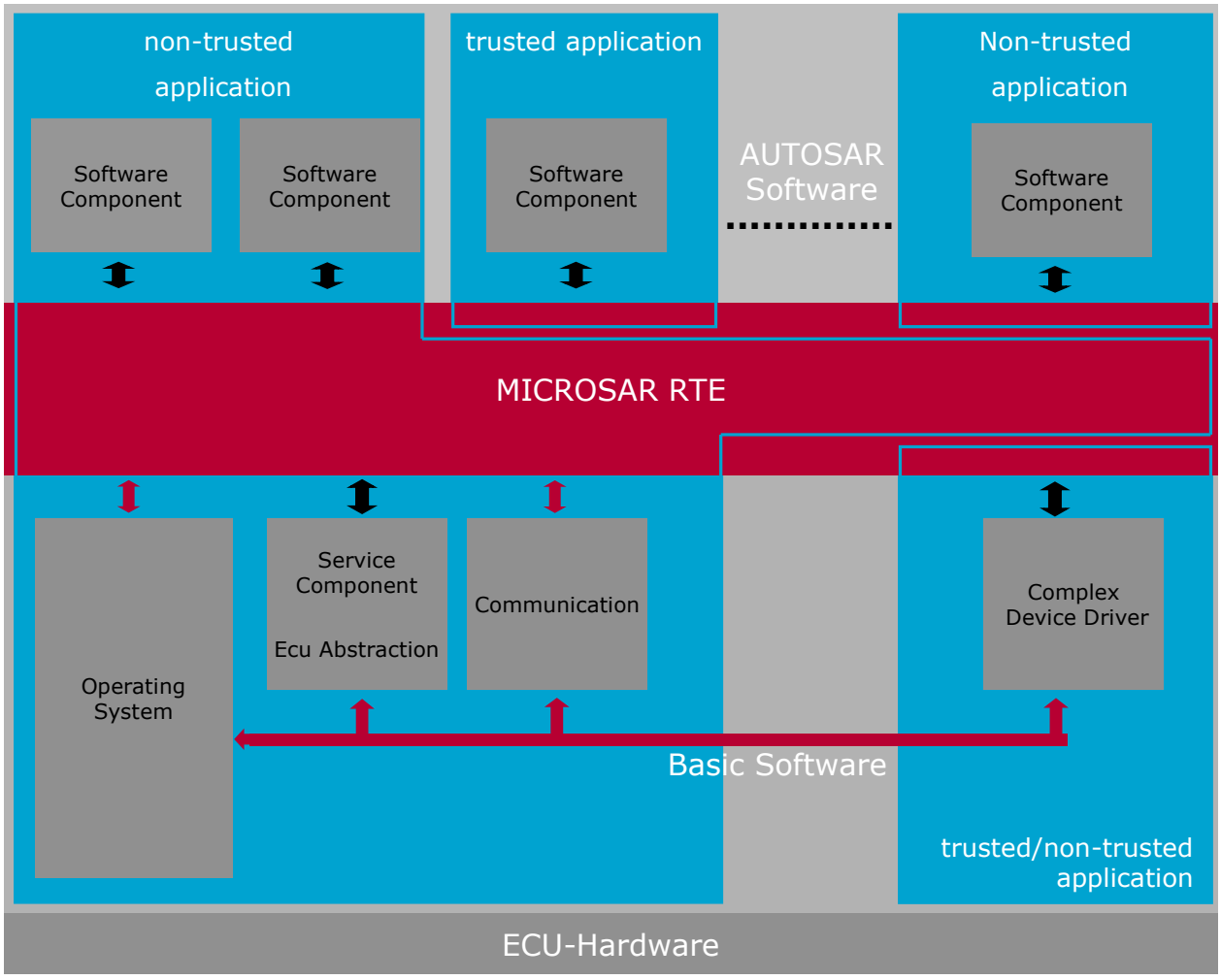


Figure 4-5 Non-trusted RTE Partitioning example

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 6.2).
- 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 mode connections with mode disabling dependencies or mode triggers between different 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 4.8.1)
- 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 4.8.2)
- reading modes with the Rte_Mode API
- explicit and implicit exclusive areas

4.6.4 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 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 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 4.6.3.2) 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. 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 RTE assumes that the calls of the COM callbacks are done from the OS application that contains the BSW Scheduler tasks.

4.6.5 Memory Protection Integration Hints

4.6.5.1 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.

4.6.5.2 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 4.5.

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.

4.6.5.3 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 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` 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.

4.7 Multicore support

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

4.7.1 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 6.3.

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

4.7.2 BSW in Multicore Systems

The MICROSAR RTE assumes that all BSW modules with direct RTE interaction (e.g. COM and NVM) are located in a single BSW OS Application on a single BSW core. The only exceptions are BSW modules like OS and ECUM that need to be available on all cores and service BSW like the WdgM with special multicore support. See chapter 4.7.3 for details. The BSW OS Application is the OS Application that contains the tasks with the schedulable entities. The RTE assumes that all COM and NVM callbacks are called from this BSW OS Application.

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 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.

4.7.3 Service BSW in Multicore Systems

The MICROSAR 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.

Also the `WdgM_Mainfunction` 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.

4.7.4 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 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` parameter in the `RteGeneration` parameters.

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

4.8 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.

4.8.1 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 a schedulable entitiy `Rte_ComSendSignalProxyPeriodic`, which periodically calls the COM APIs when a partition without BSW transmitted data.

The schedulable entity `Rte_ComSendSignalProxyPeriodic` should be mapped to the same task as `Com_MainFunctionTx` with a lower position in task so that it can update the signals before they are transmitted by 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.

4.8.2 Client Server Communication

Also client server calls between SWCs in different partitions are 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. Here, doing 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.

5 API Description

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

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.

For an interfaces overview please see Figure 2-2.

5.1 Data Type Definition

The MICROSAR 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`.

Floating point data types with double precision may not be used for data elements with external connectivity, because the MICROSAR COM layer lacks support for 64 bit data types.

5.1.1 Invalid Value

The MICROSAR 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.

5.1.2 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.

5.1.3 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.

5.2 API Error Status

Most of the RTE APIs provide an error status in the API return code. For easier evaluation the MICROSAR 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.

5.3 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.

5.3.1 <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 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.

The optional activation parameter is valid for all above described trigger conditions with the exception of 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 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.

5.3.2 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
Currently it is not supported to define a runnable activation reason over partition boundaries in multicore and memory protection systems.

5.4 SWC Exclusive Areas

5.4.1 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 3.6.10.	
Call Context	
This function can be used inside runnable entities.	

5.4.2 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 3.6.10.	
Call Context	
This function can be used inside runnable entities.	

5.5 BSW Exclusive Areas

5.5.1 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 3.6.10.
Call Context
This function can be used inside a schedulable entity in Task or Interrupt context.

5.5.2 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 3.6.10.
Call Context
This function can be used inside a schedulable entity in Task or Interrupt context.

5.6 Sender-Receiver Communication

5.6.1 Rte_Read

Prototype	
`Std_ReturnType` **`Rte_Read_<p>_<d>`** `( [IN Rte_Instance instance,] OUT <DataType> *data [, OUT Rte_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_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_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).	

5.6.2 Rte_DRead

Prototype	
`<DataType> **Rte_DRead_<p>_<d>** ( [IN Rte_Instance instance][, OUT Rte_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).	

5.6.3 Rte_Write

Prototype	
`Std_ReturnType` **`Rte_Write_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> data [, OUT Rte_TransformerError transformerError] )` `Std_ReturnType` **`Rte_Write_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> *data [, OUT Rte_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.
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).	

5.6.4 Rte_Receive

Prototype	
`Std_ReturnType` **`Rte_Receive_<p>_<d>`** `( [IN Rte_Instance instance,] OUT <DataType> *data [, OUT uint16 *length][, OUT Rte_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.
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).	

5.6.5 Rte_Send

Prototype	
`Std_ReturnType` **`Rte_Send_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> data [, OUT Rte_TransformerError transformerError] )` `Std_ReturnType` **`Rte_Send_<p>_<d>`** `( [IN Rte_Instance instance,] IN <DataType> *data [, IN uint16 length] [, OUT Rte_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.
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.
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).	

5.6.6 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!	

5.6.7 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!

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.

5.6.8 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!	

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.

5.6.9 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.
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!	

5.6.10 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 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 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 indicated 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).	

5.6.11 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).	

