ETAS

RTA-RTE V6.8.0
Reference Manual

DRIVING | EMBEDDED EXCELLENCE

Copyright

The data in this document may not be altered or amended without special notification from ETAS GmbH. ETAS GmbH undertakes no further obligation in relation to this document. The software described in it can only be used if the customer is in possession of a general license agreement or single license. Using and copying is only allowed in concurrence with the specifications stipulated in the contract. Under no circumstances may any part of this document be copied, reproduced, transmitted, stored in a retrieval system or translated into another language without the express written permission of ETAS GmbH.

©Copyright 2019 ETAS GmbH, Stuttgart.

The names and designations used in this document are trademarks or brands belonging to the respective owners.

Document: 10756-RM-001 EN - 05-2019

Revision: 92501 [RTA-RTE 6.8.0]

This product described in this document includes software developed by the Apache Software Foundation (`http://www.apache.org/`).

Contents

1 About this Manual 8
1.1 Who Should Read this Manual? 8
1.2 Document Conventions 9
1.3 Acronyms and Abbreviations 9

2 Invocation 11
2.1 Command-line Usage 11
2.2 Output Files 11
2.3 OS Configuration File 14
2.4 COM OIL File 14
2.5 RTE Configuration Constants 15
2.6 Screen Output 17
2.7 Error and Information Messages 18
2.8 Exit Codes 18
2.9 RTE Library 18
2.10 User Configuration File 19

3 Command-line options 20
3.1 Examples 20
3.2 Interaction with ECUC configuration 20
3.3 -- 22
3.4 --append-name-to-buffer 23
3.5 --atomic-assign 24
3.6 --bit-pack-type 25
3.7 --bsw 26
3.8 --bsw-scope-limit-defns 27
3.9 --calibration-disable 28
3.10 --calibration-instantiation 29
3.11 --calibration-method 30
3.12 --client-server-global-optimization 31
3.13 --com-symbolic-sigs 32
3.14 --com-version 33
3.15 --contract 34
3.16 --deviate-allow-unmapped-swci-config 35
3.17 --deviate-appl-impl-compu-method 36
3.18 --deviate-appl-impl-display-format 37
3.19 --deviate-bsw-any-partition 38
3.20 --deviate-allow-supportsmulti-sharedmemorys 39
3.21 --deviate-enum-cast 40
3.22 --deviate-group-calibration-none 41
3.23 --deviate-ignore-datatype-semantics 42
3.24 --deviate-implicit-cat2-mdd 43
3.25 --deviate-implicit-modify-for-loopbacks 44
3.26 --deviate-memmap-decls 45
3.27 --deviate-omit-implicit-cds 46
3.28 --deviate-physical-dimension-compatibility 47
3.29 --deviate-prefer-no-empty-executions 48

3.30 --deviate-split-swci-support 49
3.31 --deviate-trace-implicit-api 51
3.32 --deviate-unconnected-pmode-behavior 52
3.33 --disable-warning 53
3.34 --error-as-warning 54
3.35 --error-report 55
3.36 --exclusive-area-optimization 56
3.37 --fast-init 57
3.38 --file 58
3.39 --force-basic-tasks 59
3.40 --have-64bit-int-types 60
3.41 --help 61
3.42 --implicit-allocation-method 62
3.43 --implicit-read-return-const 63
3.44 --implicit-use-global-buffers 64
3.45 --incremental-build 65
3.46 --initial-value-rounding 66
3.47 --ioc-header 67
3.48 --ioc-xml-namespace 68
3.49 --local-mcsd 69
3.50 --makedep 70
3.51 --mcore-spinlocks-always 71
3.52 --mcsd-policy 72
3.53 --measurement 73
3.54 --memory-sections 74
3.55 --notimestamps 75
3.56 --operating-system 76
3.57 --optimize 77
3.58 --os-define-osenv 78
3.59 --os-fp 79
3.60 --os-header 80
3.61 --os-output-param 81
3.62 --os-permit-extended-tasks 82
3.63 --os-task-as-function 83
3.64 --os-xml-namespace 84
3.65 --output 85
3.66 --period 86
3.67 --preferred-intra-core-protection-scheme 87
3.68 --protection-threshold-copy-bytes 88
3.69 --quiet 89
3.70 --report 90
3.71 --rte 91
3.72 --samples 92
3.73 --strict-config-check 93
3.74 --strict-initial-values-check 94
3.75 --strict-unconnected-rport-check 95
3.76 --sws 96
3.77 --task-recurrence 97

3.78 --template-path 98
3.79 --terminate-background-tasks 99
3.80 --test-license 100
3.81 --text-value-spec-policy 101
3.82 --toolchain-significant-len 102
3.83 --use-partition-sections 103
3.84 --variability-also-bind 104
3.85 --version 105
3.86 --vfb-trace 106
3.87 --warn-directive 107
3.88 --warning-as-error 108
3.89 --xfrm-ignore-inplace 109

4 Configuration **110**
4.1 Supported namespace and schema versions 110
4.2 References 111
4.3 Packages 114
4.4 Software Components 115
4.5 AUTOSAR Types and Data Conversion 128
4.6 Interfaces 142
4.7 Measurement 146
4.8 NVRAM 150
4.9 AUTOSAR Modes 155
4.10 Internal Behavior 155
4.11 Implementation 174
4.12 Signals 175
4.13 System Signal Group 176
4.14 PDU Type 176
4.15 ECU Types 177
4.16 Composition 177
4.17 ECU Instances 179
4.18 System Description 181
4.19 ECU Description 190
4.20 Vendor Specific XML Extensions 198
4.21 Post-build 198
4.22 Variability 199
4.23 Support for the atpSplitable Stereotype 207

5 RTE Conventions **209**
5.1 Name Space 209
5.2 Software-Component Naming 209

6 RTE API Reference **210**
6.1 API Parameter Passing 210
6.2 Data Types 210
6.3 Rte_Call 212
6.4 Rte_Prm 213
6.5 Rte_CData 214
6.6 Rte_Enter 215

6.7 Rte_Exit . . . 216
6.8 Rte_IFeedback . . . 217
6.9 Rte_Feedback / Rte_SwitchAck . . . 218
6.10 Rte_IInvalidate . . . 219
6.11 Rte_Invalidate . . . 220
6.12 Rte_IRead . . . 221
6.13 Rte_IWrite . . . 222
6.14 Rte_IWriteRef . . . 222
6.15 Rte_IrvIRead . . . 223
6.16 Rte_IrvIWrite . . . 224
6.17 Rte_IrvRead . . . 225
6.18 Rte_IrvWrite . . . 226
6.19 Rte_IStatus . . . 227
6.20 Rte_IsUpdated . . . 227
6.21 Rte_MainFunction . . . 228
6.22 Rte_Mode . . . 229
6.23 Rte_Ports . . . 230
6.24 Rte_NPorts . . . 230
6.25 Rte_Port . . . 231
6.26 Rte_Pim . . . 232
6.27 Rte_Read . . . 233
6.28 Rte_DRead . . . 234
6.29 Rte_Receive . . . 235
6.30 Rte_Result . . . 237
6.31 Rte_Send . . . 238
6.32 Rte_Start . . . 239
6.33 Rte_Stop . . . 239
6.34 Rte_Switch . . . 240
6.35 Rte_Tick_Timeouts . . . 241
6.36 Rte_Trigger . . . 242
6.37 Rte_IrTrigger . . . 242
6.38 Rte_Write . . . 243

7 RTE Runnable API Reference 245
7.1 Supported RTE Events . . . 245
7.2 Signature . . . 246
7.3 SWC Initialization . . . 247

8 VFB Tracing 248
8.1 Enabling VFB Tracing . . . 248
8.2 Trace Events . . . 248
8.3 Trace Event Implementation . . . 251
8.4 Optimization . . . 252

9 Memory Mapping and Compiler Abstraction 253
9.1 Memory mapping principles . . . 253
9.2 Memory mapping for code objects . . . 253
9.3 Memory mapping for data objects . . . 257
9.4 Reporting RTE objects to other AUTOSAR tooling . . . 260

9.5 Compiler Abstraction 263

10 External Dependencies **264**

10.1 C Library 264
10.2 OS Configuration 264
10.3 AUTOSAR COM 267
10.4 Operating System 268
10.5 Calibration 270

11 Parameters of Implementation **277**

11.1 AUTOSAR Common Published Information 277
11.2 API Legitimacy 277
11.3 Tasks and Runnable Entities 277
11.4 Queued Communication 278
11.5 Scheduling 278
11.6 Modes and Mode Switches 278
11.7 Inter-ECU Communication 279

12 AUTOSAR Revision Support **280**

13 Contact, Support and Problem Reporting **281**

1 About this Manual

The manual provides a complete reference to the syntax and semantics of the RTE configuration language, the operation of the RTA-RTE RTE generation tool, RTEGen, and the syntax and semantics of the generated RTE interface.

- Chapter 2 describes how to invoke the RTE generator and what output to expect
- Chapter 3 provides a reference of the command-line options of RTA-RTE.
- Chapter 4 provides a reference for the AUTOSAR XML used to configure RTA-RTE.
- Chapter 5 describes the RTA-RTE namespace, software component and API naming conventions in RTA-RTE.
- Chapter 6 presents a reference to the API as seen by software components. The API includes calls for sender-receiver and client-server communication, concurrency control and access to data memory sections.
- Chapter 7 explains how the runnable entities are declared using the RTE API and provides a reference to the different classes of runnable entity.
- Chapter 8 describes how VFB tracing events are configured and used.
- Chapter 9 explains how elements within the generated RTE can be mapped to different memory segments using the AUTOSAR memory mapping and compiler abstraction.
- Chapter 10 describes the external objects (e.g. OS objects) required by a generated RTE and defines the APIs provided by external AUTOSAR modules that are used by the generated RTE.
- Chapter 11 defines limits and constraints imposed by RTA-RTE on the generated RTE.

1.1 Who Should Read this Manual?

The RTA-RTE Reference Manual is intended for the software engineer who understands the concepts and general techniques of developing an RTE-based application and needs to know key technical detail about configuration and implementation.

It is assumed that the reader is familiar with the RTA-RTE User Guide.

1.1.1 Related Documents

This document is intended to be read in conjunction with the *RTA-RTE User Guide*.

This document also references information contained in the AUTOSAR Software Specifications, in particular *AUTOSAR Specification of RTE*.

1.2 Document Conventions

Notes that appear like this contain important information that you need to be aware of. Make sure that you read them carefully and that you follow any instructions that you are given.

Notes that appear like this describe things that you will need to know if you want to write code that will work on any target processor.

In this guide you'll see that program code, header file names, C type names, C functions and API call names all appear in the `courier` typeface. When the name of an object is made available to the programmer the name also appears in the courier typeface, suitably modified in accordance with the RTE naming conventions. So, for example, a runnable called Runnable1 appears as a handle called `Runnable1`.

1.3 Acronyms and Abbreviations

AUTOSAR	AUTomotive Open System ARchitecture - a standardized software architecture targeted at automotive applications aimed at fostering the reuse of application software over multiple vehicle platforms.
BNF	Backus-Naur Form; a notation used to describe language grammars.
ECUC	AUTOSAR ECU Configuration
RTA-OSEK	An AUTOSAR SC1 and OSEK 2.2.3 compatible operating system from ETAS GmbH.
RTE	AUTOSAR Run-Time Environment. See "Introduction to the RTE" in the *RTA-RTE User Guide*.
RTA-RTE	The ETAS AUTOSAR RTE Generator Product. This includes the AUTOSAR RTE Generator Tool responsible for reading the AUTOSAR XML configuration and generating the RTE and associated C header files. RTA-RTE distributions also include the RTE library, all user documentation and an example application.
XML	eXtensible Markup Language used to describe AUTOSAR configurations.
RTEGen	The ETAS AUTOSAR RTE generator tool responsible for reading the AUTOSAR XML configuration and generating the RTE and associated C header files.

