MICROSAR Classic BRE

Technical Reference

Version 4.36.0

Authors	vissi, visso, vislsa
Status	Released

Document Information

Reference Documents

No.	Title	Version
[1]	Specification of RTE	R20-11
[2]	Specification of Operating System	R20-11
[3]	Specification of ECU State Manager	R20-11
[4]	Specification of Standard Types	R20-11
[5]	Specification of Platform Types	R20-11
[6]	Specification of Compiler Abstraction	R20-11
[7]	Specification of Memory Mapping	R20-11
[8]	Glossary	R20-11
[9]	Specification of BSW Module Description Template	R20-11
[10]	Specification of Default Error Tracer	R20-11
[11]	Vector DaVinci Configurator Pro Online help	

Table 1-1 Reference documents

Scope of the Document

This document describes the MICROSAR Classic BRE. It is assumed that the reader is familiar with AUTOSAR including the AUTOSAR Classic RTE specification and the BSW specification of AUTOSAR Classic OS and ECUM. The description of those components is not part of this documentation. The related documents are listed in Table 1-1.

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

Contents

Illustrations

Tables

1 Introduction

The MICROSAR Classic BRE is a subset of the AUTOSAR RTE called Basic Runtime Environment (BRE). The goal of the BRE is to enable the realization of projects that do not intend to use a full featured RTE. As the RTE is mandatory in the AUTOSAR architecture the BRE provides a basic set of the RTE functionality that simplify the creation of projects that do not utilize an AUTOSAR Software Component (SWC) Architecture.

The BRE is based on the MICROSAR Classic RTE that has been limited to a reduced feature set. As a consequence, many APIs and the implementation itself continue to refer to “RTE”. To understand this document better BRE and RTE shall be taken as synonyms for the same functionality.

This document describes the MICROSAR Classic BRE configuration using DaVinci Configurator Pro, its generation process and the provided API.

Chapter 2 gives an introduction to the MICROSAR Classic BRE. This brief introduction to the supported subset of 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.

The BRE generation process is described in chapter 3. 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.

The RTE API description in chapter 4 covers the API functionality implemented in the MICROSAR Classic BRE subset of the AUTOSAR RTE and also the APIs which need to be implemented by the Integrator.

The description of the RTE configuration in chapter 5 covers the task mapping and the code generation settings in DaVinci Configurator Pro. A more detailed description of the configuration tool can be found in the online help of the DaVinci Configurator Pro [11].

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

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

1.1 Architecture Overview

The following figure shows where the MICROSAR Classic BRE is located in the AUTOSAR architecture. The BRE is a replacement of the RTE with reduced functionality.

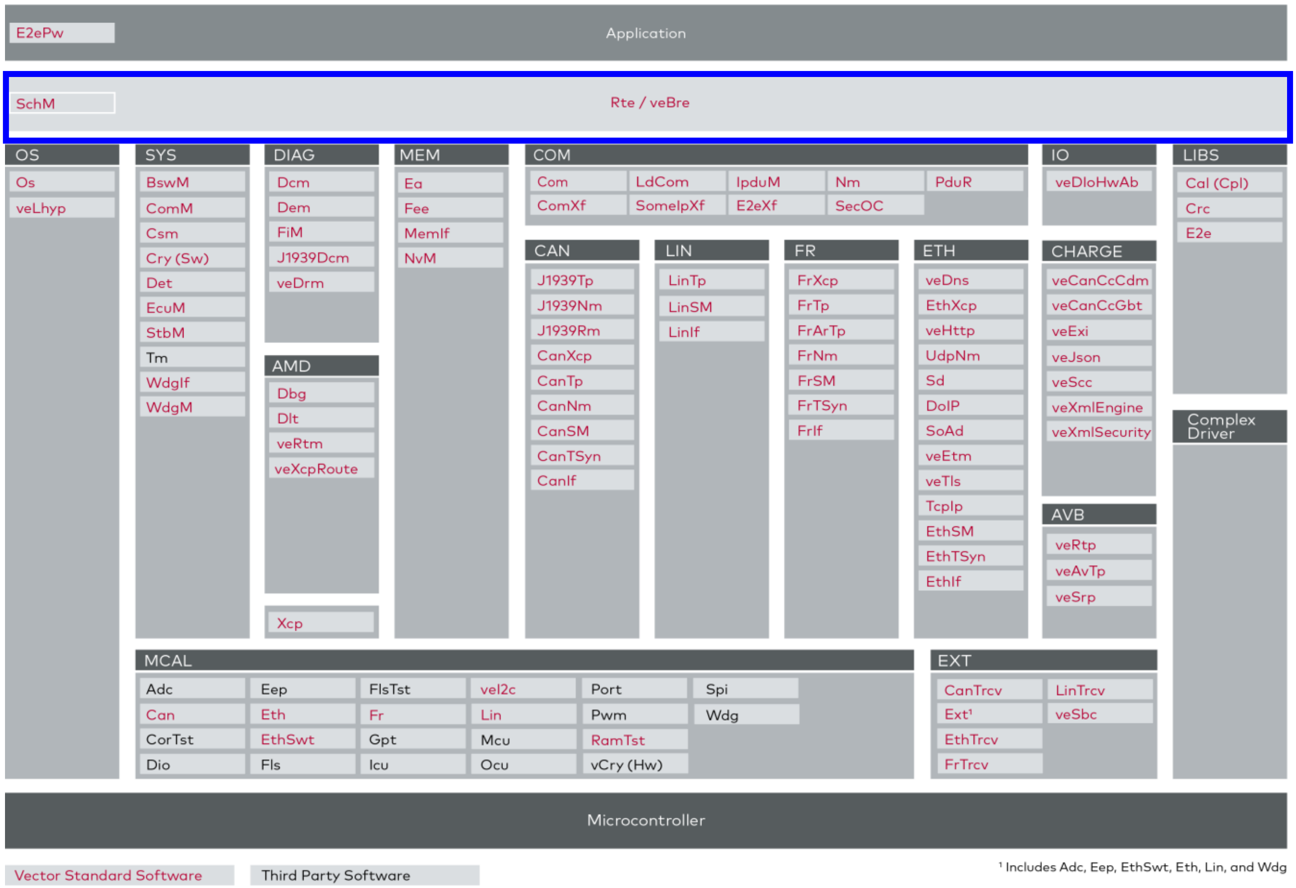

Figure 1-1 AUTOSAR architecture

BRE functionality overview:

- Generation of Application Header files for Service SWCs

SchM functionality overview:

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

2 Functional Description

2.1 Features

The BRE is not defined in AUTOSAR. It implements a subset of the AUTOSAR RTE.

The following features are supported:

Supported Features
Generation of Application Header files for Service SWCs. These are required to compile BSW modules with a service interface.
Execution of cyclic and background triggered schedulable entities (BSW Main-Functions)
Exclusive area handling for BSW modules
OS task body generation
VFB Trace Hooks
Generation on windows and linux (x86_64 glibc 2.26)
Support for Multicore and Memory Protection
BSW-Scheduler Calibration Parameter Access [API: SchM_CData]

Table 2-1 Supported features

The following RTE key features are not supported:

Category	Description
Functional	Application Software Components (SWCs) and all features that relate to this functionality such as Data-Mapping, Memory-Mapping and Service-Mapping. The Task-Mapping of the BRE is reduced to the mapping of BSW scheduleable entities (Main-Functions).
Functional	Realization of RTE APIs (beside SchM functionality). RTE APIs called by the BSW need to be implemented by the integrator and linked with the application.