5.7 Data Element Invalidation

5.7.1 Rte_Invalidate

Prototype	
`Std_ReturnType` **`Rte_Invalidate_<p>_<d>`** `( [IN Rte_Instance instance] [, OUT Rte_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).
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).	

5.7.2 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!	

5.8 Mode Management

5.8.1 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).	

5.8.2 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).	

5.8.3 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.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC).	

5.8.4 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 indicated the status of a mode switch, triggered by an `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).	

5.9 Inter-Runnable Variables

5.9.1 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!	

5.9.2 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!

5.9.3 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!	

5.9.4 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!	

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.

5.10 Per-Instance Memory

5.10.1 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.

5.11 Calibration Parameters

5.11.1 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.	

5.11.2 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.

5.12 Client-Server Communication

5.12.1 Rte_Call

Prototype	
`Std_ReturnType` **`Rte_Call_<p>_<o>`** `( [IN Rte_Instance instance,] {IN type [*]inputparam,}* {OUT type *outputparam,}* {INOUT type *inoutputparam,}* )`	
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.
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.
RTE_E_TIMEOUT	Returned by a synchronous call after the timeout has expired and no other error occurred. The arguments are not changed.
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).	

5.12.2 Rte_Result

Prototype	
`Std_ReturnType` **`Rte_Result_<p>_<o>`** `( [IN Rte_Instance instance,] {OUT type *outputparam,}* {INOUT type *inoutputparam,}* )`	
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.
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.
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).	

5.13 Indirect API

5.13.1 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).	

5.13.2 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).	

5.13.3 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).	

5.14 RTE Lifecycle API

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

5.14.1 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 have been initialized especially OS and 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 runnables. Therefore Rte_Start on the other cores has to be executed first.	

5.14.2 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 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.	

5.14.3 Rte_InitMemory

Prototype
`void Rte_InitMemory ( void )`
Parameter
-
Return code
-
Functional Description
The API function `Rte_InitMemory` is a MICROSAR 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 has to be called on all cores that are used by the RTE.

Caution
Rte_InitMemory API is a Vector extension to the AUTOSAR standard and may not be supported by other RTE generators.

5.15 SchM Lifecycle API

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

5.15.1 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 initializes the BSW Scheduler and resets the timers for all cyclic triggered schedulable entities (main functions). Note that all main functions calls are activated upon return from this function.
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.

5.15.2 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.

5.15.3 SchM_GetVersionInfo

Prototype
`void` **`SchM_GetVersionInfo`** `(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.

5.16 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.

5.16.1 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, Send, Receive, Invalidate, SwitchAck, Switch, Call, Result, IrvWrite, IrvRead The <AccessPoint> is defined as follows: ▶ Enter, Exit: <ExclusiveArea> ▶ Write, Read, Send, Receive, Feedback, Invalidate: <PortPrototype>_<DataElementPrototype> ▶ Switch, SwitchAck: <PortPrototype>_<ModeDeclarationGroupPrototype> ▶ Call, Result: <PortPrototype>_<OperationPrototype> ▶ IrvWrite, IrvRead: <InterRunnableVariable>	
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 runnable context, the context of the trace hooks is also the runnable entity of an AUTOSAR software component (SWC).	

5.16.2 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, Send, Receive, Invalidate, Feedback, Switch, SwitchAck, Call, Result, IrvWrite, IrvRead The <AccessPoint> is defined as follows: ▶ Enter, Exit: <ExclusiveArea> ▶ Write, Read, Send, Receive, Feedback, Invalidate: <PortPrototype>_<DataElementPrototype> ▶ Switch, SwitchAck: <PortPrototype>_<ModeDeclarationGroupPrototype> ▶ Call, Result: <PortPrototype>_<OperationPrototype> ▶ IrvWrite, IrvRead: <InterRunnableVariable>

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 runnable context, the context of the trace hooks 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 macro that does a direct runnable invocation. If VFB trace hooks are enabled for such an Rte_Call API or for the called server runnable, these optimizations are no longer possible.
Also macro optimizations for Rte_Read, Rte_DRead, Rte_Write, Rte_IrvRead and Rte_IrvWrite APIs are disabled when VFB tracing for that APIs is 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_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

5.16.3 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 hooks 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.

5.16.4 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 hooks 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.

5.16.5 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` 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 runnable context, the context of the trace hooks 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.

5.16.6 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 runnable context, the context of the trace hooks 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.

5.16.7 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 runnable context, the context of the trace hooks 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.

5.16.8 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` 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 runnable context, the context of the trace hooks 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!

5.16.9 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 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!

5.16.10 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.	

5.16.11 Rte_[<client>_]Task_Dispatch

Prototype	
`void` **`Rte_[<client>_]Task_Dispatch`** `( TaskType task )`	
Parameter	
task	The parameter indicates the task to 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.	

5.16.12 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 COM callbacks. For API functions, the call context is the runnable context. Note: For COM callbacks the context could be the task context or the interrupt context!

5.16.13 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.

5.16.14 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.	

5.16.15 Rte_[<client>_]Runnable_<cts>_<re>_Start

Prototype	
`void` **`Rte_[<client>_]Runnable_<cts>_<re>_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>_<re>_Return`.	
Call Context	
This function is called inside RTE generated task bodies.	

5.16.16 Rte_[<client>_]Runnable_<cts>_<re>_Return

Prototype	
`void` **`Rte_[<client>_]Runnable_<cts>_<re>_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>_<re>_Start`.	
Call Context	
This function is called inside RTE generated task bodies.	

5.17 RTE Interfaces to BSW

The RTE has standardized Interfaces to the following basic software modules

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

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

5.17.1 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 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.

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.

5.17.2 Interface to Transformer

Used Transformer API
ComXf_<transformerId>
ComXf_Inv_<transformerId>
SomeIpXf_<transformerId>
SomeIpXf_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.

5.17.3 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
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 OS specific)
IocWrite
IocRead
IocWriteGroup
IocReadGroup
IocSend
IocReceive

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!

5.17.4 Interface to NVM

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.

5.17.5 Interface to XCP

In addition to the usage of the Com and the OS module as described by AUTOSAR, the MICROSAR RTE generator optionally can also take advantage of the MICROSAR 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 6.6 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

5.17.6 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 4.8.1 for details.

Provided Schedulable Entity
Rte_ComSendSignalProxyPeriodic

5.17.7 Interface to DET

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

See chapter 3.8.1 for details.

Used DET API
Det_ReportError

6 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

6.1 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.

6.2 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 RTE generation.

The creation of tasks is done in OS Configuration Editor in the in the DaVinci Configurator. The **Task Mapping Assistant** has to be used to assign the triggered functions (runnables and schedulable entities) to the tasks.