URI	Uniform Resource Identifier – a character string that identifies (names) a resource. Within XML a URI identifies a namespace.

2 Invocation

The RTA-RTE RTE generator is a Win32 executable that provides multiple functions:

- Generation of a “contract” API for use by software components during development.
- Generation of an optimized production RTE for a specific target ECU.
- Optional creation of an OS configuration for RTE created OS objects.
- Optional creation of a COM OIL configuration for RTE created COM objects.

Each core function forms an execution phase. The two “contract” and “RTE” phases are typically widely separated in time with the “contract” phase occurring before development of a component starts and the “RTE” phase after component deployment is complete.

RTEGen takes one or more XML-based configuration files as input. The structure of these files is defined in Chapter 5.

2.1 Command-line Usage

The RTE generator is invoked from the command line as follows:

```
RTEGen [options] <input files>
```

Command line options can be specified using either short or long (GNU style) names – the supported options are listed below.

Any number of XML input files can be specified.

When a command line option takes an argument, the argument can be specified either as a trailing word or with an equals sign. For example, given an option opt with argument `arg` the option could be specified as either “`--opt arg`” or “`--opt=arg`”. The two forms are equivalent and can be mixed on the command line.

The ordering of command line parameters is unimportant: options and XML files can be mixed freely. Command line options are read left-to-right and are processed before any input files are read.

See Chapter 3 for a reference to all RTA-RTE command-line options.

2.2 Output Files

Each execution of the RTE generator in RTE-generation phase creates output for a single ECU instance. In contract phase, files are generated for each application-software component specified on the command line.

Table 2.1 describes the output files generated by the RTE generator in RTE-generation and contract phases.

File	Description	Contract?	RTE?
`Rte.h`	Core RTE header file.	✓	✓
`Rte_Intl.h`	Private RTE declarations and definitions.	✓	✓
`Rte_Main.h`	RTE lifecycle API declarations.	✓	✓
`Rte_Lib.c`	The RTE library.	✗	✓
`Rte.c`	The RTE source file.	✗	✓
`Rte_Cbk.h`	C header file containing prototypes for all call-back functions created within the generated RTE.	✗	✓
`Rte_Const.h`	A C header file containing RTE configuration constants. See Section 2.5.	✗	✓
`Rte_Hook.h`	VFB trace hook definitions	✗	✓
`Rte_Type.h`	C header file containing definitions of the types described in the input file. This file is automatically included by other generated files.	✓	✓
`Rte_<SWC>.h`	A C header file containing the RTE API customized for each software component specified in the input file. This file is the component's application header file.	✓	✓

File	Description	Contract?	RTE?
<TaskName>.c (Vendor mode only)	A C source file containing a single OSEK task for each <TaskName> defined in the configuration file. In compatibility mode task bodes are created within the generated RTE file.	✗	✓
Rte_BSWMD.arxml	AUTOSAR XML file describing the features of the generated RTE code. Note that the McSupportData is written to a separate file following the AUTOSAR splittable pattern.	✓	✓
Rte_McSupportData.arxml	AUTOSAR XML file containing the McSupportData, that is, information need to generate A2L files for measurement/calibration tools. This can be regarded as an excerpt from the BSW Module Description and indeed can be merged into it following the AUTOSAR splittable pattern.	✗	✓
Rte_Catalog.xml	An XML file containing the actual filenames used in the RTE output.	✓	✓

Table 2.1: Generated Output Files

The following files are optional – whether or not they are generated depends on the configuration of the RTE generator.

File	Description	Contract?	RTE?
Rte.err	RTE error file (see –err option).	✓	✓
OS configuration	An XML/OIL configuration file for the AUTOSAR/OSEK Operating System.	✗	✓
COM configuration	An OIL/configuration file for AUTOSAR COM R1.0.	✗	✓

Table 2.2: Optional Output Files

2.2.1 Redirecting Output

By default all output files are generated in the current directory – this is typically the same directory as the input file. The `--output` option can be used to direct specific output files to a defined folder.

For example, the RTA-RTE RTE generator can be directed to write all generated C files to folder abc using the following option:

```
--output="[*.c]abc"
```

The pattern specified using the `-o` option does not need to include a wild card. The following option directs only the generated RTE to folder `abc`:

```
--output "[Rte.c]abc"
```

The `--output` option can be specified multiple times on the command-line. Options are processed left-to-right and therefore different patterns for output redirection are also processed left-to-right. For example, to redirect `Rte_Type.h` to one folder and all other generated header files to another folder one could use one must specify the more general pattern last:

```
--output "[Rte_Type.h]folder1" --output "[*.h]folder2"
```

2.3 OS Configuration File

RTA-RTE can optionally generate an OS configuration file[1] that defines all OS objects used by the generated RTE.

The generated OS configuration file does not contain any target specific information – only target-neutral OS objects are defined. Therefore the file should be combined with an additional OS configuration file that defines target information (e.g. OS status, hook usage, ISRs, etc.). Depending on the OS, the additional OS configuration could reference the generated OS configuration file using a “#include” mechanism (RTA-OSEK and generic OSEK), auxiliary OIL files (RTA-OSEK only) or by merging XML files (RTA-OS3.0).

Generation of the OS configuration requires system information and therefore it is created during RTE phase only.

See Section 10.2 for details of the OS objects created within the OS configuration file.

2.4 COM OIL File

RTA-RTE can optionally generate a AUTOSAR COM R1.0 configuration file, `rta-com.oil`. This file defines all COM objects (messages and I-PDUs) used by the generated RTE.

[1] The filename depends on the selected OS plug-in. The default filename used by the RTA-OSEK plug-in is `rta-osek.oil`.

As with the generated OS configuration, the COM configuration does not contain any target specific information – only target-neutral COM objects are defined. Therefore the file should be used with a "wrapper" OIL file that defines target information (COM status, timebase etc.) and the references the generated OIL file using the "`#include`" mechanism.

Generation of the COM configuration requires information on how software components communicate and therefore it is created during RTE phase only.

Support for generating a COM configuration file is not included in all versions of RTA-RTE.

2.5 RTE Configuration Constants

During RTE phase, RTA-RTE creates the file `Rte_Const.h`. This file defines constants derived from the configuration that are used to optimize the compilation of the RTE library.

2.5.1 C Library

By default, RTA-RTE is independent of the C library and uses the RTE library function `Rte_memcpy` when copying memory.

Alternatively, RTA-RTE will use the standard C Library `memcpy` function if the symbol `RTE_LIBC_MEMCPY` is defined when compiling the RTE library and RTE generated code. Use of the standard function from the C library may be preferred if, for example, the target compiler supports a built-in function that compiles to inline optimal assembler.

The `RTE_LIBC_MEMCPY` symbol can either be placed within the user configuration file (see Section 2.10) or on the command-line when compiling the RTE library and RTE generated code, for example:

```
... -DRTE_LIBC_MEMCPY
```

2.5.2 Calibration Method

Constant	Description
RTE_CALPRM_SINGLE_PTR	Defined if the selected global calibration method is "single".
RTE_CALPRM_DOUBLE_PTR	Defined if the selected global calibration method is "double".
RTE_CALPRM_INITRAM	Defined if the selected global calibration method is "initram".
RTE_CALPRM_NONE	Defined if the selected global calibration method is "none".

2.5.3 Measurement

Constant	Description
RTE_MEASUREMENT_SUPPORT	Defined as “1” if measurement is enabled in the RTE module configuration and “0” otherwise.

2.5.4 Counters

The generated OS configuration uses two counters, `Rte_Tick_Counter` for periodic activities (schedule table or periodic alarms) and `Rte_Tout_Counter` for sporadic alarms (timeouts, etc.). The tick rate for the counters is defined in `Rte_Const.h`.

Constant	Description
RTE_PERIODIC_COUNTER_TICK_-INTERVAL_US	The tick interval (in microseconds) of the counter used to drive the generated RTE’s Schedule Table or periodic alarms.
RTE_ALARM_COUNTER_TICK_-INTERVAL_US	The tick interval (in microseconds) of the counter used to drive the generated RTE’s timeout alarms. Not defined if no timeout alarm is required.

The counters are not ticked directly by user code but instead calls the generated API Rte_Tick_Timeouts at the counter’s tick rate. (see Section 6.35).

Main Function

The period of the RTE’s main function defaults to 10ms but can also be set explicitly on the command-line. The invocation rate in milliseconds is defined in Rte_Const.h.

Constant	Description
RTE_MAINFUNCTION_PERIOD_US	The tick interval (in microseconds) of RTE’s main function. Invocation of the RTE’s main function is only required when runnable entity minimum start intervals (see Section 4.10.10) are used.

2.5.5 OS Integration

The `Rte_Const.h` file includes constants that define the OS API and OS configuration format in use.

Constant	Description
RTE_OSAPI_AUTOSAR_R10	Defined if an AUTOSAR R1.0 compatible OS API is being used.
RTE_OSAPI_AUTOSAR_R30	Defined if an AUTOSAR R3.0 compatible OS API is being used.
RTE_OSAPI_OSEK	Defined if an OSEK compatible OS API is being used.
RTE_OSCFG_RTAOSEK	Defined if RTA-OSEK OS configuration file fragment is being used.
RTE_WOWP_EVENTS	Number of RTE defined events used within RTE generated code for handling timeouts and RTE activity.
RTE_OS_EVENTS	Number of OS events in use for runnable activation.
RTE_NULL_SCHEDULE	Defined if no periodic runnable entities exist.

The generated constants can be used to adapt application code to varying configurations. For example, an ISR activated every millisecond can be written to automatically tick the RTE's counter at the correct rate irrespective of the configured TimingEvents as follows:

```
#define DELAY_FACTOR (RTE_PERIODIC_COUNTER_TICK_INTERVAL_US / \
                                     US_PER_TICK )

static uint16 count = DELAY_FACTOR;

ISR(my1msISR)
{
  if ( --count == 0 )
  {
    Rte_Tick_Timeouts();
    count = DELAY_FACTOR;
  }
}
```

2.6 Screen Output

All screen output appears on the standard output. The RTA-RTE RTE generator will output the phase of generation followed by a log of operations performed. For example:

```
c:\rte_projects>\RTEGen --c /MyPkg/MySWC MyFile.xml
RTA-RTE v4.0.0
Copyright (C) ETAS GmbH 2004-2011
Loading MyFile.xml... done
```

```
URI: http://autosar.org/schema/r4.0
Phase is Contract (license verified, permanent)
Validating DOM... done
Building types database... done
Building reification tree... done
Generating intermediate XML... done
Generating RTE C... done
Generation complete

The following files were generated:
        Rte_MySWC.h
        Rte_Type.h
```

2.7 Error and Information Messages

The RTA-RTE RTE generator presents information on the progress of RTE generation using a system of status messages. Messages have the following classification:

Fatal – the detected error prevents further processing and the RTE generator terminates immediately. No RTE or associated files are generated.

Error – the detected error is serious but does not prevent further processing. No RTE or associated files are generated.

Warning – the detected error does not prevent further processing. The RTE and associated files will be generated but should not be considered to be correct until the source of the warning has been investigated.

Information – a status message that does not indicate an error.

2.8 Exit Codes

In addition to progress and error messages, the RTA-RTE RTE generator returns the following error codes that can be used to confirm the success or otherwise of RTE generation:

- 0 : Success – the application headers (RTE and contract phase) or other files (RTE phase only) were generated without error.
- 1 : Failure – the input configuration was found to be invalid or generation failed for an environmental reason such as the output location not being writable.
- Other: Unexpected internal failure of the generator.

2.9 RTE Library

In addition to the generated `Rte.c`, RTA-RTE generates library code in `Rte_Lib.c` that must be compiled and linked along with the generated code and the application code to form the final executable.

The location to which `Rte_Lib.c` is generated can be altered using the `--output` command-line option.