Table 2-2 Not Supported features

2.2 Initialization

The BRE 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).

The usage of the Api for the initialization of cyclic trigger (SetRelAlarm or SetAbsAlarm) can be selected with the parameter RteGeneration/RteUseAbsAlarmInitialization.
The problem with the use of SetRelAlarm is that between the different calls for different alarm initialization the time advances so that in the end the alarm expire has different OS tick times. With the use of SetAbsAlarm it can be achieved that all runnables are executed in the same OS tick. This ensures that the execution order of runnables is exactly as defined in the task mapping. The usage of SetAbsAlarm requires the configuration of proper offsets.

2.3 Exclusive Areas

An exclusive area (EA) can be used to protect a part of the BSW code regarding mutual exclusive access. AUTOSAR specifies different implementation methods which can be configured during ECU configuration of the BRE. See also Chapter 5.6.

- OS Interrupt Blocking
- All Interrupt Blocking
- OS Resource

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

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 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.

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

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.4 Error Handling

2.4.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [10], if development error reporting is enabled in the `RteGeneration` parameters (i.e. by setting the parameters `DevErrorDetect` and / or `DevErrorDetectUninit`).

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

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

Service ID	Service
0x00	SchM_Init
0x01	SchM_Deinit
0x03	SchM_Enter
0x04	SchM_Exit
0x15	Rte_Switch
0x18	Rte_SwitchAck
0x1C	Rte_Call
0x2C	Rte_Mode
0x70	Rte_Start
0x71	Rte_Stop
0xF0	Rte_Task

Table 2-3 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_TRANSITIONSTATE	Mode machine is in an invalid state

Table 2-4 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`.

3 BRE Generation and Integration

3.1 Embedded Implementation

This chapter gives necessary information for the integration of the MICROSAR Classic BRE into an application environment of an ECU. Embedded Implementation

The delivery of the BRE consists out of these files:

File	Description	Integration Tasks
Rte_<CT>.h	Generated file that contains the APIs for one SWC. It needs to be included into the SWC code. This header file is the only file to be included in the component code. It is generated to the `Components` subdirectory by default.	-
Rte_<CT>_Type.h	Generated file that contains SWC specific type definitions. It is generated to the `Components` subdirectory by default.	-
SchM_<BSWM>.h	Generated file that contains the APIs for one BSW module. It needs to be included into the BSW module code.	-
SchM_<BSWM>_Type.h	Generated file that contains BSW module specific type definitions.	-
<CT>_MemMap.h	Generated file with template areas that can be adapted by the user. Template contains SWC specific part of the memory mapping. It is generated to the `Components` subdirectory by default.	Adapt the dedicated code areas within that file. See hints within that file.
Rte.c	Generated file that contains the main implementation of the BRE.	-
Rte_<OsApplication>.c	Generated file that contains OsApplication specific parts of the generated BRE (only generated when OsApplications are configured).	-
Rte.h	Generated file that contains BRE internal declarations.	-
Rte_Main.h	Generated file that contains the lifecycle API.	-
Rte_Cfg.h	Generated file that contains the configuration for the BRE.	-
Rte_Cbk.h	Generated file that contains prototypes for COM callbacks.	-
Rte_Hook.h	Generated file that contains relevant information for VFB tracing.	-
Rte_Type.h	Generated file that contains the application defined data type definitions and BRE internal data types.	-

Rte_DataHandleType.h	Generated file that contains the data handle type declarations required for the component data structures.	-
Rte_UserTypes.h	Generated file with template areas that can be adapted by the user. It is generated if either user defined data types are required for Per-Instance memory or if a data type is used by the BRE but generation is skipped with the `typeEmitter` attribute.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_MemMap.h	Generated file with template areas that can be adapted by the user. It contains BRE specific part of the memory mapping.	Adapt the dedicated code areas within that file. See hints within that file.
Rte_Compiler_Cfg.h	Generated file with template areas that can be adapted by the user. It contains BRE specific part of the compiler abstraction	Adapt the dedicated code areas within that file. See hints within that file.
Rte.oil	Generated file that contains the OS configuration for the BRE.	-
Rte_Needs.ecuc.arxml	Generated file that contains the BRE requirements on BSW module configuration for Os, Com, LdCom, Xcp and NVM.	-
Rte.html	Generated file that contains information about RAM / CONST consumption of the generated BRE as well as a listing of all triggers and their OS events and alarms.	-

Table 3-1 Generated Files of RTE Generation Phase

3.2 BRE Integration

The AUTOSAR RTE implements typically a huge variety of APIs that are called by the BSW such as the COM, ECUM, BSWM and COMM. Depending on the RTE configuration these API calls are then forwarded to the application software components. Thereby the RTE also implements internal state machines that allow a more abstract access to BSW modes.

Using the BRE there is no application SWC design available, so the application has to implement the RTE APIs called by the BSW directly. As the number and names of these APIs highly depend on the BSW configuration this documentation cannot provide a complete list of the APIs that need to be implemented during the BRE integration.

Note
The BRE generates empty function bodies of APIs, which are called by the BRE and need to be implemented by the application.

These function bodies can be found in Rte*.c and are excluded from the compilation by #ifdef BRE_ENABLE_UNCONNECTED_RTE_APIS.

To simplify the integration, these function bodies can be copied to a separate file and used as a starting point for the implementation of the application.

The Technical References of the BSW modules as well as the AUTOSAR standard define the semantics and APIs that will have to be implemented while integrating the BRE. Additionally, chapter 4 provides a high-level overview on the APIs classes that need to be implemented (if required by the BSW configuration).

We have marked these APIs with the following mark:

Edit
This API is not generated by the MICROSAR Classic BRE and must be implemented by the integrator.

In order to integrate the BRE we recommend the following steps:

1. Configure and enable your BSW modules
2. Activate the BRE (enable the BSW Module “Rte”)
3. Configure the BRE
   - Task Mapping of BSW Main-Functions
   - Exclusive Area handling
4. Generate the BRE (using the RTE generator) along with your BSW modules
5. Compile and link your project
6. Implement application calls to the BSW in your application
7. Fix linker errors by implementing the RTE APIs required by the BSW. The Rte*.c files contain deactivated placeholder APIs that can be used as base for the implementation.

Reference

See chapter 4 for details on the APIs classes your application may have to implement.

Carefully read the API descriptions of the BSW modules that require the implementation of RTE APIs. This information can be found in the Technical References of the MICROSAR Classic BSW modules or in the AUTOSAR documents defined in Table 1-1.

See chapter 5 for details on the BRE configuration process.