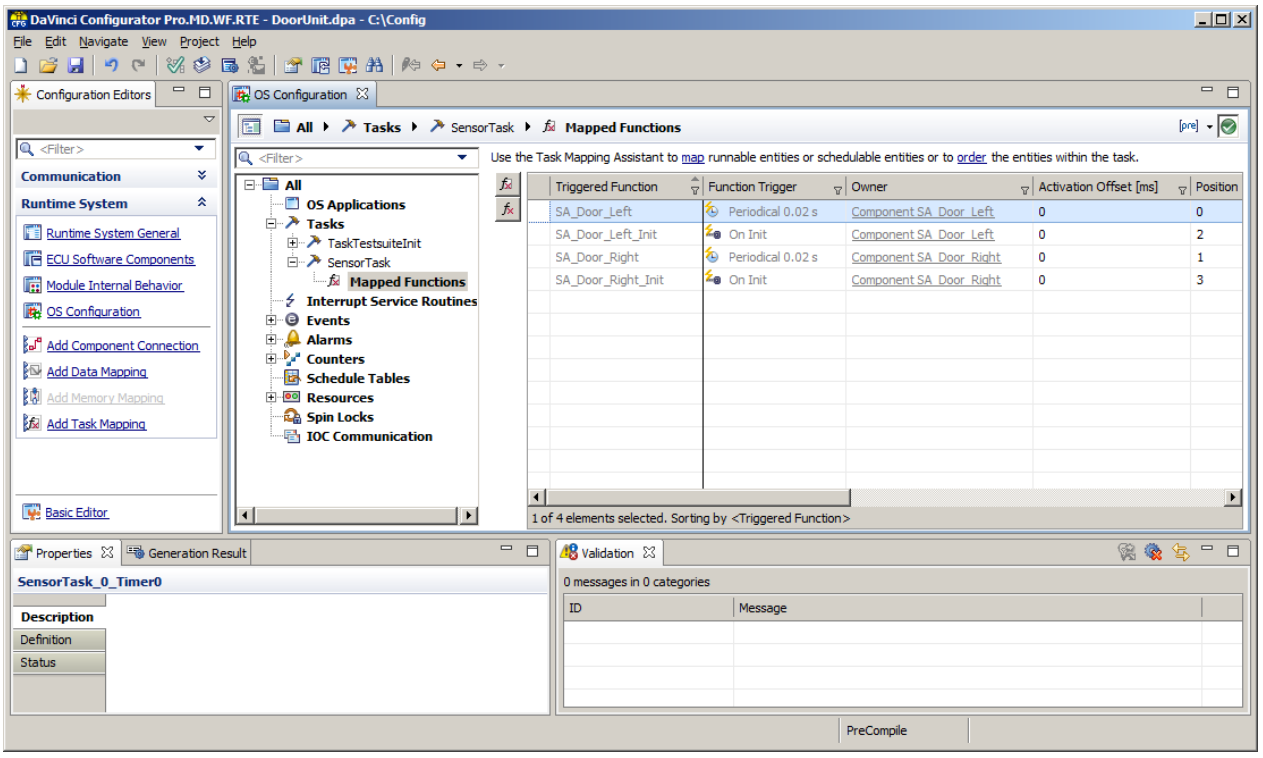


Figure 6-1 Mapping of Runnables to Tasks

The MICROSAR 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 Typ `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 OS Schedule Tables.

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.

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.

6.3 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 contains only 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.

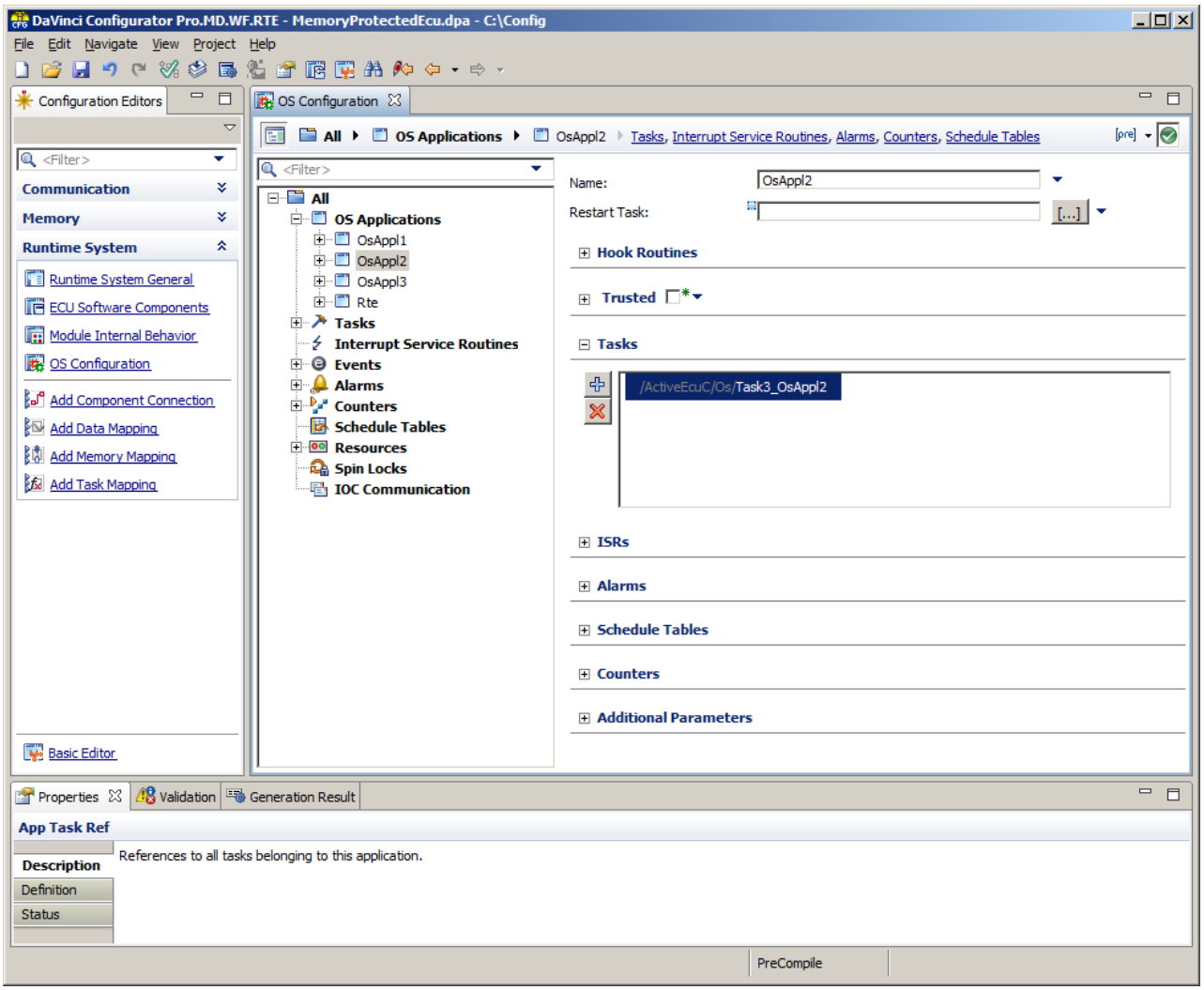


Figure 6-2 Assignment of a Task to an OS Application

Caution

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

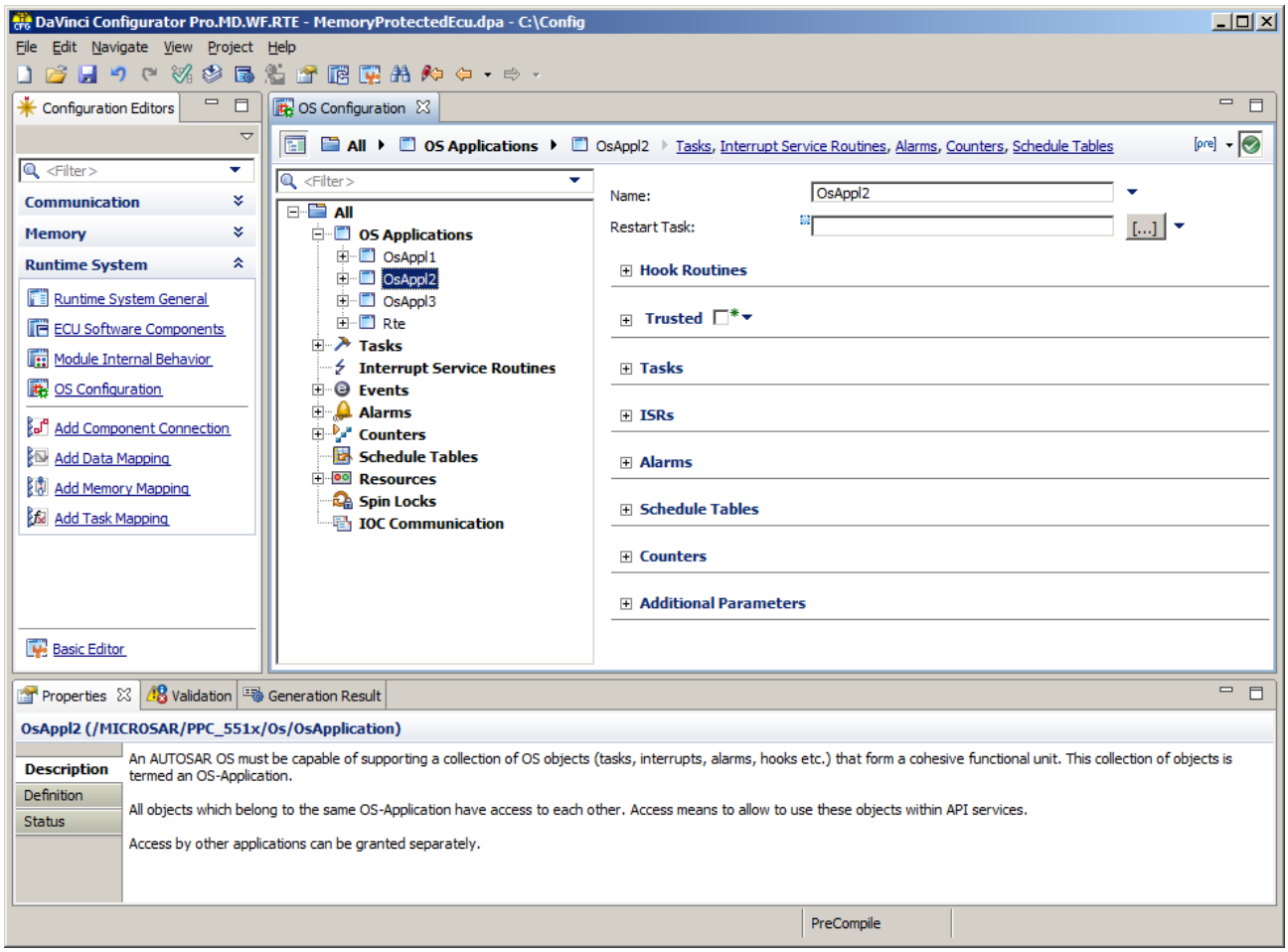


Figure 6-3 OS Application Configuration

6.4 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 has to configure the following parts of the NvM configuration.