RTA-RTE optimizes the `Rte_Lib.c` when it is generated and also through preprocessor constants (Section 2.5) defined in `Rte_Const.h`.

The RTE library must be recompiled each time the input configuration changes.

It is forbidden to call functions found in `Rte_Lib.c` except where documented in Chapter 6.

2.10 User Configuration File

RTA-RTE includes use of an optional user configuration file `Rte_UserCfg.h` that can be used to modify how the generated RTE and the RTE library are compiled.

RTA-RTE includes a default `Rte_UserCfg.h` and therefore it is only necessary to define a custom file to define different definitions.

The following constants can be defined in `Rte_UserCfg.h` to modify how the generated RTE and the RTE library are compiled.

Definition	Notes
RTE_LIBC_MEMCPY	When defined the use of the RTE library function `Rte_memcpy` is replaced by the standard C library function `memcpy`.

*For definitions within a custom user configuration file to have any effect the compiler's include path must be set so that the new user configutation file is read **before** the default file.*

3 Command-line options

The operation of the RTE generator is controlled via command-line options. All options begin with either the '-' or '@' characters; any other parameter on the command-line is interpreted as an input filename.

Parameters (i.e. filenames) specified either on the command-line or in sub-files that contain spaces must be quoted according to the rules of the invoking environment, e.g.:

```
RTEGen [options] "input filename.xml"
```

3.1 Examples

To display the RTA-RTE product and RTE generator versions using long-form option names:

```
RTEGen --version
```

To generate the contract for a software component 'swcA' in package 'pkgB' using short option names:

```
RTEGen --contract=/pkgB/swcA input.xml
```

To generate the RTE for the ECU instance referenced from ECU configuration `ecuConfig` in package 'pkgC' using short option names:

```
RTEGen --rte=/pkgC/ecuConfig ...
```

To generate the RTE using commands from subfile `MyCommandLine` while suppressing all informational messages:

```
RTEGen --quiet=3 --file MyCommandLine
```

To use the '`#warning`' pre-processor directive when issuing a warning within generated C code instead of the default '`#pragma message`':

```
RTEGen --warn=warning ...
```

3.2 Interaction with ECUC configuration

AUTOSAR defines certain configuration settings within the `RteGeneration` container that can also be specified on the RTA-RTE command-line:

ECUC Parameter	RTA-RTE Command-line option
`RteOptimizationMode`	`--optimize`
`RteCalibrationSupport`	`--calibration-method`
`RteVfbTraceEnabled`	`--vfb-trace`

ECUC Parameter	RTA-RTE Command-line option
`RteMeasurementSupport`	`--measurement`
`RteToolChainSignificantCharacters`	`--toolchain-significant-len`

For RTE generation phase, the option can be set either in the ECUC file or on the command-line. If specified in both places then RTA-RTE will use the command-line value – this enables simple override of the “fixed” configuration value.

For Contract phase, RTA-RTE does not read the ECUC generation container and therefore the options can only be specified on the command-line.

3.3 --

Description:

End of options

Name:

--

Parameter:

(None)

Default:

(None)

Notes:

Indicates the end of options list. All tokens on the command line after this option are treated as filenames.

This option is required when one or more filenames start with "--".

Example:

```
RTEGen.exe --rte=auto -- --model--file--name--with--dashes.arxml
```

3.4 `--append-name-to-buffer`

Description:

Append name to buffer symbol.

Name:

`--append-name-to-buffer`

Parameter:

This option takes a single parameter that specifies whether to include (‘1’) or exclude (‘0’) the name from the created receive buffer name.

Default:

0

Notes:

When RTA-RTE creates buffers to handle receive data or store measurable data, it names them using incrementing integers.

When this option is enabled, RTA-RTE appends the data element or operation argument name to the standard buffer name to make the generated code easier to read, with the risk that the identifiers become too long for some compilers or static checkers.

Example:

To cause RTA-RTE to append the data element name to the generated buffers:

```
--append-name-to-buffer=1
```

When this option is enabled, RTA-RTE creates receive buffers with names of the form `Rte_Rx_000000_<name>`.

3.5 `--atomic-assign`

Description:

Specify the SwBaseTypes that are assigned atomically on your target platform.

Name:

`--atomic-assign`

Parameter:

This option takes a comma-separated list of SwBaseType `shortNames` that describe types that do not need concurrency protection (e.g. `RTE_ATOMIC16()`).

Default:

All assigments are regarded as potentially in need of protection against read-modify-write errors.

Notes:

This option must be used with care: if applied to a type that is not atomic on your target platform, subtle run-time errors may occur that will be hard to track and eliminate.

Note that this option affects SwBaseTypes with the given shortName(s). It does not attempt to match SwBaseTypes with a different shortName, even if the size and alignment are the same. (RTA-RTE does not know how, for example, nativeDeclaration might affect atomicity).

Example:

To suppress concurrency protection on 16- and 8-bit AUTOSAR Platform types, the following is sufficent:

```
--atomic-assign=uint16,uint8,sint16,sint8
```

3.6 `--bit-pack-type`

Description:

Specify underlying `ImplementationDataType` for bit-packed flags.

Name:

`--bit-pack-type`

Parameter:

This option takes a single parameter which is a reference to the `ImplementationDataType` to use to contain bitfields.

Default:

`/AUTOSAR_Platform/ImplementationDataTypes/uint16`

Notes:

None.

Example:

To use the `uint32` platform type for holding bit-packed flag in generated code, use the following command:

```
--bit-pack-type=/AUTOSAR_Platform/ImplementationDataTypes/uint16
```

3.7 `--bsw`

Description:

Select “BSW” generation phase to generate the BSW Scheduler components only.

Name:

`--bsw`

Parameter:

This option takes a single parameter, that specifies either the ECU instance or the ECU configuration for which BSW generation should occur.

Default:

N/A

Notes:

The ECU instance `<ECUI>` must be specified using an absolute instance reference. (See Section 4.2.4.)

It is an error if the input XML contains any SWC configuration data.

Example:

To select BSW generation phase for ECU configuration `/pkg/ecu` use the following command:

```
--bsw=/pkg/ecu
```

3.8 `--bsw-scope-limit-defns`

Description:

Control use of scope-limiting definitions for BSW.

Name:

`--bsw-scope-limit-defns`

Parameter:

This option takes a single parameter, <P>, that determines whether scope-limiting definitions are generated for BSW APIs within the Module Interlink Header file. Generation is not required for AUTOSAR compliance.

Default:

If this option is not specified scope limiting definitions are generated in the same form as used for application header files.

Notes:

This option is supported by the OutputC plug-in.

Example:

To use enable generation of scope-limiting definitions, use:

```
--bsw-scope-limit-defns=on
```

3.9 `--calibration-disable`

Description:

Disable RTE calibration supported for specified SWC type.

Name:

`--calibration-disable`

Parameter:

This option takes a single parameter, `<SWC>`, which must be an absolute reference to the SWC type for which calibration should be disabled.

Default:

N/A

Notes:

This option has no effect if the selected global calibration method is ‘none’. Calibration can also be disabled for individual SWC types using the ECU Configuration description.

Example:

To disable calibration for SWC `/pkg/swcA` use the following command:

```
--calibration-disable=/pkg/swcA
```

3.10 `--calibration-instantiation`

Description:

Determines whether RTA-RTE allocates memory (RAM) for calibration instances or imports labels.

Name:

`--calibration-instantiation`

Parameter:

This option takes a single parameter, that specifies whether to import (`import`) or allocate memory (‘`allocate`’) for calibration instances.

Default:

`allocate`.

Notes:

None.

Example:

The command-line option:

```
--calibration-instantiation=allocate
```

Causes RTA-RTE to allocate RAM buffers for each calibration group. These buffers should be initialized by the application before the RTE is started. (See section **??**.)

Alternatively, the command-line option:

```
--calibration-instantiation=import
```

Causes RTA-RTE only to import labels using the `extern` keyword. These labels can be set to buffers which are initialized externally to the generated RTE code.

3.11 `--calibration-method`

Description:

Select the global calibration method.

Name:

`--calibration-method`

Parameter:

This option takes a single parameter that specifies the selected calibration method. Supported values are: `none`, `singlePointered` `doublePointered`, `initializedRam`, and `singlePointered2`

Default:

`none`

Notes:

The selected calibration method affects the data structures and generated functions created to support calibration. The API presented to SW-Cs within the application header is not affected.

For RTE generation phase, this option can be set both in the ECUC file and on the command-line. If specified in both places then RTA-RTE will use the command-line value.

Example:

To select *single-pointered* method:

```
--calibration-method=singlePointered
```

3.12 `--client-server-global-optimization`

Description:

Select whether or not non-AUTOSAR optimizations of inter-partition client-server communication should be performed.

Name:

`--client-server-global-optimization`

Parameter:

This option takes a single parameter which enables ("on" or "1") or disables ("off" or "0") use of non-AUTOSAR optimizations for inter-partition client-server communication.

Default:

Disabled ("off") for AUTOSAR compliance.

Notes:

None.

Example:

To enable the use of non-AUTOSAR optimizations for inter-partition client-server communication:

```
--client-server-global-optimization=on
```

3.13 `--com-symbolic-sigs`

Description:

Use symbolic names for COM or LdCom signal handles.

Name:

`--com-symbolic-sigs`

Parameter:

None.

Default:

By default, RTA-RTE generates an RTE that uses a `ComSignal`'s or `LdComIPdu`'s numerical handle ID when invoking COM or LdCom API functions.

Notes:

When this option is specified the generated RTE uses the symbolic name of the signal (the `shortName` of the corresponding `ComSignal` or `LdComIPdu`) instead of the handle ID. This option must be specified when RTA-RTE is used with an AUTOSAR v1.0 compliant COM.

Example:

To enable use of symbolic signal names:

```
--com-symbolic-sigs
```

3.14 --com-version

Description:

Modify the generated code to be appropriate for the given version of AUTOSAR COM.

Name:

`--com-version`

Parameter:

The parameter <V> specifies the required version. Supported and default COM versions are dependent on the selected RTA-RTE backend processor.

Default:

Dependent on selected backend processor.

Notes:

None.

Example:

To specify use of AUTOSAR COM v1.0:

```
--com-version=1.0
```

3.15 --contract

Description:

Execute contract phase for a specific software module.

Name:

`--contract`

Parameter:

This option takes a single parameter that must be an absolute reference to the ApplicationSoftwareComponentType type or BswImplementation for which contract-phase headers should be generated.

Default:

N/A

Notes:

Use `--contract` to support the AUTOSAR RTE Contract Phase or Basic Software Scheduler Contract Phase. RTA-RTE will generate an application header file for the specified application software component type or BSW implementation. To generate headers for multiple software modules, the `--contract` option can be repeated on the command line. You cannot mix contract phase and generation phase in the same run of RTA-RTE.

Example:

To generate the contract phase headers for two hypothetical Software Component Types swcA and swcB:

```
--contract=/myPackage/ApplicationSwComponentTypes/componentA --contract=/myPackage/A
```

3.16 `--deviate-allow-unmapped-swci-config`

Description:

Enable SWC instances within the RTE module configuration to be mapped to a different ECU instance.

Name:

`--deviate-allow-unmapped-swci-config`

Parameter:

This option permits ("1") or forbids ("0") SWC instances within the RTE module configuration to be mapped to a different ECU instance. When permitted a warning will be issued for each unmapped instance but generation will continue.

Default:

Forbid ("0").

Notes:

None.

Example:

To allow unmapped SWC instances:

```
--deviate-allow-unmapped-swci-config=1
```

3.17 `--deviate-appl-impl-compu-method`

Description:

This option suppresses the error that would be generated by having a CompuMethod on both an Application Data Type and its mapped Implementation Data Type.

Name:

`--deviate-appl-impl-compu-method`

Parameter:

This option enables (“1”) or disables (“0”) the deviation.

Default:

Enabled (“1”).