3.3 Include Structure

3.3.1 BRE Include Structure

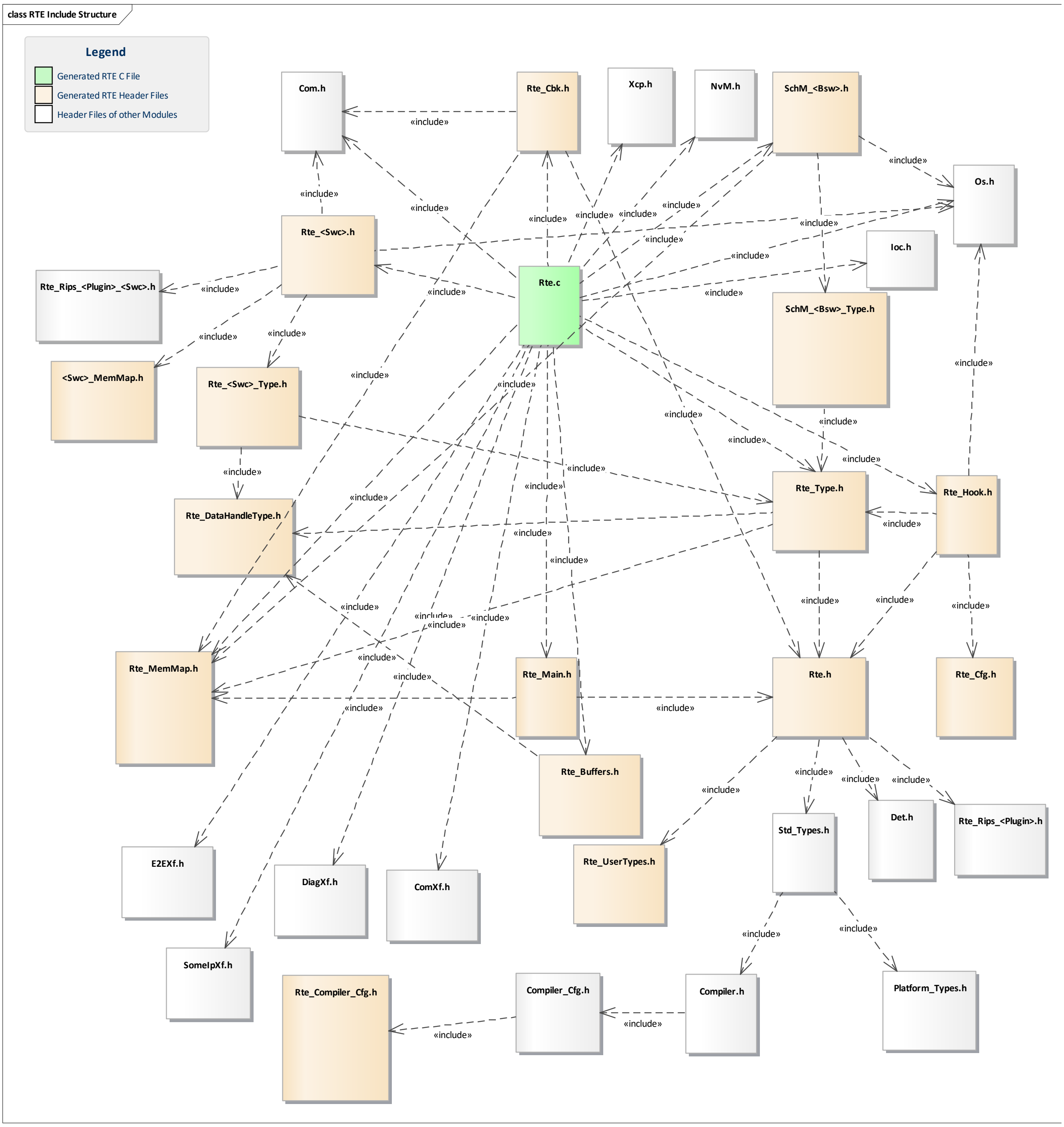

Figure 3-1 BRE Include Structure

3.3.2 BSW Include Structure

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

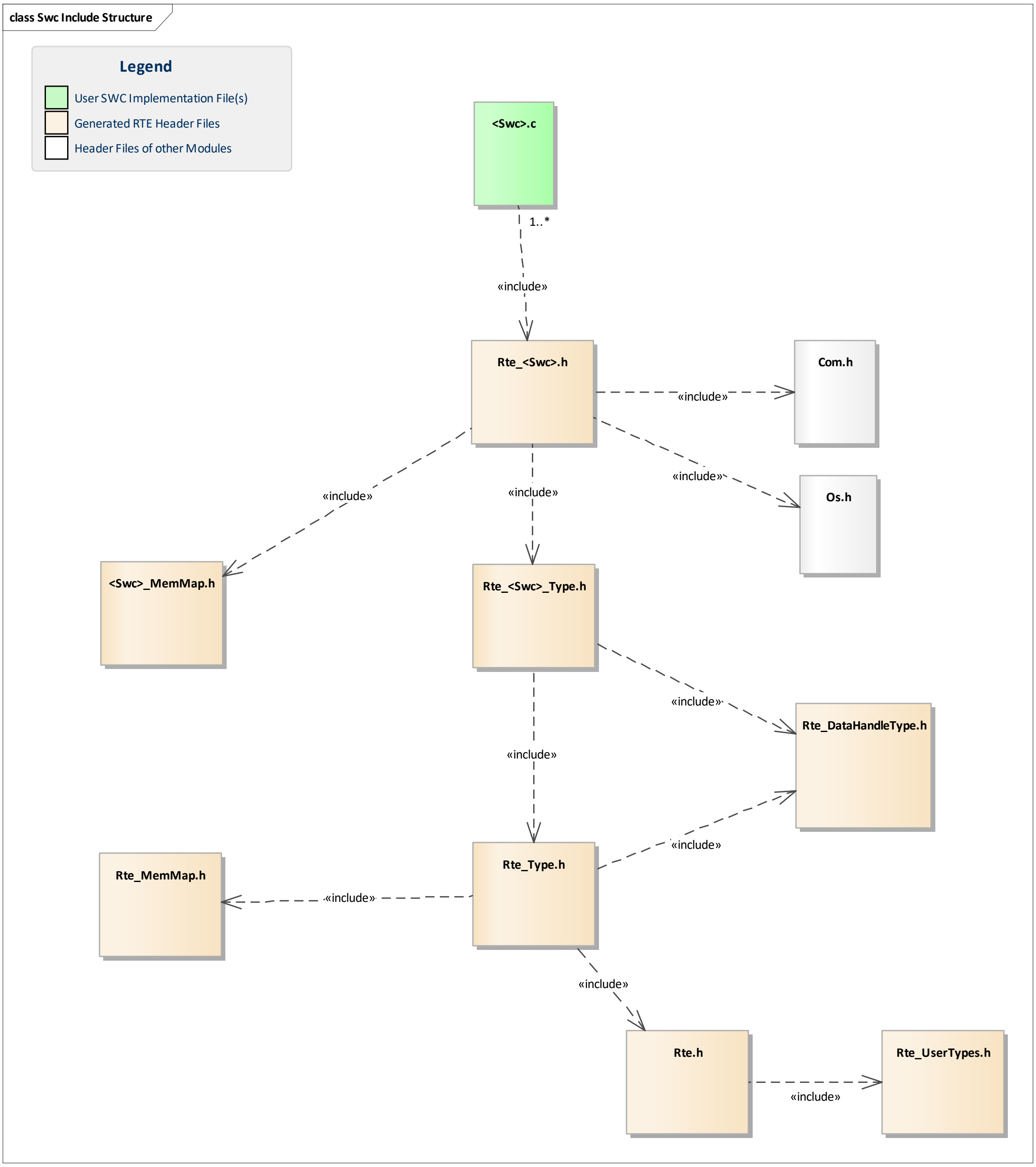

Figure 3-2 BSW Include Structure

3.4 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_CONST	<Swc>_CONST	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	■															
<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			■													
<Swc>_START_SEC_VAR_INIT_UNSPECIFIED <Swc>_STOP_SEC_VAR_INIT_UNSPECIFIED				■												
RTE_START_SEC_VAR_NOINIT_UNSPECIFIED RTE_STOP_SEC_VAR_NOINIT_UNSPECIFIED					■											
<Swc>_START_SEC_VAR_NOINIT_UNSPECIFIED <Swc>_STOP_SEC_VAR_NOINIT_UNSPECIFIED						■										
RTE_START_SEC_CONST_UNSPECIFIED RTE_STOP_SEC_CONST_UNSPECIFIED							■									
<Swc>_START_SEC_CONST_UNSPECIFIED <Swc>_STOP_SEC_CONST_UNSPECIFIED								■								
RTE_START_SEC_CODE RTE_STOP_SEC_CODE									■							
<Swc>_START_SEC_CODE <Swc>_STOP_SEC_CODE										■						
RTE_START_SEC_APPL_CODE RTE_STOP_SEC_APPL_CODE											■					
RTE_START_SEC_<OSAPPL>_CODE RTE_STOP_SEC_<OSAPPL>_CODE									■							
RTE_START_SEC_<TASKNAME>_CODE RTE_STOP_SEC_< TASKNAME >_CODE									■							