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

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

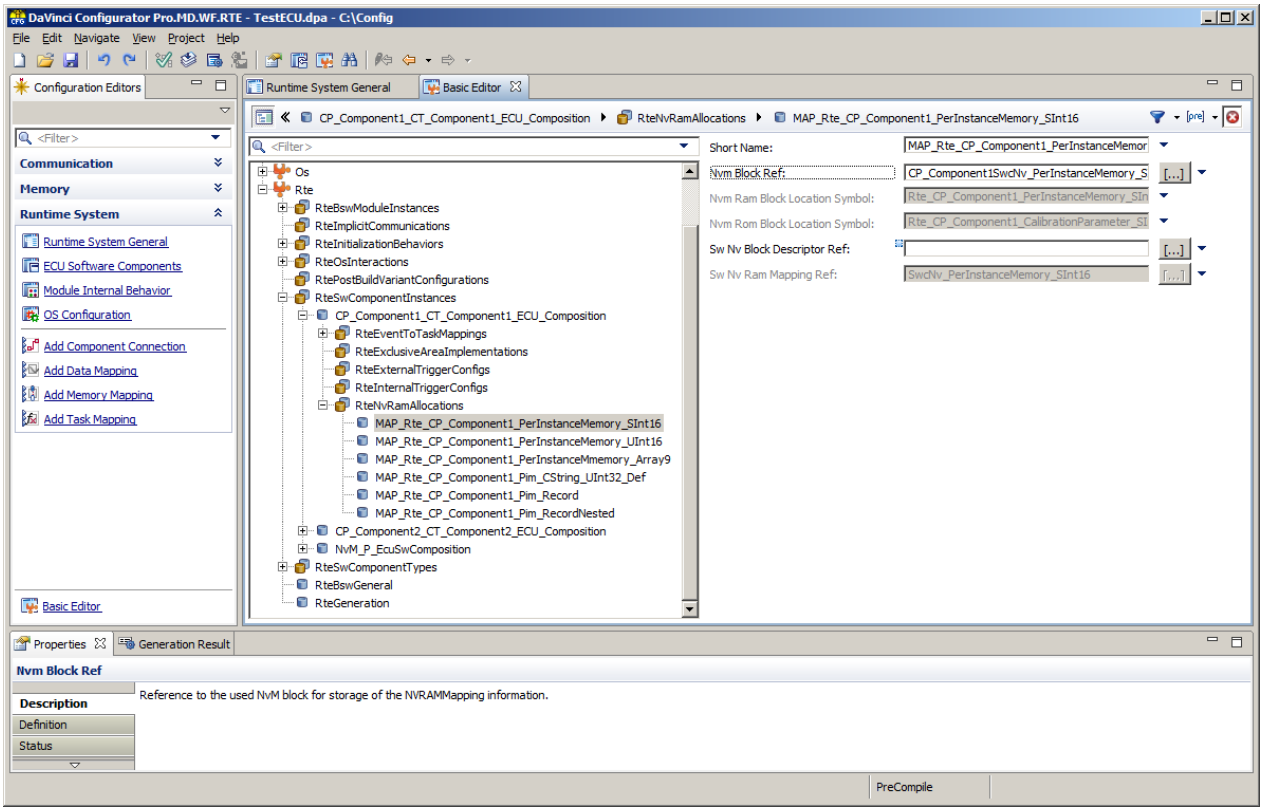


Figure 6-4 Mapping of Per-Instance Memory to NV Memory Blocks

6.5 RTE Generator Settings

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

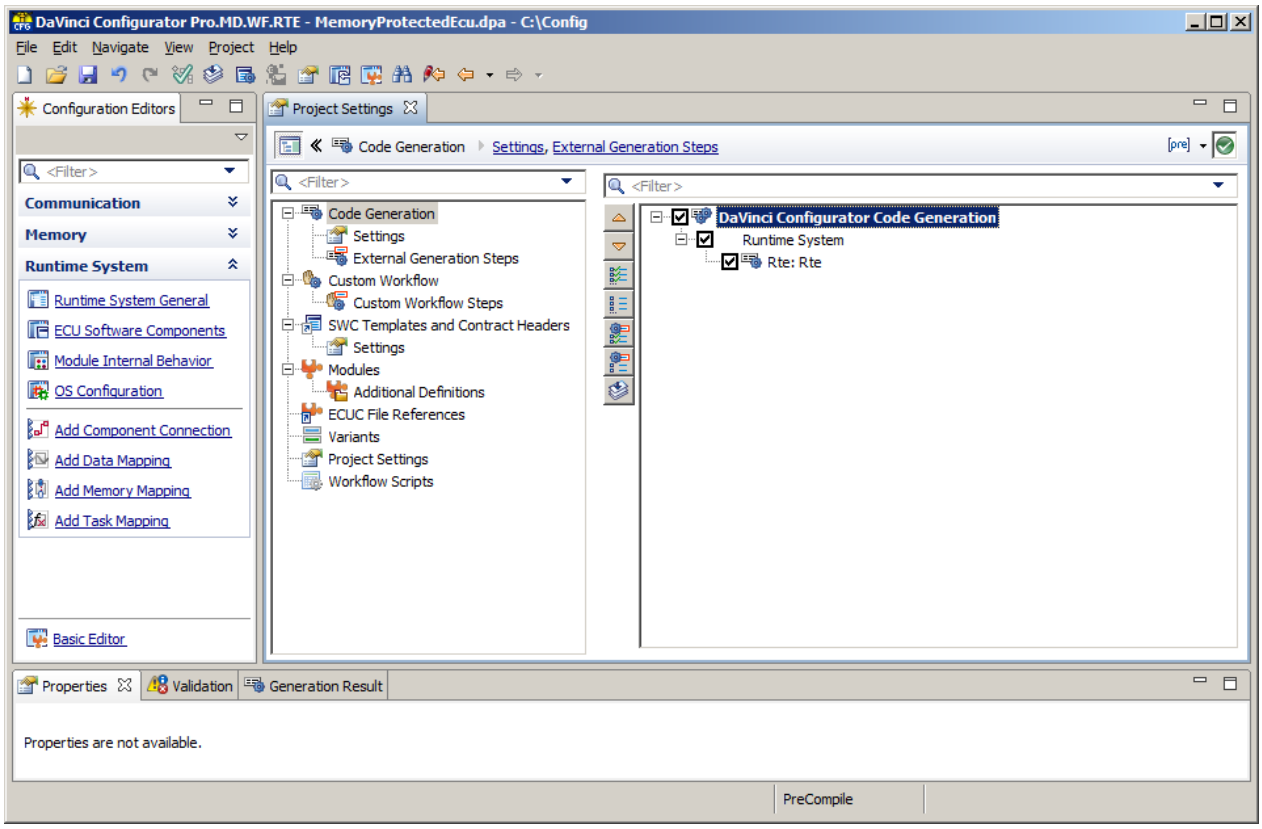


Figure 6-5 RTE Generator Settings

6.6 Measurement and Calibration

The MICROSAR 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 memories and inter-runnable variables. Calibration parameters are represented as characteristic objects.