Notes:

Standard AUTOSAR behavior specifies that when the CompuMethod is in both an Implementation Data Type and its mapped Application Data Type that this is an error.

Example:

To make RTA-RTE raise a configuration error when the CompuMethod is on both the Application Data Type and its mapped Implementation Data Type (which is standard AUTOSAR behavior):

```
--deviate-appl-impl-compu-method=off
```

3.18 `--deviate-appl-impl-display-format`

Description:

This option suppresses the error that would be generated by having a DisplayFormat on both an Application Data Type and its mapped Implementation Data Type.

Name:

`--deviate-appl-impl-display-format`

Parameter:

This option enables (“on” or “1”) or disables (“off” or “0”) the deviation.

Default:

Enabled (“on”).

Notes:

Standard AUTOSAR behavior specifies that when the DisplayFormat is in both an Implementation Data Type and its mapped Application Data Type that this is an error.

Example:

To make RTA-RTE raise a configuration error when the DisplayFormat is on both the Application Data Type and its mapped Implementation Data Type (which standard AUTOSAR behavior)

```
--deviate-appl-impl-display-format=off
```

3.19 `--deviate-bsw-any-partition`

Description:

Enable mapping of BSW to any OS partition.

Name:

`--deviate-bsw-any-partition`

Parameter:

This option enables (“1”) or disables (“0”) an RTA-RTE deviation from the AUTOSAR RTE specification. When enabled BSW may be mapped to any OS partition.

Default:

Disabled (“0”).

Notes:

None.

Example:

To enable mapping of BSW to any partition:

```
--deviate-bsw-any-partition=1
```

This option has received limited testing in this release of RTA-RTE. If mapping to any partition is enabled the generated RTE must be thoroughly tested before use.

3.20 `--deviate-allow-supportsmulti-sharedmemorys`

Description:

Allow supportsMultipleInstantiation to be set on a SWCT with staticMemorys.

Name:

`--deviate-allow-supportsmulti-sharedmemorys`

Parameter:

This option enables ("1") or disables ("0") an RTA-RTE deviation from the AUTOSAR RTE specification. According to the specification, it is an error to set supportsMultipleInstantiation on an InternalBehavior that contains staticMemorys. When this option is enabled, the error is reduced to a warning, and an error is only raised if the input model actually contains multiple instances of the related SWCT.

Default:

Disabled ("0").

Notes:

None.

Example:

`--deviate-allow-supportsmulti-sharedmemorys=1`

3.21 --deviate-enum-cast

Description:

Explicitly cast literals used in enumerations (AUTOSAR TEXTTABLEs).

Name:

`--deviate-enum-cast`

Parameter:

This option takes a single parameter, <N>, that specifies whether the option is enabled (‘1’) or disabled (‘0’).

Default:

Disabled (AUTOSAR compliant).

Notes:

RTA-RTE writes preprocessor define directives for the symbolic values in TEXTTABLEs according to AUTOSAR (rte_sws 3810).

In addition to this, if the `--deviate-enum-cast` option is enabled, RTA-RTE also emits an explicit cast to the underlying ImplementationDataType.

Regardless of this option, RTA-RTE writes a ‘U’ suffix to the numeric literal if the underlying SwBaseType is unsigned or missing.

Example:

```
#define E1_VALUE1 34U
```

with `--deviate-enum-cast=1` becomes

```
#define E1_VALUE1 (myUnsignedEnumType)34U
```

3.22 `--deviate-group-calibration-none`

Description:

Control grouping of calibration parameters.

Name:

`--deviate-group-calibration-none`

Parameter:

This option takes a single parameter, <N>, that specifies whether to enable (‘1’) or disable (‘0’) the grouping of calibration parameters for ‘none’ calibration method.

Default:

Disable grouping (AUTOSAR compliant).

Notes:

RTA-RTE instantiates calibration parameters. For the single- and double-pointered calibration methods all parameters are grouped according to the assigned SwAddrMethod. This option enables grouping to also be applied when the ‘none’ calibration method is selected.

RTA-RTE will apply grouping when no flatmap instance descriptor is available since the descriptor is required to assign a name to the parameter instance.

Example:

To cause RTA-RTE to group calibration parameters when using the ‘none’ method:

```
--deviate-group-calibration-none=1
```

3.23 `--deviate-ignore-datatype-semantics`

Description:

Control semantic checks when checking type correctness.

Name:

`--deviate-ignore-datatype-semantics`

Parameter:

This option takes a single parameter, <P>, that specifies whether to ignore (‘1’) or enable (‘0’) the semantic checking of connected data types

Default:

Enable check (‘0’).

Notes:

AUTOSAR specifies compatibility rules for connected DataPrototypes involving the referenced AutosarDataType and any CompuMethods, Units, or PhysicalDimensions involved. Additionally, for R4.x projects, the AUTOSAR specification states that when VariableDataPrototypes are not compatible it is still permitted to connect them if they conform to further rules about whether automatic data conversion is possible.

In some configurations, it may be necessary to allow connection of DataPrototypes that do not fully conform to the AUTOSAR compatibility rules.

With this option specified, the compatibility and convertibility checks are very much relaxed, with the CompuMethod, Unit, and PhysicalDimension being completely ignored.

In addition, RTA-RTE normally checks that any CompuMethod referenced by an ImplementationDataType is permitted according to constr_1158, raising an error if the check fails. If this option is specified, then the error is downgraded to a warning, allowing the generation to continue.

Example:

To ignore semantic checks:

```
--deviate-ignore-datatype-semantics=1
```

3.24 `--deviate-implicit-cat2-mdd`

Description:

Enable mode disabling dependency for category 2 runnables.

Name:

`--deviate-implicit-cat2-mdd`

Parameter:

This option permits (“1”) or forbids (“0”) mode disabling dependency for *implicitly* category 2 runnables.

Default:

Forbid (“0”).

Notes:

Prior to AUTOSAR R4.0, a runnable is implicitly category 2 if it includes a synchronous call point. This option permits such runnables to have mode disabling dependencies.

Example:

To enable mode disabling dependencies for category 2 runnables:

```
--deviate-implicit-cat2-mdd=1
```

3.25 `--deviate-implicit-modify-for-loopbacks`

Description:

Enable “data modify” semantics for implicit access to data items where there is a loopback assembly connector.

Name:

`--deviate-implicit-modify-for-loopbacks`

Parameter:

This option enables (“on” or “1”) or disables (“off” or “0”) “data modify” semantics for implicit access to data items where a loopback assembly connector exists.

Default:

Disabled (“off”).

Notes:

Without this option, RTA-RTE implements AUTOSAR rules for the visibility and propagation of implicit data, and creates uninitialized Write Buffers for implicit writers to support that. One consequence of this is that Runnables using `Rte_IWriteRef` to write parts of a complex type will result in undefined values being propagated on the other members of the type.

When this option is enabled, if you connect the `PPort` back to an `RPort` characterized by the same interface in the same swc prototype, then RTA-RTE initializes the implicit writeback buffers from the definitive, global buffer before the Runnable enters. This enables the use of partial writes of complex data by multiple runnables without the propagation of undefined variables.

It is permitted, but not necessary, to configure a `DataReadAccess` in the `Runnable` for the configured `RPort`.

Example:

To enable “data modify” semantics for implicit access to data items:

```
--deviate-implicit-modify-for-loopbacks=on
```

3.26 `--deviate-memmap-decls`

Description:

Select whether or not memory allocation sequences should be used for declarations as well as definitions.

Name:

`--deviate-memmap-decls`

Parameter:

This option takes a single parameter which enables ("on" or "1") or disables ("`off`" or "0") use of MemMap for declarations.

Default:

Enabled ("on").

Notes:

None.

Example:

To disable generation of MemMap decorations:

```
--deviate-memmap-decls=off
```

3.27 `--deviate-omit-implicit-cds`

Description:

Enable optimization of CDS for implicit S/R and IRVs.

Name:

`--deviate-omit-implicit-cds`

Parameter:

This option enables (“1”) or disables (“0”) optimization of the component data structure (CDS) for implicit access to S/R and IRVs.

Default:

Forbid (“0”).

Notes:

Optimization of the CDS removes the data handles that are not required. Optimization is possible when the SWC is:

1. Singly instantiable,
2. Delivered as source code,
3. RTE generator is in vendor mode,
4. This option is enabled.

Example:

To enable optimization of the CDS for implicit S/R and IRVs:

```
--deviate-omit-implicit-cds=1
```

3.28 `--deviate-physical-dimension-compatibility`

Description:

Specify physical dimension compatibility rules.

Name:

`--deviate-physical-dimension-compatibility`

Parameter:

This option enables (“1”) or disables (“0”) an RTA-RTE deviation from the AUTOSAR RTE specification.

Default:

Disabled (“0”).

Notes:

This option modifies how the RTE generator validates compatibility of physical dimensions. By default RTA-RTE validates according to AUTOSAR rules and thus the physical dimensions must have the same short-name and attributes. However when this option is enabled RTA-RTE only checks the attributes, e.g. length exponent, match and permits the short-names to differ. This enables different elements that represent the same physical dimensions to be connected but should be used with care since physical dimensions with matching attributes can still represent different physical quantities. See the AUTOSAR documentation for further details.

Example:

To enable non-AUTOSAR compatibility rules for physical dimension comaptibility:

```
--deviate-physical-dimension-compatibility=1
```

3.29 `--deviate-prefer-no-empty-executions`

Description:

Enable ("on" or "1") or disable ("off" or "0") optimizations to runnable entity scheduling that avoid empty executions of tasks containing only runnables (or schedulable entities) triggered by the same source, at the expense of loss of the guarantee that the runnables will execute after being activated during the execution of the task.

Name:

`--deviate-prefer-no-empty-executions`

Parameter:

This option enables ("on" or "1") or disables ("off" or "0") optimizations to the scheduling of runnables that reduce execution overhead but cause the behavior to deviate from the AUTOSAR specification and may cause activations occurring during runnable execution to be delayed or lost.

Default:

Disabled ("off").

Notes:

When enabled, activation flags are elided for tasks containing only runnables that are activated by the same trigger source and such tasks are required to be configured with an activation limit of one. This avoids empty task executions in the case of bursts of runnable activations but activations occurring while the task executes are delayed or lost, deviating from the AUTOSAR requirement that activations of a runnable during its execution be honored.

Example:

```
--deviate-prefer-no-empty-executions=on
```

3.30 --deviate-split-swci-support

Description:

Split Sw-Cs across OsApplications

Name:

`--deviate-split-swci-support`

Parameter:

This option enables (“on” or “1”) or disables (“off” or “0”) splitting `ApplicationSofwareComponents` across protection boundaries, i.e. by mapping the `Runnables` to `OsTasks` in different `OsApplications`.

Default:

Disabled (“off”).

Notes:

When enabled, this option allows `RteEvents` from a SWC instance to be mapped to `OsTasks` in different `OsApplications`, in violation of AUTOSAR 4.3.0 `[SWS_Rte_07347]`.

The use of split SWCs is subject to the following restrictions. RTA-RTE will reject any configuration that violates these restrictions with an error.

1. There can be at most one instance of a given SW-C on the ECU if the SW-C instance is split.
2. BSW modules cannot be split, unless the `--deviate-bsw-any-partition` option is also set to “on”.
3. A split SW-C cannot use `ExclusiveAreas`.
4. A Runnable of a split SW-C may not be started by multiple RteEvents if it accesses port data or IRVs.
5. Each Data Item, Operation, Trigger, or Mode Group in a port (as appropriate for the port type) of a split SW-C must only be accessed by runnables mapped to tasks of one `OsApplication`.
6. Inter-runnable Variables within a split SW-C can be accessed by runnables mapped to different `OsApplications`, provided that:
   - There are not both multiple writing `OsApplications` and multiple reading `OsApplications` (that is, an IRV cannot be used for M:N communication between `OsApplications`)
   - There are not both multiple writing `OsApplications` and a reading Runnable in one of the writing `OsApplications` (that is, an IRV cannot be used for mixed intra- and inter-`OsApplication` N:1 communication).