Table 3-2 Compiler abstraction and memory mapping

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

- Rte_Compiler_Cfg.h

- Rte_MemMap.h

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

3.4.1 Memory Sections for Software Components

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

The variable part `<Swc>` is the camel case software component type name.

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

- <Swc>_MemMap.h

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

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

4 API Description

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

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

Abbreviation	Description
<cts>	ComponentTypeSymbol
<r>	RunnableEntity
<ap>	AccessPoint
<i>	PortInterface
<p>	PortPrototype
<o>	OperationPrototype
<d>	DataElementPrototype
<cp>	CalibrationParameter
<m>	ModeGroupPrototype
<v>	InterRunnableVariable
<n>	PerInstanceMemory

Table 4-1 Abbreviations for API Name Placeholders

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

4.1 API Error Status

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

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

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

The Rte_ApplicationError macro returns the application errors without overlayed errors.

4.2 BSW Exclusive Areas

4.2.1 SchM_Enter

Prototype	
void **SchM_Enter_<Bsw>_<ExclusiveArea>** (void)	
Parameter	
–	
Return code	
–	
Existence	
This API exists when at least one schedulable entity has configured access (canEnterExclusiveArea) to an exclusive area in the internal behavior of the BSW module description.	
Functional Description	
The function SchM_Enter_<bsw>_<ea>() implements access to the exclusive area. The exclusive area is defined in the context of a BSW module and may be accessed by all schedulable entities of that module via this API. This function is the counterpart of SchM_Exit_<bsw>_<ea>(). Each call to SchM_Enter_<bsw>_<ea>() must be matched by a call to SchM_Exit_<bsw>_<ea>() in the same schedulable entity. One exclusive area must not be entered more than once at a time, but different exclusive areas may be nested, as long as they are left in reverse order of entering them. For restrictions on using exclusive areas with different implementation methods, see section 2.3.	
Call Context	
This function can be used inside a schedulable entity in Task or Interrupt context.	

4.2.2 SchM_Exit

Prototype	
void **SchM_Exit_<Bsw>_<ExclusiveArea>** (void)	
Parameter	
–	
Return code	
–	
Existence	
This API exists when at least one schedulable entity has configured access (`canEnterExclusiveArea`) to an exclusive area in the internal behavior of the BSW module description.	
Functional Description	
The function `SchM_Exit_<bsw>_<ea>()` implements releasing of the exclusive area. The exclusive area is defined in the context of a BSW module and may be accessed by all schedulable entities of that module via this API. This function is the counterpart of `SchM_Enter_<bsw>_<ea>()`. Each call to `SchM_Enter_<bsw>_<ea>()` must be matched by a call to `SchM_Exit_<bsw>_<ea>()` in the same schedulable entity. One exclusive area must not be entered more than once at a time, but different exclusive areas may be nested, as long as they are left in reverse order of entering them. For restrictions on using exclusive areas with different implementation methods, see section 2.3.	
Call Context	
This function can be used inside a schedulable entity in Task or Interrupt context.	

4.3 Mode Management

4.3.1 Rte_Switch

Prototype	
`Std_ReturnType` **`Rte_Switch_<p>_<m>`** `( IN Rte_ModeType_<ModeDeclarationGroup> mode )`	
Parameter	
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).	

Edit
This API is not generated by the MICROSAR Classic BRE and must be implemented by the integrator.

4.3.2 Rte_Mode

Prototype	
`Rte_ModeType_<ModeDeclarationGroup>` **`Rte_Mode_<p>_<m>`** `( void )`	
Parameter	
-	
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).	

Edit
This API is not generated by the MICROSAR Classic BRE and must be implemented by the integrator.

4.3.3 Enhanced Rte_Mode

Prototype	
`Rte_ModeType_<ModeDeclarationGroup> Rte_Mode_<p>_<m> ( OUT Rte_ModeType_<ModeDeclarationGroup> previousMode, OUT Rte_ModeType_<ModeDeclarationGroup> nextMode )`	
Parameter	
`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 provides 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).	

Edit
This API is not generated by the MICROSAR Classic BRE and must be implemented by the integrator.

4.3.4 Rte_SwitchAck

Prototype	
`Std_ReturnType` **`Rte_SwitchAck_<p>_<m>`** `( void )`	
Parameter	
-	
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 a `Rte_Switch` call. Depending on the configuration, the API can be either blocking or non-blocking.	
Call Context	
This function can be used inside a runnable entity of an AUTOSAR software component (SWC).	

Edit
This API is not generated by the MICROSAR Classic BRE and must be implemented by the integrator.

4.4 Client-Server Communication

4.4.1 Rte_Call

Prototype	
Std_ReturnType **Rte_Call_<p>_<o>** ({IN type [*]inputparam,}* {OUT type *outputparam,}* {INOUT type *inoutputparam,}*)	
Parameter	
[*]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_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.
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).	

Edit
This API is not generated by the MICROSAR Classic BRE and must be implemented by the integrator.

4.5 Calibration Parameters

4.5.1 SchM_CData

Prototype	
`<DataType> SchM_CData_<Bsw>_<cp> ( void )`	
`<DataType> *SchM_CData_<Bsw>_<cp> ( void )`	
Parameter	
-	
Return code	
<DataType>	For primitive data types the return value contains the content of the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
<DataType> *	For composite data types and string types the return value contains the reference to the calibration parameter. The return value is of type <DataType>, which is the type of the calibration element prototype.
Existence	
This API exists for each per instance parameter specified in the internal behavior of the BSW module description.	
Functional Description	
The function `SchM_CData_<Bsw>_<cp>()` can be used to access BSW local calibration parameters.	
Call Context	
This function can be used inside a schedulable entity in Task or Interrupt context.	

4.6 RTE Lifecycle API

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

4.6.1 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.	

4.6.2 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.	
Call Context	
This function has to be called by the ECU state manager.	

4.6.3 Rte_InitMemory

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

4.7 SchM Lifecycle API

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

4.7.1 SchM_Init

Prototype	
void **SchM_Init** (void)	
Parameter	
-	
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).	

4.7.2 SchM_Start

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

4.7.3 SchM_StartTiming

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

4.7.4 SchM_Deinit

Prototype	
void **SchM_Deinit** (void)	
Parameter	
-	
Return code	
-	
Functional Description	
This function finalizes the BSW Scheduler and stops the timer which triggers the main functions.	
Call Context	
This function has to be called by the ECU state manager from task context.	