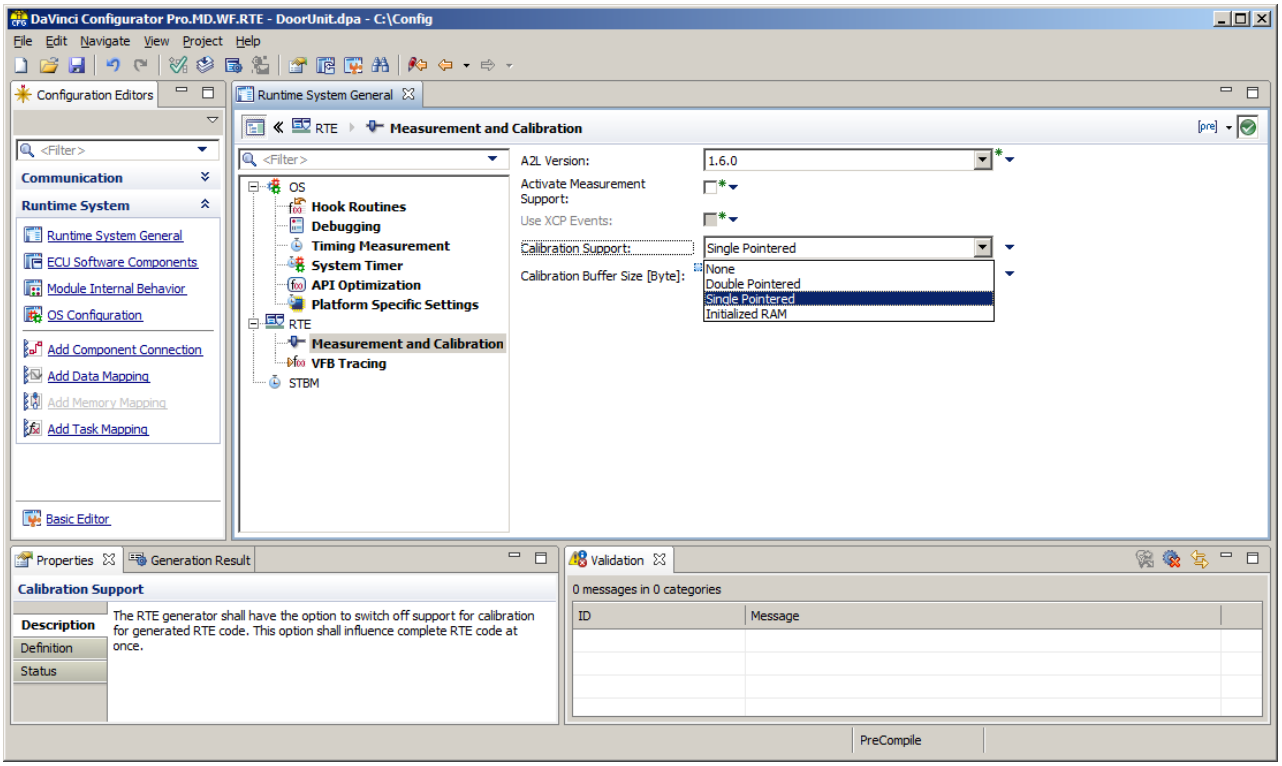


Figure 6-6 Measurement and Calibration Generation Parameters

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 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.

Within the `Rte.a2l` file, the measurement objects are grouped by SWCs. When inter-ECU sender/receiver communication shall be measured, the groups will also contain links to measurement objects with the name of the COM signal handle. These measurement objects have to be provided by the COM.

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 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.

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 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

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

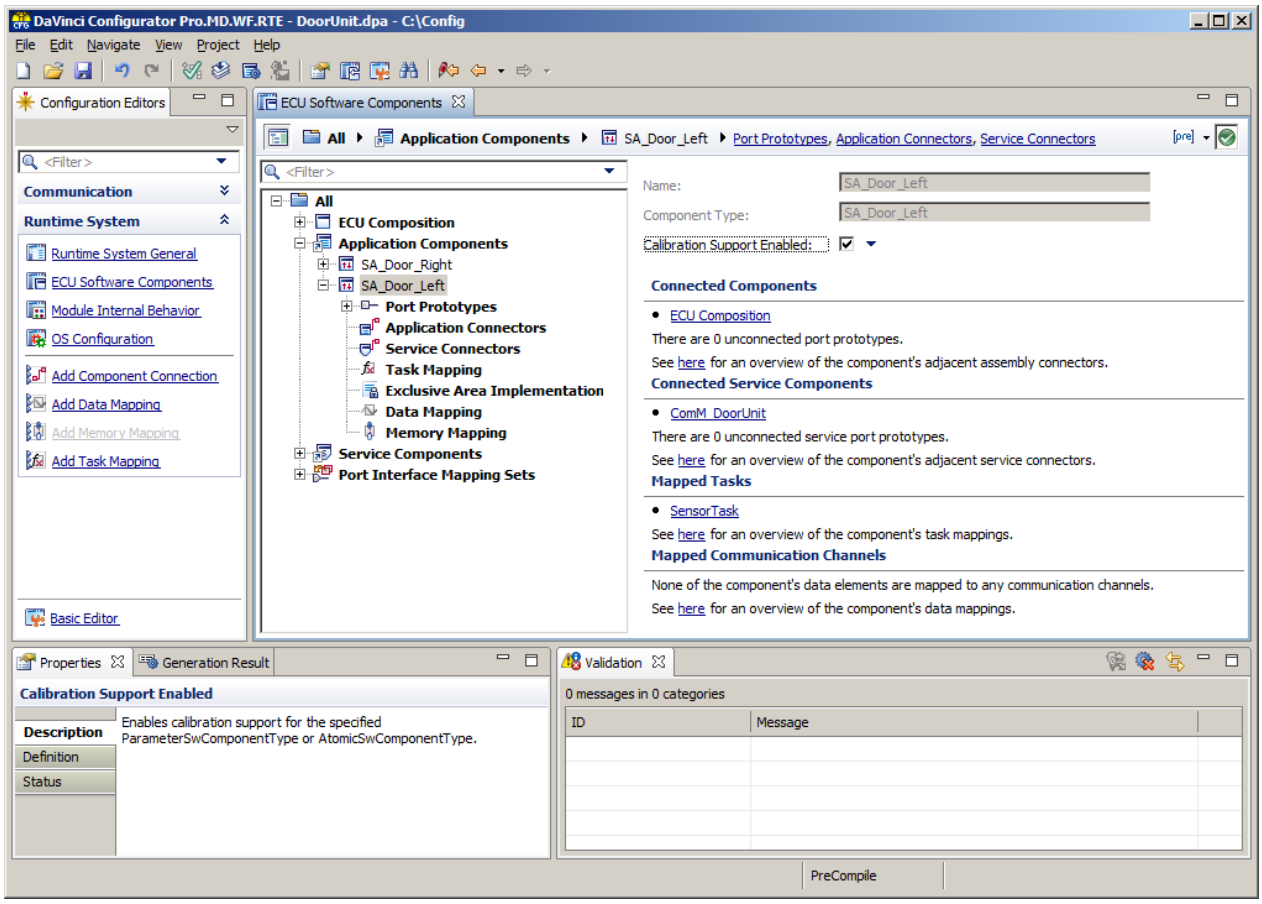


Figure 6-7 SWC Calibration Support Parameters

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.