   These restrictions only apply to IRVs that are accessed from multiple `OsApplications`. IRVs that are accessed only from a single `OsApplication` may be used as normal.

When this option is enabled, `InterrunnableVariables` might need to cross protection boundaries. RTA-RTE will silently augment the input model with port communication to handle this case, causing the `IoC` to be invoked where needed.

Example:

To map runnables from the same SWCInstance to `OsTasks` in different `OsApplications`,

```
--deviate-split-swci-support=on
```

3.31 `--deviate-trace-implicit-api`

Description:

Enable generation of VFB trace hooks for implicit API.

Name:

`--deviate-trace-implicit-api`

Parameter:

This option enables (“1”) or disables (“0”) an RTA-RTE deviation from the AUTOSAR RTE specification. When set VFB trace hook calls are added for implicit API functions / macros.

Default:

Disabled (“0”).

Notes:

None.

Example:

To enable VFB trace hook generation for the implicit API:

```
--deviate-trace-implicit-api=1
```

3.32 `--deviate-unconnected-pmode-behavior`

Description:

Control behavior of unconnected mode PPorts.

Name:

`--deviate-unconnected-pmode-behavior`

Parameter:

This option controls whether a mode manager `Rte_Switch` API stores the current mode (“on”) or discards the input parameters (“off”). When enabled the behavior of the API is an RTA-RTE deviation from the AUTOSAR RTE specification since AUTOSAR requires an unconnected mode manager to discard the inputs.

Default:

Disabled (“off”).

Notes:

None.

Example:

To enable storing of the current mode by an unconnected mode manager:

```
--deviate-unconnected-pmode-behavior=on
```

3.33 `--disable-warning`

Description:

Disable display of specified warning.

Name:

`--disable-warning`

Parameter:

The option takes a single parameter. <STR>, that specifies the identifier of the warning or informational message to be disabled. The option can be specified multiple times to disable multiple warnings.

Default:

`#pragma message`

Notes:

This option can disable the **display** of both warning and informational messages. When disabled RTA-RTE does not show the message and does not count the warning or information in the totals.

This option can be disabled within the INI file by setting the flag `DisableWarningOption` to "`disable`" within the section `Options`.

Example:

To disable information message `I53-7701`:

```
--disable-warning=I53-7701
```

3.34 `--error-as-warning`

This option is deprecated and will be removed in a future version of RTA-RTE. It should not be used in new projects. Existing projects should be updated to no longer use this option.

Description:
Demote specific error to warning.

Name:
`--error-as-warning`

Parameter:
The option takes a single parameter <STR>, that specifies the identifier of the error message to be demoted to a warning. The option can be specified multiple times to demote multiple errors.

Default:
N/A

Notes:
This option produces undefined behavior and should not be used.

This option should be disabled in production projects by adding `ErrorAsWarningOption=disable` to the `[Options]` section of `RTEGen.ini`.

This option demotes the severity of a message from "E" (error) to "W" (warning). As a result, RTA-RTE does not stop processing and will continue to attempt to generate code.

Because an error has occurred, behavior in subsequent steps is undefined.

Example:
To demote error message `E53-1234` to `W53-1234`:

```
--error-as-warning=E53-1234
```

3.35 `--error-report`

Description:

Select the message output method.

Name:

`--error-report`

Parameter:

This option takes a single parameter, <`Method`>, that specifies the destination and format of the Information, Warning, and Error messages. Supported values are

- `console` format messages and write to the standard error stream.
- `file` create file `Rte.err`. Format the messages as for console and write them to this file along with a summary.
- `xml` create file `RteErr.xml`. Format the errors and summary in XML and write to the file.

Default:

"console"

Notes:

Example:

To send all generated errors to a file use the following option:

```
--error-report=file
```

3.36 `--exclusive-area-optimization`

Description:

Set optimization policy of RTE exlusive area APIs.

Name:

`--exclusive-area-optimization`

Parameter:

This option takes a single parameter, <P>, that specifies whether to “`enable`” or disable “`disable`” optimization

Default:

“enable”

Notes:

When exclusive area optimization is disabled:

- For explicitly accessed exclusive areas the generated `Rte_Enter`/`Rte_Exit` APIs are not mapped “null” implementations even when all accessors are already in mutual exclusion (e.g. mapped to same task).
- For implicitly accessed exclusive areas, RTA-RTE will perform no optimization to “null” implementation of enter/exit locks created within the highest priority task.

Example:

To disable optimization of exclusive area access:

```
--exclusive-area-optimization=disable
```

3.37 `--fast-init`

Description:

Enable fast activation for mode switch events.

Name:

`--fast-init`

Parameter:

This option takes a single parameter, <REF>, that specifies either an atomic SWC type or a ModeSwitchEvent.

Default:

The default activation policy is AUTOSAR compliant activation therefore this option needs to be specified for each event that is to be activated by the non-AUTOSAR compliant mechanism.

Notes:

Enables ModeSwitchEvents to be activated by a non-AUTOSAR compliant mechanism (for example, a function call from the body of a task started outside the control of the RTE). ModeSwitchEvents may be specified either individually by name or as a group by naming the SWC type to which they belong. This avoids the complexity inherent in AUTOSAR-compliant activation for mode switch activations and is especially useful for “init” runnables that are activated only once.

Example:

To enable fast activation for RTE Event ev1 within an internal behavior IB:

```
--fast-init=/pkg/IB/ev1
```

3.38 `--file`

Description:

Read options from command file.

Name:

`--file`

Parameter:

The file from which commands are read.

Default:

N/A

Notes:

Read command-line options from the specified file in addition to any read from the command line. The option can be used recursively; a file read using `--file` can include other files.

The '`@`' character can be used as a synonym for the `-file` option. The space separating '`@`' and <FILE> into separate command-line arguments is optional.

Command line parameters included with this must observe the same rules as if they were specified directly on the command line with the exception that options can be split across multiple lines.

Comments can be included in a command file. A comment starts with semicolon character (';') either at the start of the line or after some whitespace. Text up to the end of the line is ignored.

The file should be a plain ASCII text file. No special file extension is required.

Example:

The following examples both read options from the file `project.rte`:

```
--file=project.rte
```

```
@project.rte
```

3.39 `--force-basic-tasks`

Description:

Force basic tasks.

Name:

`--force-basic-tasks`

Parameter:

This option takes no parameters. If omitted, then the task's forced-basic semantics are taken from the ECU Configuration file (see Section 4.19.2).

Default:

N/A

Notes:

When specified RTA-RTE uses forced-basic semantics (see *RTA-RTE User Guide*) for all tasks in the ECU instance for which the RTE is being generated. This option overrides any settings in the ECU Configuration file.

Example:

To enable force-basic semantics for all tasks:

```
--force-basic-tasks
```

3.40 `--have-64bit-int-types`

Description:

Enable support for 64-bit platform types for use within generated data-transformation.

Name:

`--have-64bit-int-types`

Parameter:

This option takes a single parameter, <P>, that specifies whether to enable (‘1’) or disable (‘0’) the use of 64-bit types.

Default:

Disabled (‘0’)

Notes:

This option enables use of 64-bit types within generated data-transformation functions. Since these types are not standardized by AUTOSAR the option is disabled by default. If enabled the types must be defined when the generated RTE is compiled.

Example:

To indicate to RTA-RTE that 64-bit types are available:

```
--have-64bit-int-types=1
```

3.41 --help

Description:

Display RTE generator help.

Name:

`--help`

Parameter:

None

Default:

N/A

Notes:

Print the usage information on the standard output. Brief usage information is presented when using `-?`/`-h` and more detailed information with `--help`.

Example:

To display the help text: `--help`

3.42 `--implicit-allocation-method`

Description:

Select the allocation method used by RTA-RTE for creating implicit communication buffers. Supported methods are 'overlay' and 'task'.

Name:

`--implicit-allocation-method`

Parameter:

This option takes a single parameter which is the method to use. Supported values are "`task`" and "`overlay`".

Default:

"`overlay`"

Notes:

Method "`task`" causes a separate structure to be created and instantiated by RTA-RTE for each task's implicit buffers. The structure instance is allocated to its own memory section called `SEC_VAR_IMPLICITSR_<TASK>` where <TASK> is the task name in uppercase. Method "`overlay`" creates a single structure where tasks that cannot preempt (e.g. those at the same priority) have overlayed implicit buffers.

Example:

To enable separate structures for each task's implicit buffers:

```
--implicit-allocation-method=task
```

3.43 `--implicit-read-return-const`

Description:

Control whether or nor the CONST or VAR compiler abstraction macros are used to cast the return value from `Rte_IRead`.

Name:

`--implicit-read-return-const`

Parameter:

This option takes a single integer parameter which defines cast used.

0 Use of CONST cast disabled; the API mapping uses a VAR cast.

1 Use of CONST cast enabled.

Default:

1 (CONST cast).

Notes:

None.

Example:

To enable use of a VAR cast:

```
--implicit-read-return-const=0
```

3.44 `--implicit-use-global-buffers`

Description:

Enable or disable use of global receive buffers in place of task-specific buffers for implicit communications. This optimization is dependent on task preemption but when possible can save RAM since no additional copies of the global data are required.

Name:

`--implicit-use-global-buffers`

Parameter:

This option takes a single integer parameter which defines the enabled optimizations.

0 Optimization disabled; all implicit communication uses AUTOSAR compliant task-specific buffers.

1 Optimization of implicit communication to use global buffers is enabled. The possible optimization depends on the relative priorities of tasks containing readers and writers: for best results either map to tasks at the same priority or map to the same task.

2 As ’1’ plus all ’fast-init’ tasks use global buffer access for implicit communication. For ’fast-init’ tasks the optimization occurs irrespective of the task mapping of readers and writers since it is assumed that execution of the ’fast-init’ tasks is complete before periodic runnables (and hence normal RTE tasks) start.

Default:

0 (optimization disabled and AUTOSAR compliant task-specific buffers used).

Notes:

None.

Example:

To enable use by the generated RTE of global receive buffers:

```
--implicit-use-global-buffers=1
```

3.45 `--incremental-build`

Description:

Incremental Build.

Name:

`--incremental-build`

Parameter:

This option takes a single parameter that enables (‘1’) or disables (‘0’) incremental output of generated files. If not specified incremental output is disabled.

Default:

Build all files (incremental build disabled).

Notes:

When enabled, RTA-RTE generates files to a temporary folder and only overwrites files in the destination folder if the contents have changed.

This option turns on the `--notimestamps` option.

Example:

To enable incremental build:

```
--incremental-build=1
```

3.46 `--initial-value-rounding`

Description:

Select the rounding behavior for the calculation of initial values for integer data from physical values.

Name:

`--initial-value-rounding`

Parameter:

This option takes a single parameter which specifies the required rounding behavior for the calculation of initial values for integer data types from physical values. The supported rounding behaviors are ‘truncate’ meaning truncation towards zero and ‘nearest’ meaning rounding to nearest, with half values rounding away from zero.

Default:

truncate

Notes:

When a physical value is given with an ApplicationValueSpecification the computation of the corresponding internal value may result in a fractional value that must be rounded in the case that the destination data type is an integer. This option allows that rounding behavior to be selected.

The ‘truncate’ behavior means to truncate towards zero, so for example 2.3 and 2.8 both become 2 and -1.2 and -1.9 both become -1.

The ‘nearest’ behavior means to round to nearest, with half values rounding away from zero, so for example 2.4 becomes 2, 2.8 and 2.5 both become 3, -1.2 becomes -1 and -1.7 and -1.5 both become -2.

Example:

To select rounding to nearest behavior:

```
--initial-value-rounding=nearest
```

3.47 `--ioc-header`

Description:

Set the IOC header file used.