4.7.5 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.	

Caution

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

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

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

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

4.8 VFB Trace Hooks

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

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

Prototype	
`void Rte_[<client>_]<API>Hook_<cts>_<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: SwitchAck, Switch, Call The <AccessPoint> is defined as follows: ▶ Switch, SwitchAck: <PortPrototype>_<ModeDeclarationGroupPrototype> ▶ Call: <PortPrototype>_<OperationPrototype>	
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 hook is also the runnable entity of an AUTOSAR software component (SWC).	

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

Prototype	
void **Rte_[<client>_]<API>Hook_<cts>_<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: Switch, SwitchAck, Call The <AccessPoint> is defined as follows: ▶ Switch, SwitchAck: <PortPrototype>_<ModeDeclarationGroupPrototype> ▶ Call: <PortPrototype>_<OperationPrototype>	
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 hook is also the runnable entity of an AUTOSAR software component (SWC).	

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

- RTE Life-Cycle APIs: Rte_Start, Rte_Stop

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

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

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

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

4.8.5 Rte_[<client>_]Task_Activate

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

4.8.6 Rte_[<client>_]Task_Terminate

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

4.8.7 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.	

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

4.8.9 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 the RTE API functions and in the generated task bodies.	

4.8.10 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 the RTE API functions and in the generated task bodies.	

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

Prototype	
void **Rte_[<client>_]Runnable_<cts>_<re>_Start** (void)	
Parameter	
-	
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 the RTE generated task bodies.	

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

Prototype	
void **Rte_[<client>_]Runnable_<cts>_<re>_Return** (void)	
Parameter	
-	
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 the RTE generated task bodies.	

4.9 RTE Interfaces to BSW

The RTE has standardized Interfaces to the following basic software modules

- DET
- OS

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

4.9.1 Interface to OS

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

Used OS API
SetRelAlarm
SetAbsAlarm
CancelAlarm
SetEvent
GetEvent
ClearEvent
WaitEvent
GetTaskID
ActivateTask
Schedule
TerminateTask
ChainTask
GetResource
ReleaseResource
DisableAllInterrupts
EnableAllInterrupts
SuspendAllInterrupts
ResumeAllInterrupts
SuspendOSInterrupts
ResumeOSInterrupts

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.

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

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

4.9.2 Interface to DET

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

See chapter 2.4.1 for details.

Used DET API
Det_ReportError

5 BRE Configuration

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

- assignment of runnables / scheduleable entities to OS tasks
- selection of the exclusive area implementation method
- configuration of the periodic triggers
- selection of the optimization mode
- selection of required VFB tracing callback functions
- platform dependent resource calculation

5.1 Module Activation

To enable the BRE activate the BSW Module “Rte” in the Project Settings Editor of DaVinci Configurator Pro and set the parameter /MICROSAR/Rte/RteGeneration/RteBasicRuntimeEnvironmentMode to true.

5.1.1 Configuration Variants

The BRE supports the configuration variants

- `VARIANT-PRE-COMPILE`

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

5.2 Task Configuration

The BRE supports scheduling of BSW Main-Functions. Therefore, the BSW Main-Functions (RTE term: “Runnables”) have to be mapped to a task.

The task configuration editor of DaVinci Configurator Pro also includes attributes which are part of the OS configuration. The parameters are required to control the RTE generation.

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

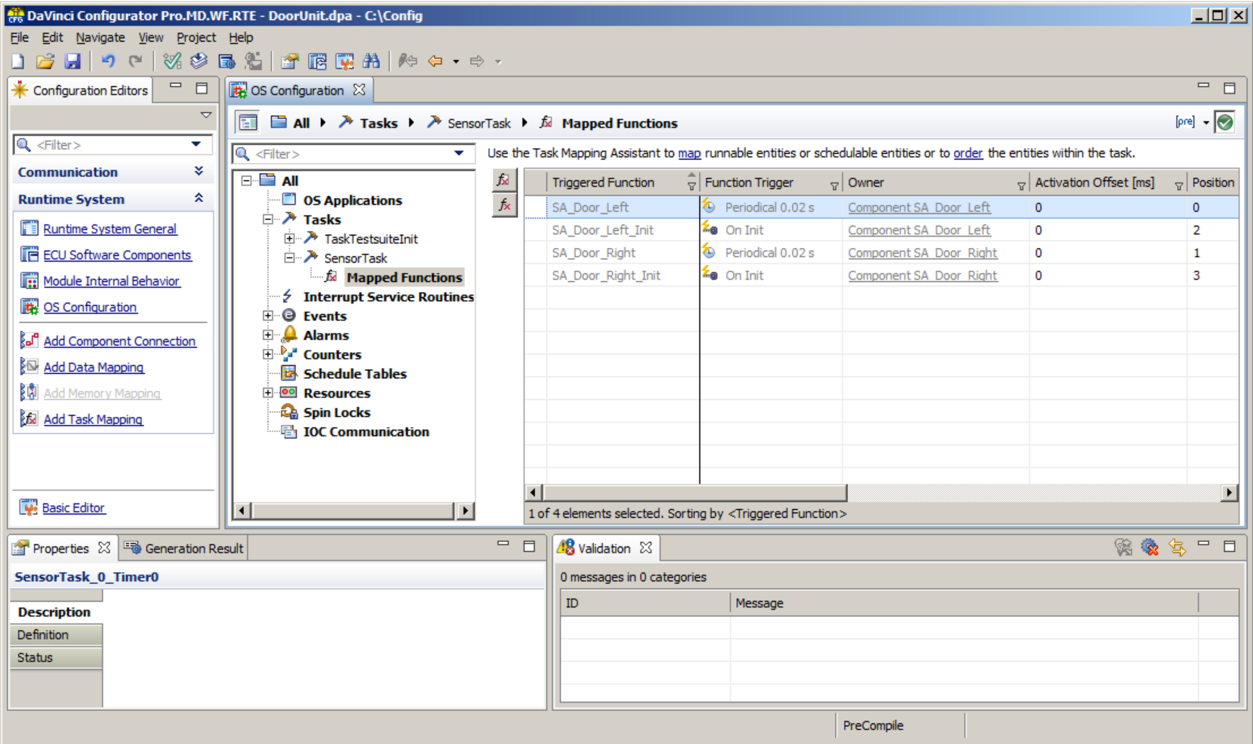

Figure 5-1 Mapping of Runnables to Tasks

The MICROSAR Classic BRE 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.

While extended tasks are used for tasks that need to wait for different RTE trigger conditions, basic tasks are used when all runnables of a 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.

In addition to the Task Type the number of possible task activations can be configured in the same dialog.

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

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

5.3 BRE Generator Settings

The following figure shows how the MICROSAR Classic BRE is enabled for code generation in DaVinci Configurator Pro.