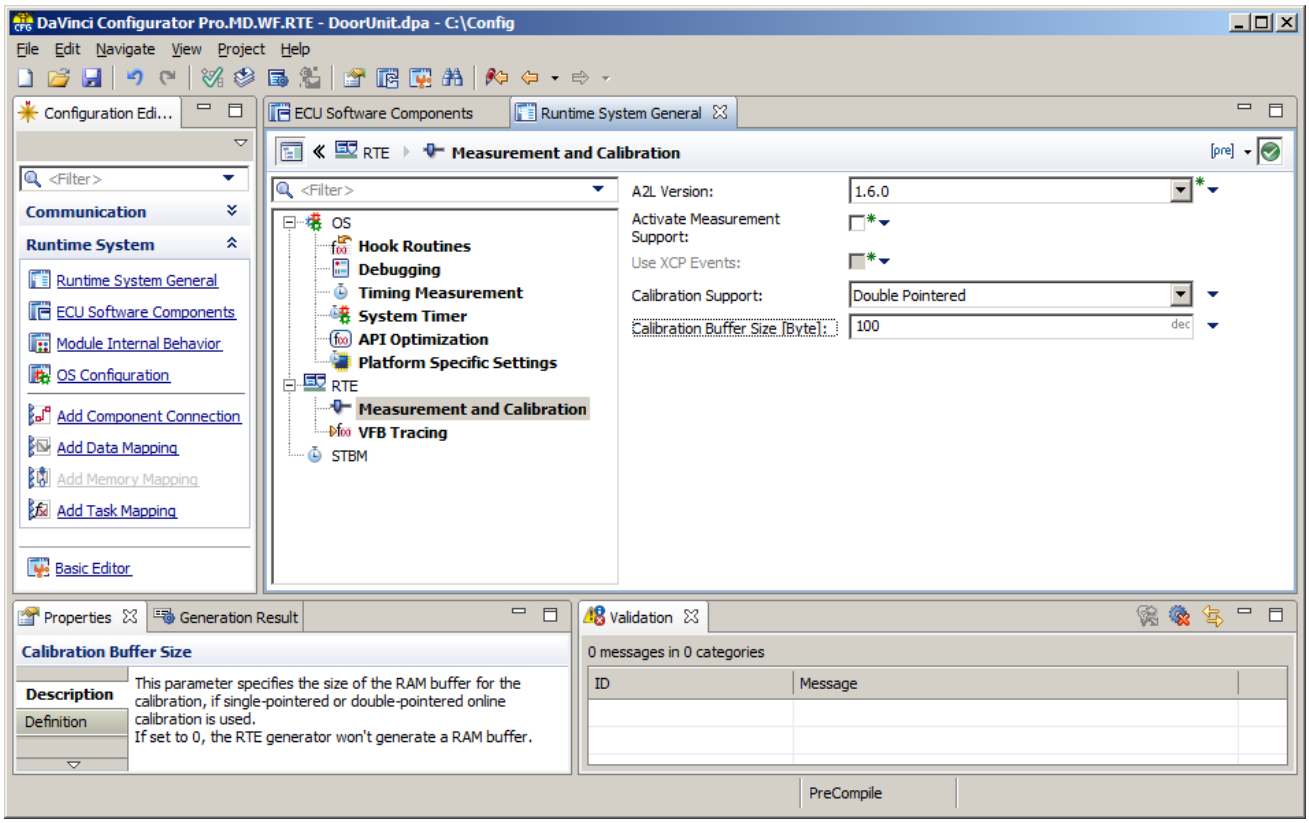


Figure 6-8 CalibrationBufferSize Parameter

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.

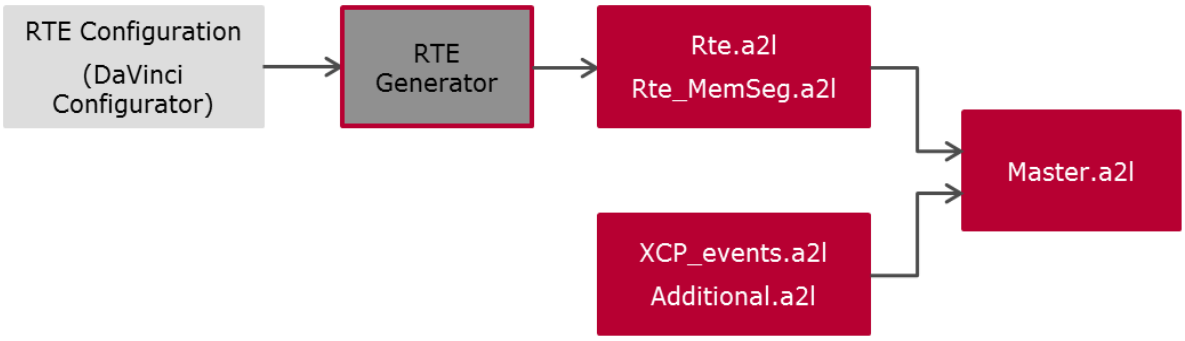


Figure 6-9 A2L Include Structure

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

6.7 Optimization Mode Configuration

A general requirement to the RTE generator is production of optimized RTE code. If possible the MICROSAR 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.