Name:

`--ioc-header`

Parameter:

This option takes a single parameter, <FILE>, that specifies the name of the IOC header file to use within generated code.

Default:

None.

Notes:

This option is valid in vendor mode only.

Example:

To use IOC header `ioc.h`:

```
--ioc-header=ioc.h
```

3.48 `--ioc-xml-namespace`

Description:

Set the XML namespace URI used in generated IOC configuration file.

Name:

`--ioc-xml-namespace`

Parameter:

This option takes a single parameter, `<URI>`, that specifies the namespace URI to be used within the generated IOC configuration file.

Default:

If this option is not specified the default namespace URI is the R4.0 default namespace.

Notes:

This option is supported by the RTA-IOC OS plug-in.

Example:

To use `http://namespace` as the namespace URI when generating the IOC configuration file, use:

```
--ioc-xml-namespace=http://namespace
```

3.49 `--local-mcsd`

Description:

Report “local McSupportData” for a specific software module.

Name:

`--local-mcsd`

Parameter:

This option takes a single parameter that must be an absolute reference to the ApplicationSoftwareComponentType type whose internal data are to be reported in McSupportData.

Default:

N/A

Notes:

This is a non-AUTOSAR generation phase that generates McSupportData for specific software modules. RTA-RTE will generate an McSupportData report containing the staticMemorys, constantMemorys and perInstanceParameters of the given modules. To include multiple modules in the McSupportData, specify the `--local-mcsd` option multiple times on the command line. You cannot mix “local MCSD phase” with any other phase in the same run of RTA-RTE.

Example:

To generate McSupportData for two hypothetical Software Component Types swcA and swcB:

```
--local-mcsd=/MyPackage/ApplicationSwComponentTypes/swcA --local-mcsd=/MyPackage
```

3.50 `--makedep`

Description:

Output dependency information for generated files.

Name:

`--makedep`

Parameter:

The option takes one parameter, `<FILE>`, which is the file to which dependency information is to be written.

Default:

N/A

Notes:

None.

Example:

To enable generation of dependency information and output it to file `rte.dep` use:

```
--makedep=rte.dep
```

3.51 `--mcore-spinlocks-always`

Description:

Enable spinlocks in multicore mode handling.

Name:

`--mcore-spinlocks-always`

Parameter:

This option enables (“1”) or disables (“0”) spinlocks in multicore mode handling.

Default:

Disabled (“0”).

Notes:

When enabled, RTA-RTE emits spinlocks for concurrency protection in Mode APIs. At the time of writing, RTA-RTE does not optimize spinlocks, so specifying this option will cause all Mode Machine Instances to use spinlocks even if there is no inter-core communication to protect against. For this reason, the option should not be used if the input model contains no inter-core mode handling.

On configurations detected as single core this option will be ignored assuming the default Disabled.

Example:

Enable inter-core mode handling: `--mcore-spinlocks-always=1`

3.52 `--mcsd-policy`

Description:

Specify options pertaining to the output of Measurement and Calibration Support Data (MCSD).

Name:

`--mcsd-policy`

Parameter:

This option takes a single parameter: a comma-separated list of options which modify the MCSD as follows:

emit-memorys emits `McDataInstance` containers for any BSW or ASW Static or Constant Memory.

phys-constrs-always RTA-RTE shall always write PhysConstrs related to every McDataInstance. If necessary, the PhysConstr will be taken from The ApplicationDataType, ImplementationDataType, CompuMethod (for enumerated types) or SwBaseType (for Category NONE, 2C or BOOLEAN). If no PhysConstr can be found or calculated, an error is raised.

mcfunction-from-shortname For all McDataInstances where RTA-RTE sees a relevant DataPrototype, that DataPrototype's shortName is copied to the McDataInstance's `McFunction`. This policy is deprecated; it was implemented as a workaround for use before RTA-RTE exported McFunction correctly.

struct-element-symbols For McDataInstances that are structure elements, the C names for the elements are emitted as their symbols. These are not global linker symbols but they do allow the full C expressions to access the elements to be constructed.

Default:

N/A

Notes:

N/A

Example:

`--mcsd-policy=emit-memorys`

3.53 `--measurement`

Description:

Globally enable (or disable) support for measurement.

Name:

`--measurement`

Parameter:

The option takes a single parameter, <V>, that specifies whether measurement is enabled (“1”, “2” and “3”) or disabled (“0”)

Default:

Enabled (“1”)

Notes:

With parameter “1”, each data element, client-server argument and inter-runnable variable that is to be measured must be configured separately.

With parameter “2”, measurement is enabled for all data elements and inter-runnable variable irrespective of the configuration within the XML input. This setting therefore enables items to be measured in 3rd party components (for which source is available) without modifying the source XML.

Parameter “3” extends “2” to also measure all client-server arguments.

For RTE generation phase, this option can be set both in the ECUC file and on the command-line. If specified in both places then RTA-RTE will use the command-line value — this enables simple override of the “fixed” configuration value.

Example:

To enable measurement for all data elements (including inter-runnable variables) irrespective of the settings in the input configuration use the following option:

```
--measurement=2
```

3.54 `--memory-sections`

Description:

Specify location of the *Memory Section Description File*.

Name:

`--memory-sections`

Parameter:

<PATH> — specifies the folder containing the memory section description file. The specification can be an absolute or relative path. A relative path is interpreted relative to the folder containing the current folder.

Default:

File “`memsect.xml`” within the folder containing the application executable. If the file is specified both in the INI configuration file and on the command-line the latter takes precedence.

Notes:

See *RTA-RTE Toolchain Integration Guide* for further details on using the *Memory Section Description File* to adapt the AUTOSAR compiler abstraction usage within generated code.

Example:

To use memory section description file `mymemsect.xml`:

```
--memory-sections=mymemsect.xml
```

3.55 `--notimestamps`

Description:

Disable timestamps in generated files.

Name:

`--notimestamps`

Parameter:

None

Default:

Include timestamps.

Notes:

Output fixed-text banner (omit date and time of generation in generated files). This option is useful when the generated output will be programmatically compared, e.g. by a source control system.

Example:

To disable timestamp generation:

```
--notimestamps
```

3.56 `--operating-system`

This option is deprecated and will be removed in a future version of RTA-RTE. It should not be used in new projects. Existing projects should be updated to no longer use this option.

Description:

Select which OS support to use.

Name:

`--operating-system`

Parameter:

The option takes a single parameter, `<OS>` that is the name of the OS to use. Supported parameters are:

- `autosar40` - generate OS APIs and configuration files compatible with an AUTOSAR 4.x operating system

Default:

`autosar40`

Notes:

The selected OS determines both the OS API used within the generated RTE and also the form of the generated OS configuration file (if any).

Example:

To select the `autosar40` OS support:

```
--operating-system=autosar40
```

3.57 `--optimize`

Description:

Set the optimization strategy for the generated RTE.

Name:

`--optimize`

Parameter:

This option takes a single parameter, <TYPE> that specifies the optimization type. Supported values are “Runtime” (optimize for speed) and “Memory” (optimize for code size).

Default:

“Runtime” (speed)

Notes:

When optimized for “Memory” (size) RTA-RTE invokes the COM API directly to access non-queued signals rather than allocating buffers for storage. The optimization strategy can also be set using the RTE Generation parameters within the ECU Configuration description. A setting on the command-line overrides a setting in the ECU Configuration description.

The short-form of this option is an uppercase letter “O”.

For RTE generation phase, this option can be set both in the ECUC file and on the command-line. If specified in both places then RTA-RTE will use the command-line value — this enables a simple override of the “fixed” configuration value.

Example:

To enable optimization for “memory” usage:

```
--optimize=Memory
```

3.58 `--os-define-osenv`

Description:

Define OSENV within `Rte_Const.h`.

Name:

`--os-define-osenv`

Parameter:

This option takes a single parameter which is the OSENV <NAME> to define. The two supported values are:

- `RTAOS40` - for versions of RTA-OS which support AUTOSAR 4.x.
- `UNSUPPORTED` - for all other operating systems.

Default:

If this option is not specified, then `OSENV_<NAME>` must be defined when the RTE is compiled.

Notes:

Usually the `OSENV_<NAME>` should be defined when the RTE is compiled. If it is not practical to change compiler flags for your project, for example the RTE is generated and compiled at different sites, then this option allows you to set the symbol when generating the RTE.

Example:

To define the OS environment as a version of RTA-OS that supports AUTOSAR 4.X:

```
--os-define-osenv=RTAOS40
```

3.59 `--os-fp`

Description:

Set whether or not user code invoked by RTE generated tasks uses floating-point operations/arithmetic support.

Name:

`--os-fp`

Parameter:

This option takes a single parameter that specifies whether floating point usage is disabled (“`off`” or “0”) or enabled (“on” or “1”).

Default:

Enabled (“on”).

Notes:

This option only affects the OIL configuration file created by AUTOSAR R1.0 OS plug-in. Its usage enables the additional optimizations included in RTA-OSEK 5.0 for when tasks do not use floating point.

Example:

To disable FP usage:

```
--os-fp=off
```

3.60 `--os-header`

Description:

Set the OS header file used.

Name:

`--os-header`

Parameter:

This option takes a single parameter, `<FILE>`, that specifies the name of the OS header file to use within generated code.

Default:

The default OS header files used are suitable for the primary target OS of the selected OS plug-in. However this option permits a different value to be set.

Notes:

This option is supported by the VFB Tracing and the OS configuration plug-ins.

Example:

To select `OS2.h` as the OS header:

```
--os-header=OS2.h
```

3.61 `--os-output-param`

Description:

Output all OS task parameters and references OR output only those that have changed.

Name:

`--os-output-param`

Parameter:

This option takes a single parameter, <P>, that specifies whether task parameters and/or referenced should be copied from the input to the generated OS configuration file. Supported values are “changed” and “`all`”.

Default:

“changed”

Notes:

When set to (“`all`”) this option causes RTA-RTE to output all OS task parameters (priority, activation limit and schedule) and OS task references (OS resources) regardless of whether they have been changed.

When set to (“changed”) this option causes RTA-RTE to output only those OS task parameters and OS task references (OS resources) that it has modified.

This option can be used with both the AUTOSAR30 and AUTOSAR40 OS support. See the *RTA-RTE Toolchain Integration Guide* for further details on working with RTA-RTE and the `osparam` option.

Example:

To output all OS parameters in the generated OS configuration:

```
--os-output-param=all
```

3.62 `--os-permit-extended-tasks`

Description:

Configure whether the RTE generator is permitted to create extended tasks.

Name:

`--os-permit-extended-tasks`

Parameter:

This option takes a single parameter, <P>, that specifies whether generation of extended tasks by RTA-RTE is permitted (“1”) or forbidden (“0”).

Default:

Enabled (“1”).

Notes:

If extended tasks are disabled (option parameter “0”) then certain runnable-entity mappings that require extended tasks are invalid; most notably the mixing of runnables triggered by TimingEvents with runnables triggered by other RTE Events.

Example:

To disable support for extended tasks:

```
--os-permit-extended-tasks=0
```

3.63 `--os-task-as-function`

Description:

Determine if generated tasks are created using the AUTOSAR OS macro TASK or as function definitions.

Name:

`--os-task-as-function`

Parameter:

This option takes a single parameter, <P>, that specifies whether tasks are output as functions or TASKs. When defined as “1” generated tasks are created as functions which can be invoked by legacy systems. See the *RTA-RTE Toolchain Integration Guide* for details.

Default:

Disabled (“0”).

Notes:

This option is supported by the C Output and OS output plug-ins. When enabled in vendor mode no task-specific header files are referenced within the generated task files.

When enabled, RTA-RTE replaces the `TASK()` macro in generated output with a function definition.

This option is incompatible with runnables that specify a minimum start interval since the execution of such runnables must be controlled by the RTE.