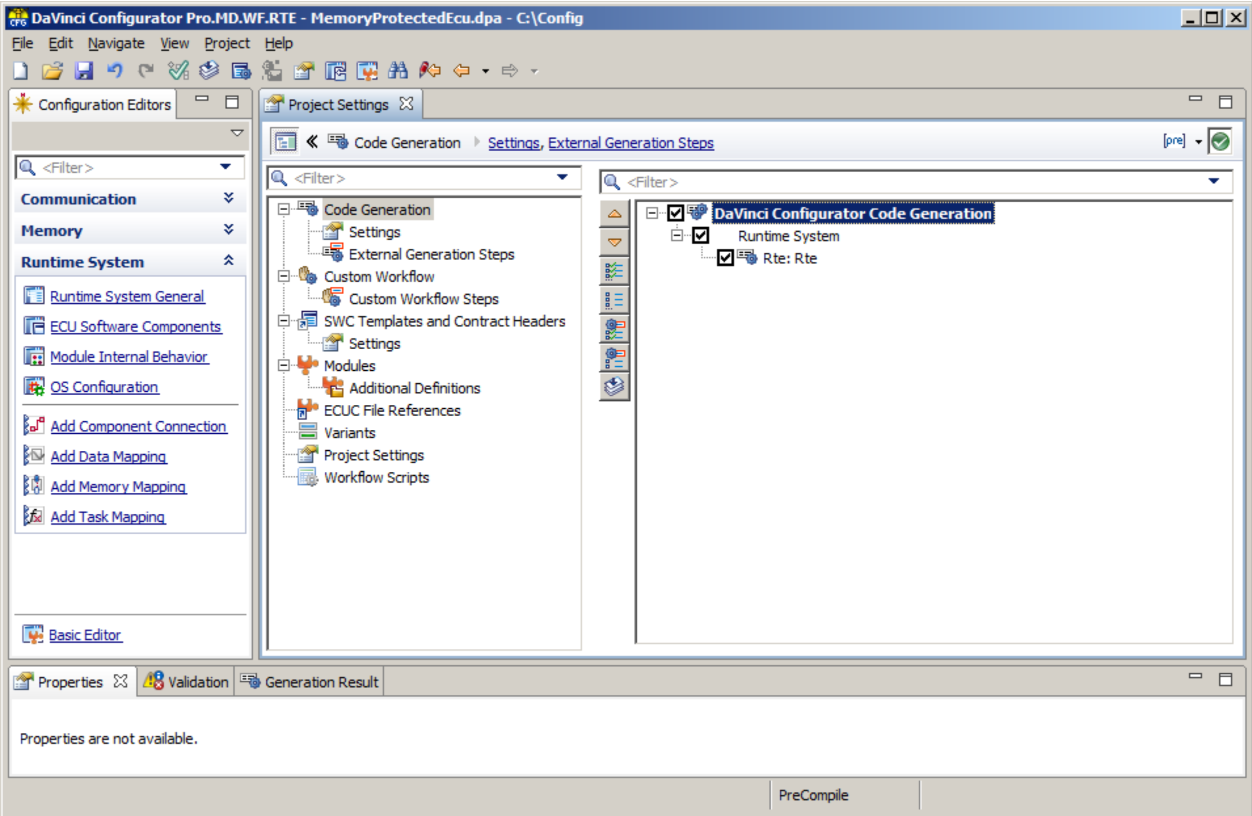

Figure 5-2 BRE Generator Settings

5.4 Optimization Mode Configuration

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

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

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

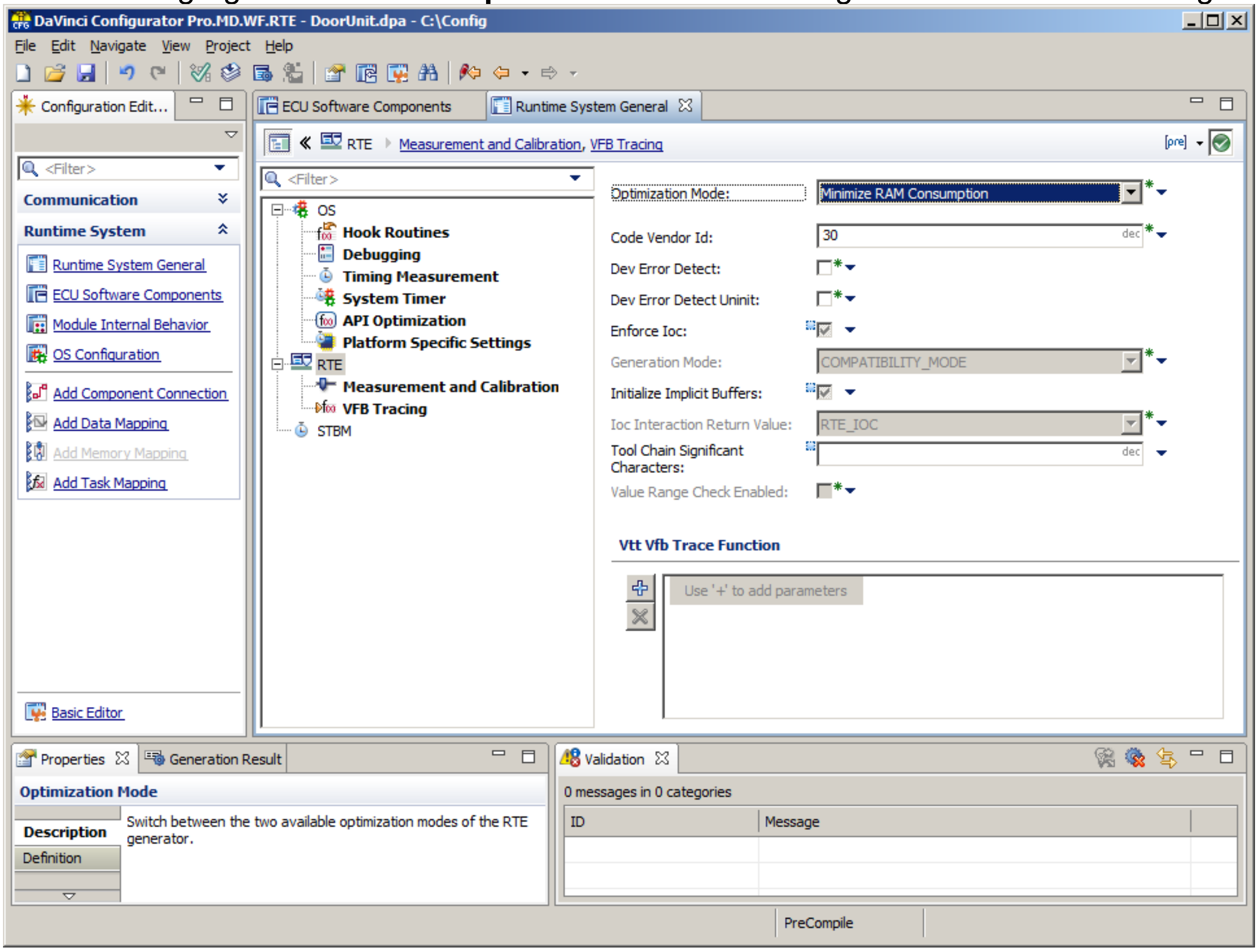

Figure 5-3 Optimization Mode Configuration

5.5 VFB Tracing Configuration

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

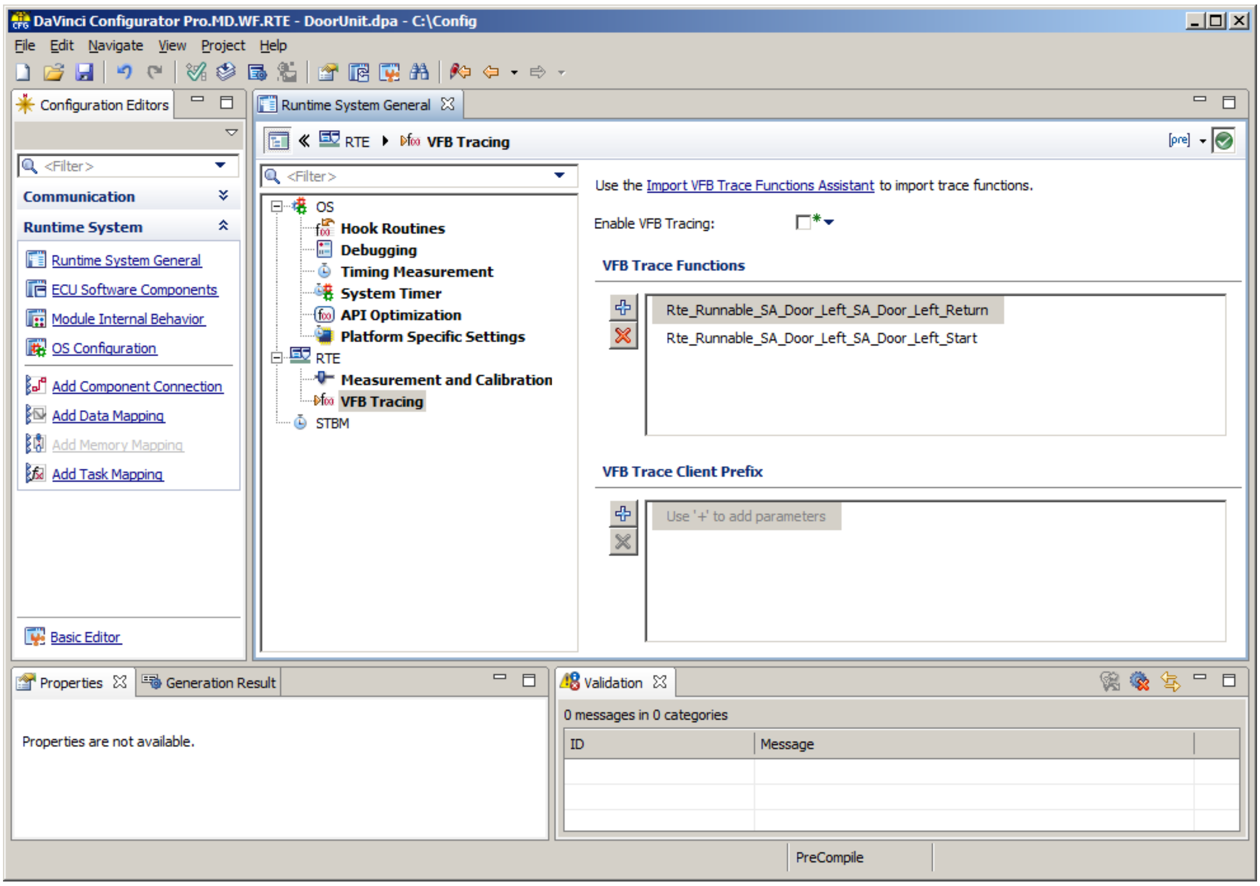

Figure 5-4 VFB Tracing Configuration

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.