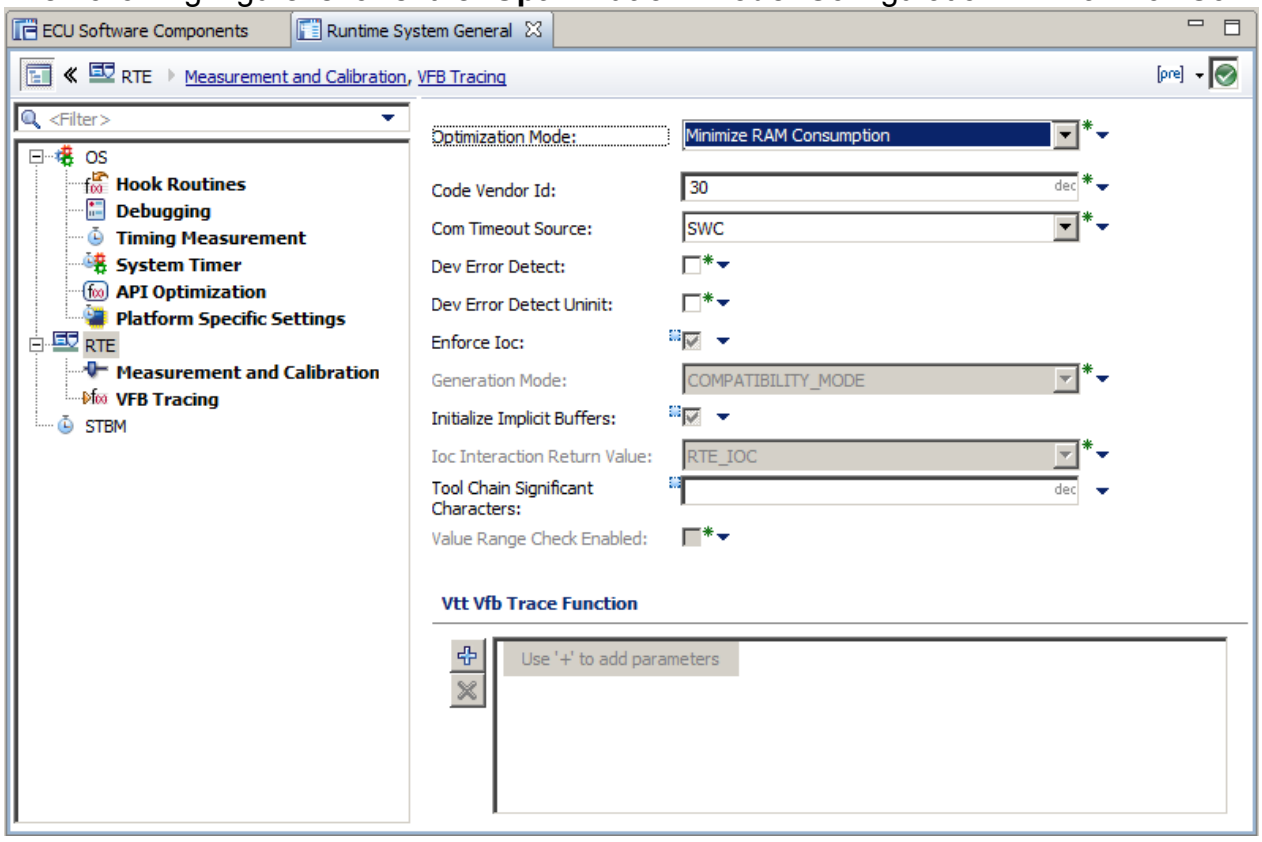


Figure 6-10 Optimization Mode Configuration

6.8 VFB Tracing Configuration

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

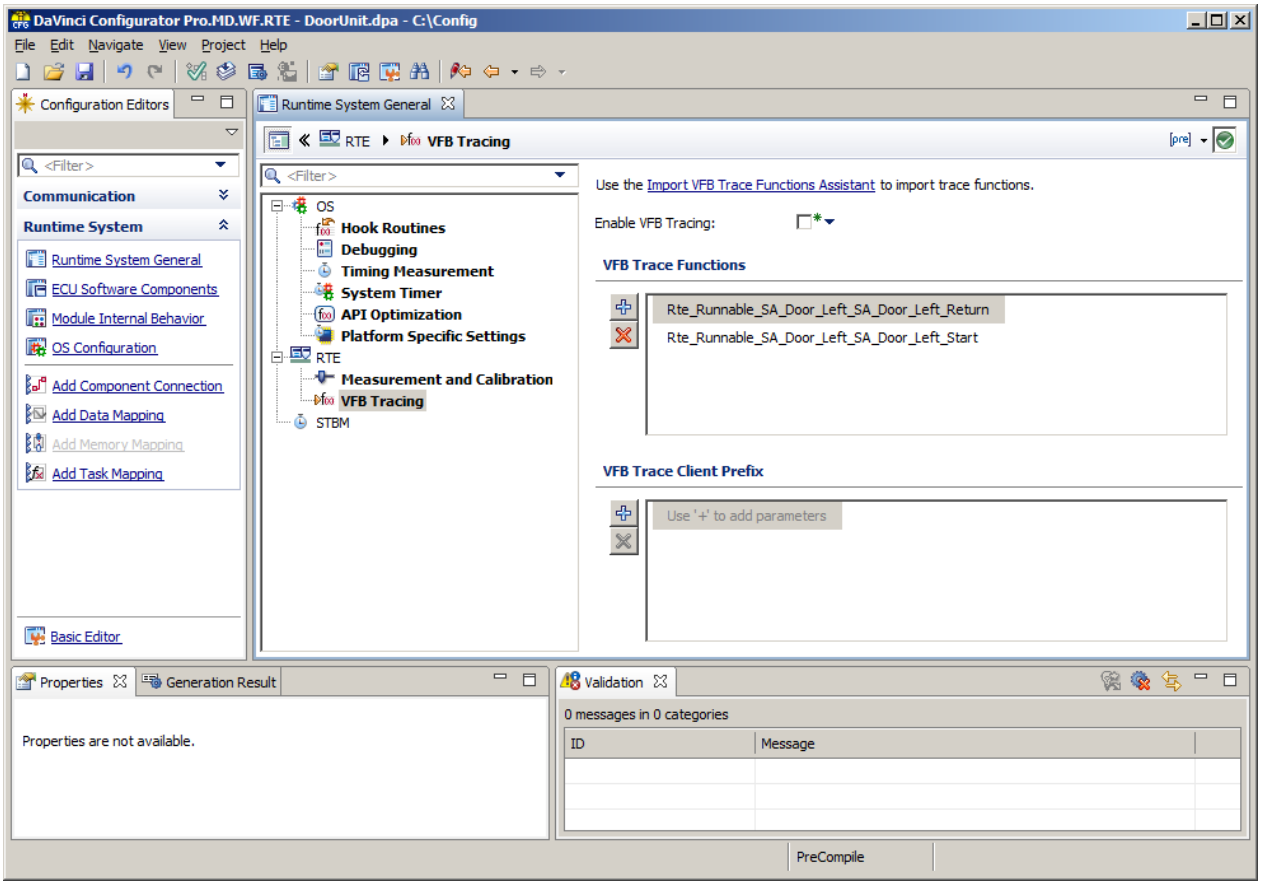


Figure 6-11 VFB Tracing Configuration

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 enable additionally 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 4.3.4 describes how a template for VFB trace hooks can be generated initially or updated after configuration changes.

6.9 Exclusive Area Implementation

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

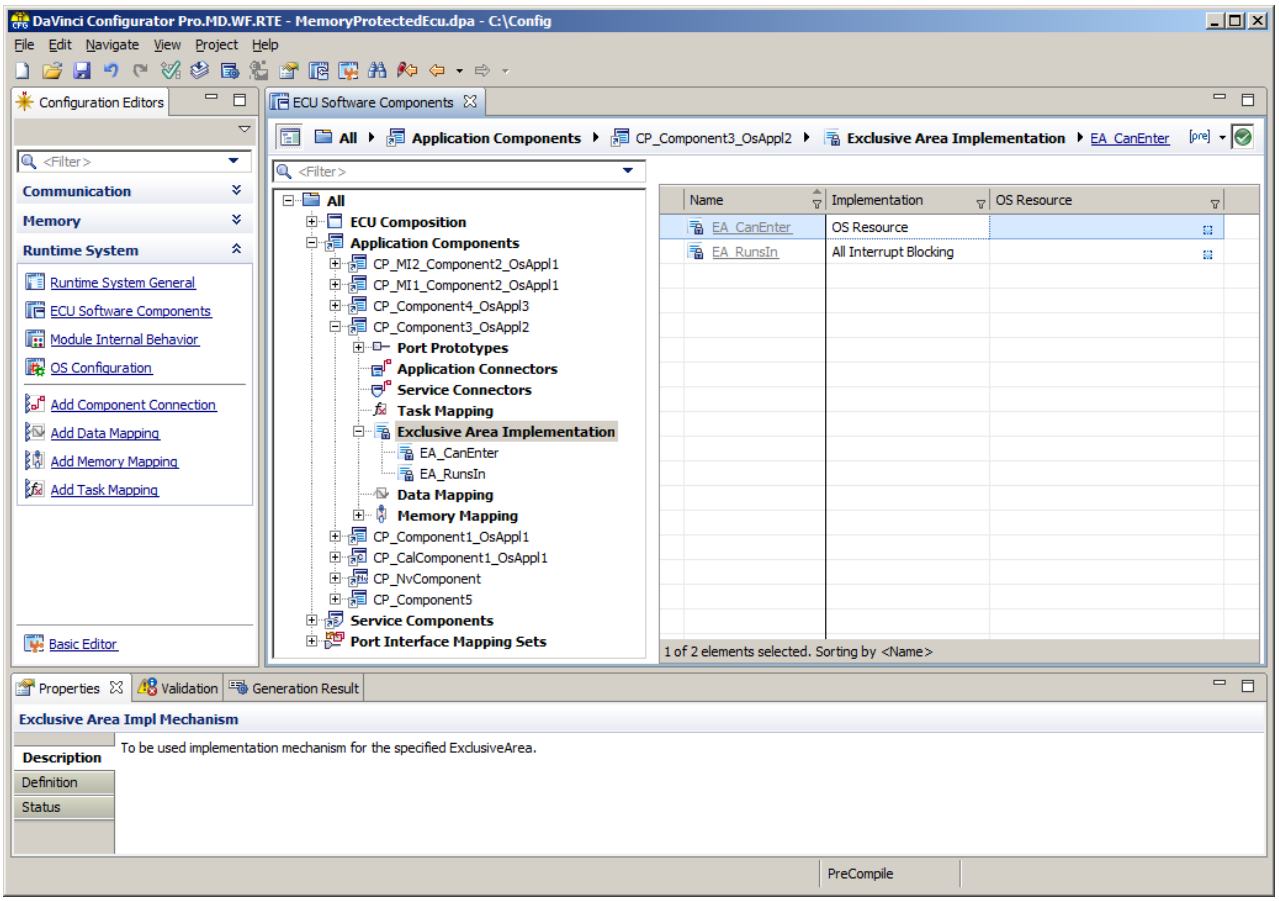


Figure 6-12 Exclusive Area Implementation Configuration

6.10 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.