Example:

To enable generation as tasks, use:

```
--os-task-as-function=1
```

3.64 `--os-xml-namespace`

This option is deprecated and will be removed in a future version of RTA-RTE. It should not be used in new projects. Existing projects should be updated to no longer use this option.

Description:
Set the XML namespace URI used in generated OS configuration file.

Name:
`--os-xml-namespace`

Parameter:
This option takes a single parameter, <URI>, that specifies the namespace URI to be used for the generated OS configuration file.

Default:
If this option is not specified the default is default namespace URI is `http://autosar.org/3.0.2`.

Notes:
This option can be used with the AUTOSAR30 OS support.

Example:
To use `http://namespace` as the namespace URI when generating the OS configuration file, use:

```
--os-xml-namespace=http://namespace
```

3.65 --output

Description:

Direct all generated output files whose names match pattern <PAT> (which can include wild cards) to folder <FLDR>.

Name:

--output

Parameter:

The option takes one parameter that typically consists of the pattern <PAT> and the folder <FLDR>. The specification of <PAT> must be enclosed in square brackets and precedes <FLDR>, for example `--output=[*.c]Source`.

If either <PAT> or <FLDR> are omitted then this option is ignored except for the value "`check_only`" which, when present, causes the RTE generator to run, display detected errors and discard all output other than the error log.

Default:

Files are generated in the current folder.

Notes:

For further details on the use of the `--output` option including how multiple options are parsed see Section 2.2.1.

Example:

To redirect all C files to `folderA` and all other files to `folderB` use the following two options in sequence:

```
--output=[*.c]folderA --output=[*]folderB
```

To log errors to a file and discard all other output use:

```
--error-report=file --output=check_only
```

3.66 --period

Description:

Declare the period at which Rte_MainFunction will be called

Name:

--period

Parameter:

This option takes a single parameter, <SEC> that specifies the time between invocations of Rte_MainFunction in seconds.

Default:

0.01 (10ms period)

Notes:

The value of <SEC> must be chosen such that all Minimum Start Intervals are integral multiples of <SEC>. For maximum efficiency, choose Greatest Common Divisor of the Minimum Start Intervals in the ECU Extract. If no Minimum Start Intervals are used then this option is not relevant.

Example:

To notify RTA-RTE that the Rte_MainFunction API will be invoked every 50ms:
--period=0.05

3.67 `--preferred-intra-core-protection-scheme`

Description:

Select the preferred scheme for the implementation of intra-core concurrency protection in RTE internal code.

Name:

`--preferred-intra-core-protection-scheme`

Parameter:

This option takes a single parameter which names the intra-core concurrency protection strategy that is preferred for RTE internal code. Supported schemes are `per-core-resources`, where an RTE internal OS resource for each processor core is locked, `os-interrupt-blocking`, where OS interrupts are suspended and `all-interrupt-blocking` where all interrupts are suspended.

Default:

`per-core-resources`

Notes:

The scheme selected with this option applies to RTE internal code where the accesses being protected are all from execution contexts on the same processor core and where the choice of scheme is not limited due to there being accesses from particular execution contexts (e.g. interrupt contexts) or due to the actions of the code being protected.

This option has no effect on the APIs implementing `ExclusiveAreas` declared in software components.

Note that at the time of writing this option only applies to a small subset of the generated code.

Example:

To prefer the use of OS interrupt blocking within RTE internal code:
`--preferred-intra-core-protection-scheme=os-interrupt-blocking`

With this setting OS interrupt blocking will be used where possible but there may be some places where this is not possible. For example where there is access via a BSW interrupt entity from a category 1 interrupt, all interrupt blocking will be used. On the other hand, where the protected code calls OS APIs such as `ActivateTask` the locking of per-core OS resources will be used, since it is not valid to call such APIs with interrupts disabled.

3.68 `--protection-threshold-copy-bytes`

Description:

Tune the amount of data that can be copied in a critical section when concurrency protection is needed.

Name:

`--protection-threshold-copy-bytes`

Parameter:

This option takes a single parameter: an integer expressing the threshold number of bytes that can be copied in one critical section. Supported values:

- "`0`": Batch all copy operations needing concurrency protection in a single critical section. May reduce latency.
- "`1`": Each copy operation needing concurrency protection is placed in its own critical section. May reduce jitter.
- *values larger than 1*: apply algorithm as per *notes* below.

Default:

"`0`"

Notes:

For thresholds greater than 1, RTA-RTE batches copy operations requiring concurrency protection:

- When there are copies to be made that require concurrency protection, RTA-RTE enters a critical section (e.g. by `GetResource`).
- RTA-RTE holds the critical section open, potentially across multiple copy operations, while there are still items to copy and the cumulative work performed (as defined below) within the critical section does not exceed the threshold.
- RTA-RTE releases the critical section.
- If not all the copy operations are complete, RTA-RTE repeats from item 1 until all the items are copied.

```
Work Performed = ( Number of bytes of data copied ) +
( Estimate of equivalent work, e.g. arithmetic operations )
```

Example:

To enable a separate lock around each copy action: `--protection-threshold-copy-bytes=1`
To set protection threshold to 32 bytes: `--protection-threshold-copy-bytes=32`
In this example, if there several data items to copy totalling 60 bytes, RTA-RTE would release the critical section (e.g. by `ReleaseResource` followed by `GetResource`) once during the batch of copy operations.

3.69 `--quiet`

Description:

Control the text output.

Name:

`--quiet`

Parameter:

The option takes a single parameter; an integer specifying the level of output required. Valid values are:

0 Verbose.

1 Normal: Suppress certain plug-in information messages (default).

2 Quiet: No output other than RTA-RTE banner during normal operation.

3 Silent: output at all during normal operation.

Default:

Normal output (level 1).

Notes:

None.

Example:

To suppress all output during normal operation:

```
--quiet=3
```

3.70 --report

Description:

Enable output of XML report.

Name:

`--report`

Parameter:

This option takes a single parameter that specifies the the report name and, optionally, the report template file.

Default:

None.

Notes:

None.

Example:

To generate the `RteObjects` report with a template:

```
--report=[template.xml]RteObjects
```

To generate the same report but without a template:

```
--report=[null]RteObjects
```

3.71 `--rte`

Description:

Select “RTE” generation phase to generate an RTE for the specified ECU name.

Name:

`--rte`

Parameter:

This option takes a single parameter, `<ECUI>` that specifies the ECU instance or the ECU configuration (ECUC value collection in R4.0) for which RTE generation should occur. The `System` element must be referenced from the ECU configuration. For R4.0 the parameter can be “`auto`” which causes the RTE generator to search for and use the single `EcucValueCollection` present in the input.

Default:

N/A

Notes:

The ECU instance `<ECUI>` must be specified using an absolute instance reference. (See Section 4.2.4.)

Example:

To enable RTE generation for `/pkg/ecu`:

```
--rte=/pkg/ecu
```

To enable automatic search for the ECUC value collection and processing of the ECU extract (R4.0 only):

```
--rte=auto
```

3.72 `--samples`

Description:

Enable creation of SWC skeleton files.

Name:

`--samples`

Parameter:

The `--samples` option takes a single parameter that specifies the samples required.

swc : Create skeleton code files consisting of empty runnable-entity bodies for each runnable in the SWC. The generated files are named `Rte_<name>.c` where <name> is the SWC name.

memmap : Create skeletons of SWC-specific memory-mapping files. The generated files are named `<name>_MemMap.h` where <name> is the SWC name.

Default:

None.

Notes:

Generated samples overwrite existing files with the same name.

The generated samples are intended to be examples only, and should be adapted for the application and target hardware before use.

Example:

To enable sample generation for SWCs:

```
--samples=swc
```

3.73 `--strict-config-check`

Description:

Enable RTE validation of input OS configuration.

Name:

`--strict-config-check`

Parameter:

This option takes a single parameter, <P>, that specifies whether to enable ('on' or 'weak') or disable ('off') the strict configuration checks.

Default:

Disable check.

Notes:

RTA-RTE supports the AUTOSAR strict configuration check; when enabled the RTE configuration must not require any OS objects that are not already present in the input configuration. For example, all runnables must be mapped to existing tasks and, if necessary, use pre-declared OsEvents and ScheduleTable/alarms for triggering.

RTA-RTE also supports "weak" configuration checks — this changes the reported error to a warning, allowing the build to continue.

Example:

To enable strict configuration checks:

```
--strict-config-check=on
```

3.74 `--strict-initial-values-check`

Description:

Enable RTE validation of input OS configuration.

Name:

`--strict-initial-values-check`

Parameter:

This option takes a single parameter, <P>, that specifies the behavior when an uninitialized calprm is encountered.

1. info: to output an informational message
2. warn: to output a warning message
3. error: to output an error message
4. none: to disable the detection of uninitialized calprms

Default:

Error.

Notes:

Example:

To reduce the severity of an uninitialized calprm to a warning:

```
--strict-initial-values-check=warn
```

3.75 `--strict-unconnected-rport-check`

Description:

Permit unconnected RPorts.

Name:

`--strict-unconnected-rport-check`

Parameter:

This option takes one argument with the value ’`off`’, ’`warn`’ or ’`error`’, which specifies what RTA-RTE should do when it detects an unconnected R-Port.

Default:

If the option is not specified, it is an error to have unconnected R-Ports

Notes:

This is the AUTOSAR external configuration switch strictUnconnectedRPortCheck.

Example:

To allow RTA-RTE to accept an input model having unconnected require ports:

```
--strict-unconnected-rport-check=off
```

To enable the check for unconnected RPorts and raise a warning when such a port is encountered:

```
--strict-unconnected-rport-check=warn
```

3.76 `--sws`

Description:

Selection of backend processor.

Name:

`--sws`

Parameter:

The parameter specifies the appropriate AUTOSAR release.

Default:

If this option is not specified then RTA-RTE examines the input XML for an XML namespace and selects the appropriate backend processor automatically.

Notes:

The `--sws` option bypasses the namespace check and explicitly selects the back-end processor. The set of valid parameter values depends on the installed back-end processors; use the `--help` option for the RTA-RTE frontend to show the valid list.

Example:

To explicitly select the R4.0 backend processor:

```
--sws=4.0
```

When explicitly selecting the backend processor ensure that the input XML is conformant to the selected AUTOSAR release.

3.77 `--task-recurrence`

Description:

Set the recurrence for externally activated periodic tasks.

Name:

`--task-recurrence`

Parameter:

This option takes a single parameter that specifies the task/OsEvent name and the recurrence rate in seconds as a `task.event=seconds`” pair. The specification of an OsEvent name is optional; if omitted the recurrence rate applies to activations of the task. Time values are specified in seconds with a period as the decimal separator.

Default:

Create OS alarms and/or schedule table entries (or use existing ones) to implement periodic RTE events.

Notes:

This option **disables** RTA-RTE’s generation of OS alarms and/or schedule table entries for periodic events. Instead RTA-RTE uses the specified task recurrences to both derive internal scaling for generated code (e.g. to map an RTE event with 20ms period to a task with an explicit recurrence of 10ms) and to validate the mapping to detect erroneous cases (e.g. mapping an RTE event with period 10ms to a task with an explicit recurrence of 20ms).

When this option is used for any task then it must be used to specify recurrence rates for all tasks with periodic events since it disables RTA-RTE’s support for generating OS mechanisms to activate periodic events.

This option can be specified many times to provide recurrence rates for multiple tasks. Alternatively the parameter can be specified as a comma-separated list of task name/rate pairs.

Example:

To specify a recurrence rate of 20ms for `taskA`:

```
--task-recurrence taskA=0.02
```

To specify a recurrence rate of 10ms for OsEvent `ev1`:

```
--task-recurrence TaskA.ev1=0.01
```

3.78 `--template-path`

Description:

Selection location of RTE library templates.

Name:

`--template-path`

Parameter:

This option takes a single parameter which specifies the folder containing RTE library files.

If a relative path is used, it is interpreted relative to the RTA-RTE application executable.