5.6 Exclusive Area Implementation

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

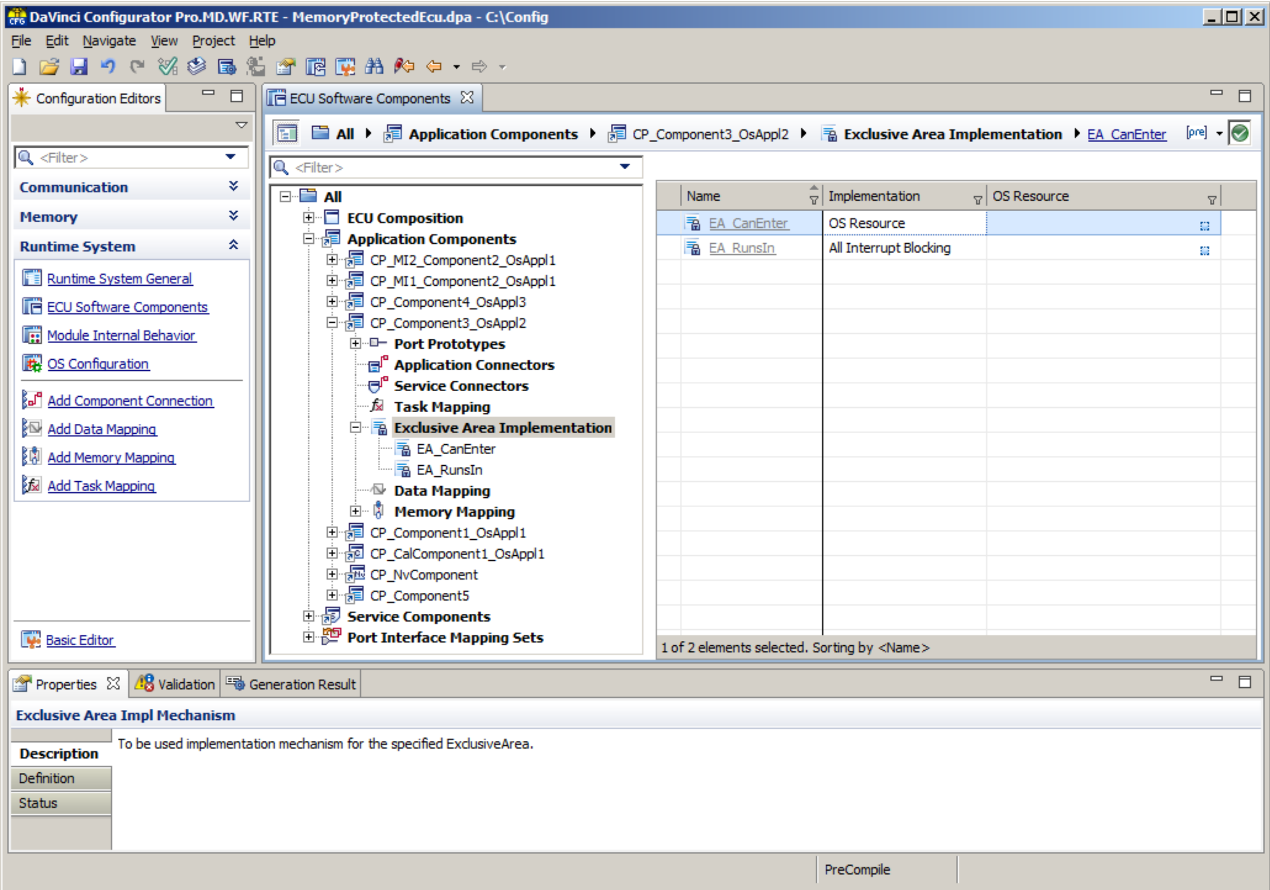

Figure 5-5 Exclusive Area Implementation Configuration

5.7 Periodic Trigger Implementation

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

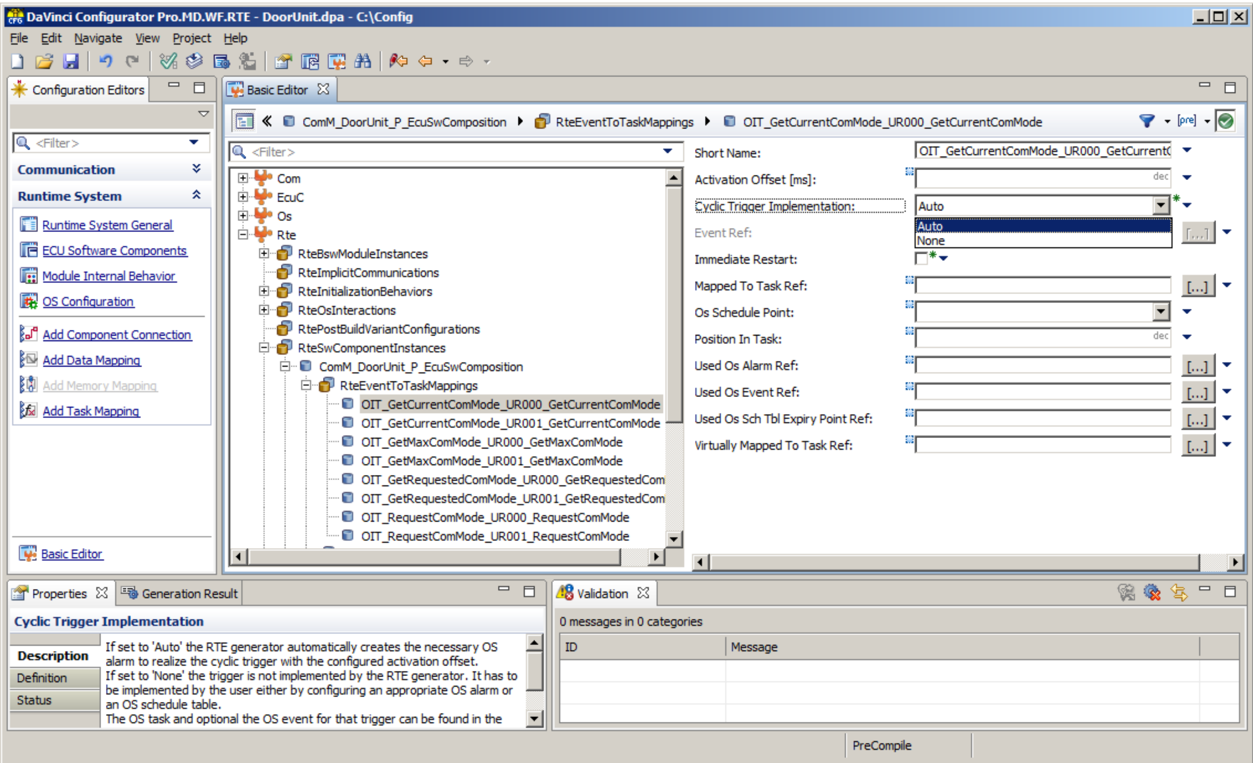

Figure 5-6 Periodic Trigger Implementation Configuration

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 5-7 HTML Report

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

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

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

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

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

5.8 Resource Calculation

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

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

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

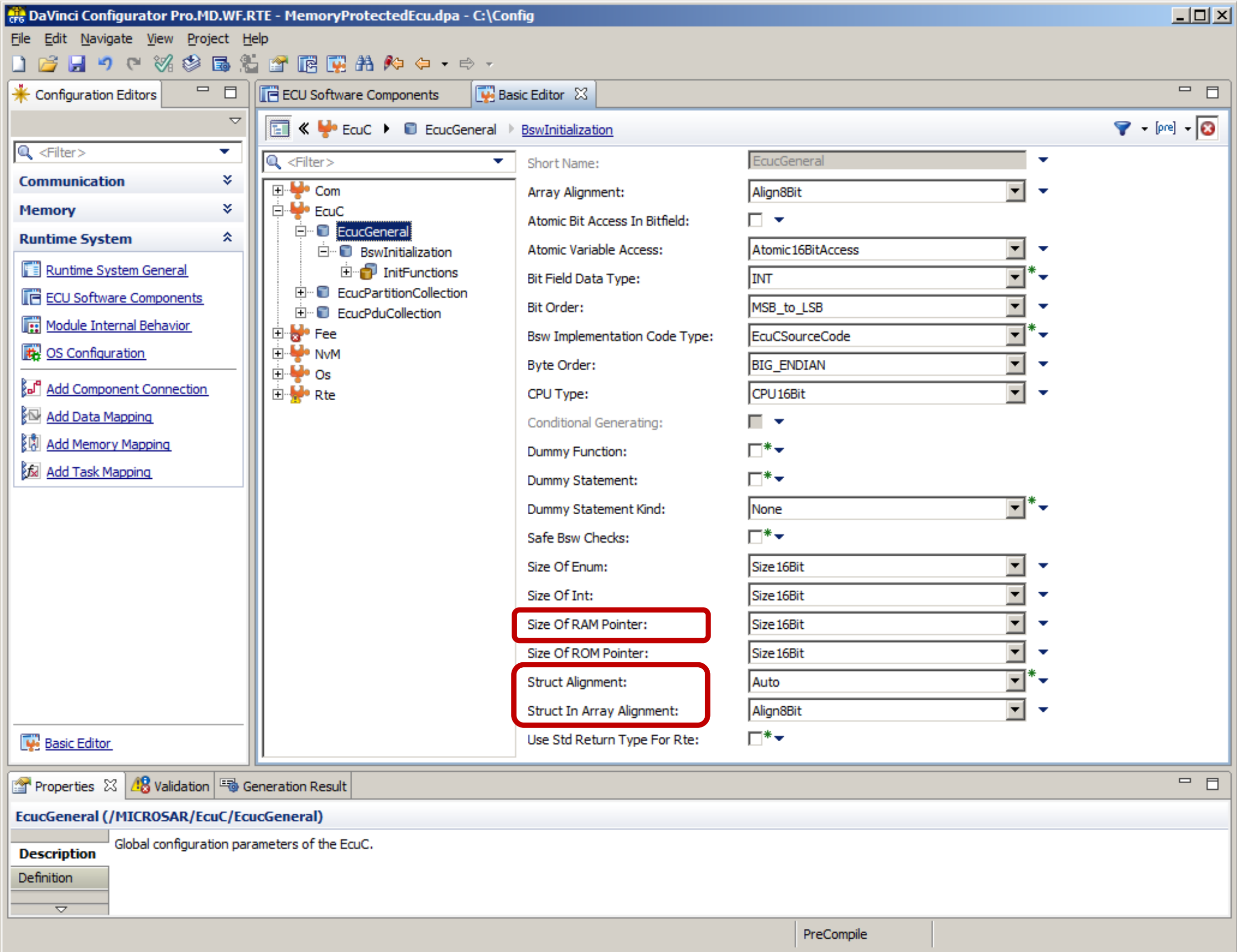

Figure 5-8 Configuration of platform settings

6 Glossary and Abbreviations

6.1 Glossary

Term	Description
DaVinci Configurator Pro	The BSW and BRE configuration and generation tool for your MICROSAR Classic BSW.

Table 6-1 Glossary

The AUTOSAR Glossary [8] also describes a lot of important terms, which are used in this document.

6.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BRE	Basic Runtime Environment
BSW	Basis Software
BSWM	Basis Software Module
Com	Communication Layer
CT	Component Type
C/S	Client-Server
EA	Exclusive Area
ECU	Electronic Control Unit
EcuM	ECU State Manager
FOSS	Free and Open Source Software
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (Vector's AUTOSAR solution)
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
SWC	Software Component
SWS	Software Specification

Table 6-2 Abbreviations

7 Additional Copyrights

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

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

Table 7-1 Free and Open Source Software Licenses

8 Contact

Visit our website for more information on

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

www.vector.com