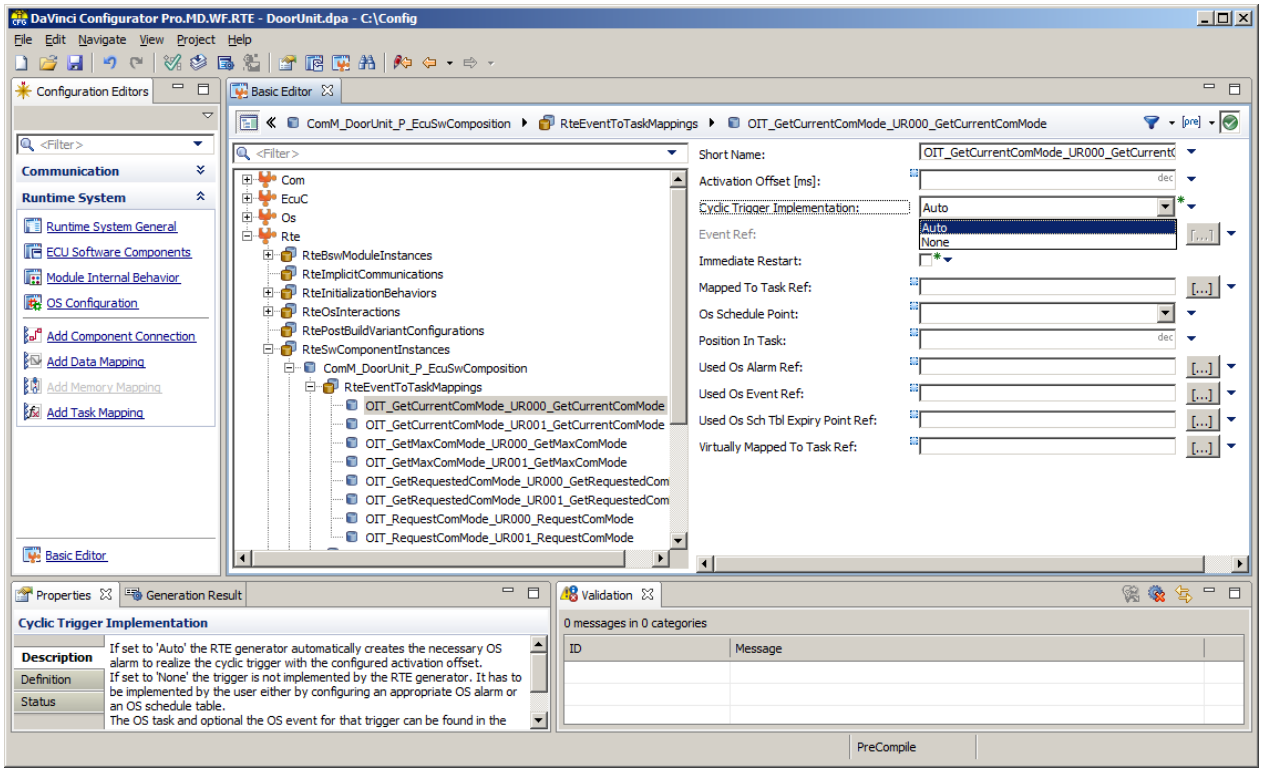


Figure 6-13 Periodic Trigger Implementation Configuration

Caution

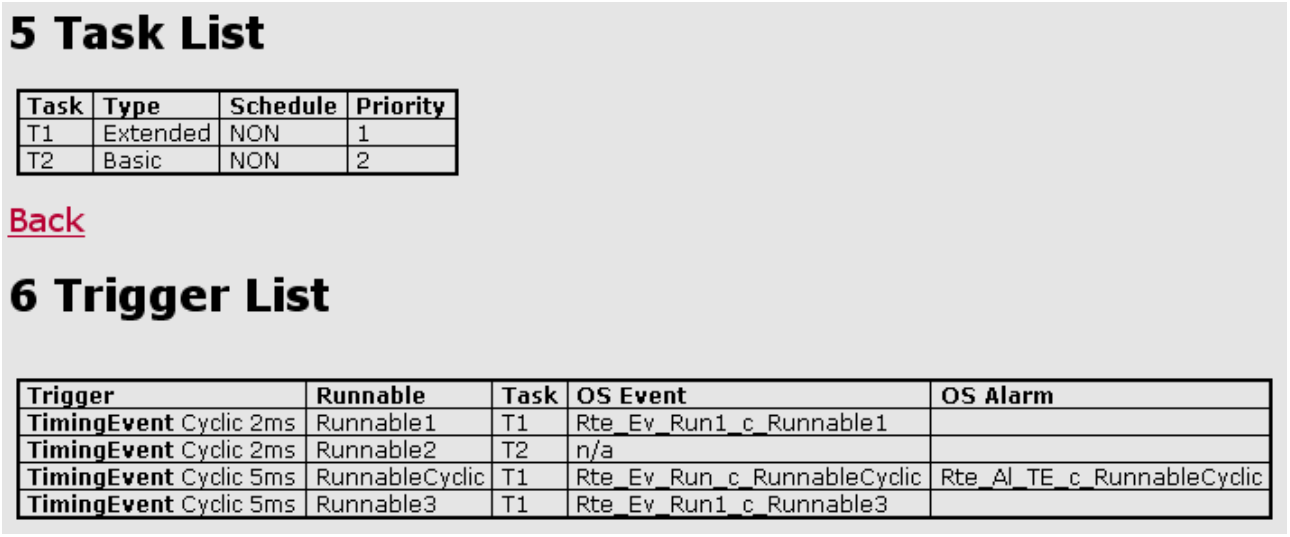
Currently it is not supported to define an activation offset and a trigger implementation per trigger. The settings can only be made for the complete runnable with potential several cyclic triggers.

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

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. 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 6-14 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.

6.11 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.

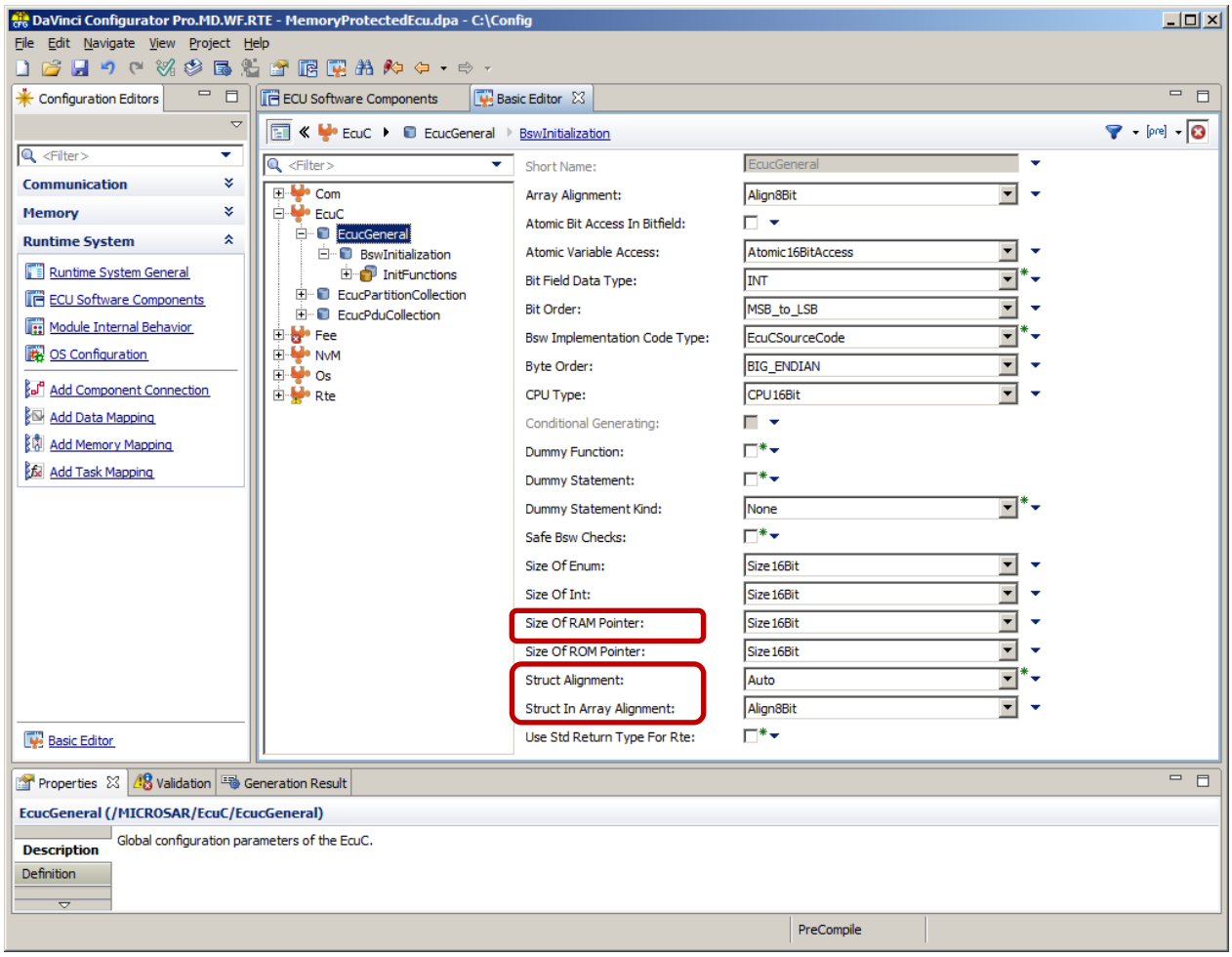


Figure 6-15 Configuration of platform settings

7 Glossary and Abbreviations

7.1 Glossary

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

Table 7-1 Glossary

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

7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
Com	Communication Layer
ComXf	Com based Transformer
C/S	Client-Server
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
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 7-2 Abbreviations

8 Additional Copyrights

The MICROSAR 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
MicrosarRteGen.exe	Perl 5.20.2	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 8-1 Free and Open Source Software Licenses

9 Contact

Visit our website for more information on

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

www.vector.com