Default:

Specified within RTE configuration INI file. If the template path is specified both in the INI configuration file and on the command-line the latter takes precedence.

Notes:

None

Example:

To select project-specific templates located in `folderA`:

```
--template-path=folderA
```

3.79 `--terminate-background-tasks`

Description:

Causes the generated RTE to terminate background tasks.

Name:

`--terminate-background-tasks`

Parameter:

This option takes no parameters.

When the option is supplied, RTA-RTE will generate task bodies for background tasks that end with a call to the `TerminateTask()` function.

When the option is not supplied, task bodies for background tasks will end with a call to `ChainTask()`, with the task's own ID as the argument.

Default:

N/A

Notes:

Normally, background tasks (`OsTasks` to which only background events are mapped) end with a call to `ChainTask()` to allow continuous execution.

When this option is used, background tasks will be terminated instead. This may be useful in cases where the continual execution of background events causes problems, or to enable utilization measurement in a lower-level execution context.

When this option is used, any background tasks will need to be activated explicitly (e.g. by a periodic timer interrupt) as they will no longer restart themselves on completion.

Example:

To cause RTA-RTE to terminate background tasks:

```
--terminate-backgound-tasks
```

3.80 `--test-license`

Description:

Display RTE license information.

Name:

`--test-license`

Parameter:

None

Default:

N/A

Notes:

Perform a license check and display the result.

Example:

To test the license in use:

```
s --test-license
```

3.81 `--text-value-spec-policy`

Description:

Adjust whether RTE writes symbols or numeric values for `TextValueSpecs`

Name:

`--text-value-spec-policy`

Parameter:

compumethod-resolution `TextValueSpec.value` is interpreted as a physical quantity to be looked up in a corresponding `TEXTTABLE` or `BITFIELD_TEXTTABLE` `CompuMethod`. The numerical equivalent is written to the generated code. If the value cannot be found RTA-RTE rejects the configuration with an error.

symbolic-pdav-always When used in a `PortDefinedArgumentValue`, `TextValueSpec.value` is treated as a symbol that is to be copied directly into the generated code. Compile-time definitions must be provided when compiling `Rte.c`.

Default:

`compumethod-resolution`

Notes:

N/A

Example:

`--text-value-spec-policy=symbolic-pdav-always`

3.82 `--toolchain-significant-len`

Description:

Specify number of significant characters in toolchain identifiers.

Name:

`--toolchain-significant-len`

Parameter:

This option takes a single parameter, <P>, that specifies the number of significant characters.

Default:

`31`

Notes:

Indicates to the RTE generator the number of significant characters checked by the toolchain. The RTE generator will then issue a warning if multiple RTE generated identifiers are not distinguishable within the specified number of characters.

For RTE generation phase, this option can be set both in the ECUC file and on the command-line. If specified in both places then RTA-RTE will use the command-line value.

Example:

To set the number of significant characters to 60:

```
--toolchain-significant-len=60
```

3.83 `--use-partition-sections`

Description:

Select whether or not partition-specific default memory sections should be used for partition-local objects without a section specified explicitly.

Name:

`--use-partition-sections`

Parameter:

This option takes a single parameter which enables ("on" or "1") or disables ("off" or "0") use of partition-specific memory sections by default for objects that are local to a partition and have no memory section specified. When enabled, the name of the EcuCPartition is used as the infix for the memory section name when one is configured, otherwise the name of the OsApplication is used.

Default:

Disabled ("off") for AUTOSAR compliance.

Notes:

None.

Example:

To enable the use of partition-specific default memory sections: `--use-partition-sections=on`

3.84 `--variability-also-bind`

Description:

Add a BindingTime that RTA-RTE will also try to honor.

Name:

`--variability-also-bind`

Parameter:

If used, this option must be given the parameter `PRE-COMPILE-TIME` which is the only supported value at this time.

Default:

Only instantiate variability with BindingTime `CODE-GENERATION-TIME`

Notes:

N/A

Example:

`--variability-also-bind=PRE-COMPILE-TIME`

3.85 `--version`

Description:

Display RTE generator version.

Name:

`--version`

Parameter:

None

Default:

N/A

Notes:

Print the RTA-RTE product and RTE generator version information on the standard output.

Example:

To display the RTA-RTE version: `--version`

3.86 `--vfb-trace`

Description:

Globally enable (or disable) the creation of VFB trace hooks in the generated RTE.

Name:

`--vfb-trace`

Parameter:

This option takes a single parameter that specifies whether generation of hook functions is enabled (“on” or “1”) or disabled (“off” or “0”).

Default:

Enabled (“on”)

Notes:

For the RTE generation phase, this option can be set both in the ECUC file and on the command-line. If specified in both places then RTA-RTE will use the command-line value.

Example:

To disable VFB trace hook generation from the command-line:

```
--vfb-trace=off
```

3.87 `--warn-directive`

Description:

Set the name (excluding the '#') of the C pre-processor directive used to issue warnings.

Name:

`--warn-directive`

Parameter:

The option takes a single parameter, <STR>, that specifies the directive name.

Default:

`#pragma message`

Notes:

None.

Example:

To select `#warn` as the warning directive used in generated RTE code:

```
--warn-directive=warn
```

3.88 `--warning-as-error`

Description:

Set whether warnings are treated as errors.

Name:

`--warning-as-error`

Parameter:

The option takes a single parameter that specifies whether to enable (‘1‘) or disable (‘0’) treatment of warnings as errors.

Default:

Treat warnings as warnings (‘0’)

Notes:

When enabled RTE treats any warning as an error and raises the error level appropriately. Note that the warning message text itself is not changed and thus may still refer to the error as a “warning”.

Note: If you wish to use this option along with `--disable-warning` then you must turn the specific message back to a warning using `--error-as-warning`.

Example:

To treat all warnings as errors but continue suppressing warning `W17-1023`:

```
--warning-as-error=1                    --error-as-warning=E17-1023
--disable-warning=W17-1023
```

3.89 `--xfrm-ignore-inplace`

Description:

Allows RTA-RTE to ignore the requirement for in-place transformation.

Name:

`--xfrm-ignore-inplace`

Parameter:

This option takes no parameters.

When the option is supplied, RTA-RTE will ignore the requirement for in-place transformation, so configurations containing `TransformationTechnologys` that use in-place buffering will be accepted.

When the option is not supplied, RTA-RTE will reject these configurations with error 2226.

Default:

N/A

Notes:

This option does not enable complete support for in-place transformation. When supplied, RTA-RTE will not raise error 2226 when a configuration contains in-place transformation, but the generated RTE code will continue to use out-of-place buffering and will call transformer functions using the out-of-place API format.

When this option is used, RTA-RTE will raise warning 3062 for each in-place transformer in the configuration.

Example:

To ignore the requirement for in-place transformation and generate with out-of-place buffering:

```
--xfrm-ignore-inplace
```

4 Configuration

RTA-RTE reads configuration files in the form of AUTOSAR XML Description fragments, i.e. XML files containing a representation of all or part of an AUTOSAR model. Files provided as AUTOSAR XML Description fragments must declare that they use the AUTOSAR 4.x namespace and must conform with an AUTOSAR XML schema.

4.1 Supported namespace and schema versions

RTA-RTE interprets AUTOSAR XML Description Fragments that use the AUTOSAR 4.x namespace (`http://autosar.org/schema/r4.0`), and validates all files declaring that namespace against the schema for AUTOSAR revision 4.4.0 (`AUTOSAR_00046.xsd`).

RTA-RTE will reject a configuration if it contains any AUTOSAR XML Description Fragment that does not declare `http://autosar.org/schema/r4.0` as the default *namespace.*

AUTOSAR XML Description Fragments provided to the RTE generator may reference different schema versions. RTA-RTE ignores the `schemaLocation` attribute and will always validate input files against the supported AUTOSAR 4.4.0 schema, regardless of the schema version that is referenced in the file itself.

AUTOSAR XML schemas are backwards-compatible within the same namespace name, so an input file conforming to an earlier version of the AUTOSAR schema will validate successfully against a later schema version.

RTA-RTE will reject a configuration if it contains any input file that does not validate against namespace `http://autosar.org/schema/r4.0` according to the AUTOSAR 4.4.0 schema.

An RTE generator requires only a subset of the information that can be expressed by elements defined by the AUTOSAR XML schema. Elements present in the input to RTA-RTE that are not required for RTE configuration are ignored as long as the definitions are compliant with the input file's associated AUTOSAR schema.

4.1.1 Overriding the validation schema

For development purposes, it is sometimes necessary to override the schema used for validation of AUTOSAR XML Description files. For example, a new version of AUTOSAR may introduce elements that cannot be validated against RTA-RTE's supported schema. Providing that these elements are not relevant for the RTE generation, the validation schema can be overridden to allow these input files to be validated successfully by RTA-RTE.

Overriding the schema used for validation may cause RTA-RTE to raise unexpected errors, or may cause errors in the generated RTE source code. The schema must only be overridden during the development stage of a project. The code produced using an overridden validation schema must never be used in production.

To override the schema, you must edit the `RTEGen.ini` file to supply the location of the

new schema file. The RTEGen.ini file is located in the same folder as the RTEGen.exe executable - normally this will be C:\ETAS\RTA-RTE\bin. The path to the schema file is supplied using the key ValidateARXMLAgainstSchema=<path> in the [Options] section. This can be supplied either as an absolute path, or as a relative path in which case the base location is the folder containing the RTEGen.exe executable.

```
[Options]
ValidateARXMLAgainstSchema=c:\schemas\schema_file.xsd
```

RTA-RTE will raise warning 2996 whenever the validation schema is overridden.

When the overriding schema extends the set of reference targets for an XML element, RTA-RTE may reject the configuration since its internal validation engine is not aware of the change. Where the reference element is not relevant to RTE generation, the additional destination elements can be declared in the RTEGen.ini file.

For example to allow <PDU-REF> elements to additionally refer to <GENERAL-PURPOSE-I-PDU> and <J-1939-DCM-I-PDU> elements:

```
[ReferenceValidationTags]
PDU-REF=GENERAL-PURPOSE-I-PDU,J-1939-DCM-I-PDU
```

When the new schema includes changes relevant to the RTE configuration the effect on RTA-RTE is undefined and may include failure of the generator and/or generated code. The schema must only be overridden during the development stage of a project. The code produced using an overridden validation schema must never be used in production.

4.2 References

Most elements within the input are named using the <SHORT-NAME> element so that they can be referenced by other elements.

The short name is used to reference an element from another element. The short name should be a valid C identifier—i.e. consist only of the characters '_', 'A-Z', 'a-z' or '0-9' (but not start with a '0-9').

An element within an AUTOSAR configuration that has a short-name defines a namespace[1]. All immediate descendant elements that are also named must have unique names. This mechanism means that even if two objects contained within different parent objects have the same name it is possible to uniquely identify an individual object using a reference that includes the name of the parent objects.

An absolute reference consists of one or more element short names separated by the '/' character and preceded by a '/' character. The reference string forms a path, much like a file-system path, that can be used to unambiguously locate the target of the reference. References may also be specified relative to a defined base package.

[1]This should not be confused with an XML namespace.

A reference never includes a trailing '/' character.

The following subsections provide more information on absolute references, relative references and instance references.

4.2.1 Absolute

An absolute reference starts with the '/' character and unambiguously identifies the target.

For example, consider an SWC type `swcA` within package `A`. An absolute reference to the SWC type would be `/A/swcA`.

When following an absolute reference, RTA-RTE always starts searching at the top-most package level. Thus the first element of an absolute reference must be the short-name of an <AR-PACKAGE> element.

4.2.2 Relative

A relative reference can be distinguished from an absolute reference because the latter always start with the '/' character, whereas the former does not. A relative reference can only be understood in the context of a given base package.

Each AUTOSAR package may optionally define *reference bases* for other AUTOSAR packages whose objects are referred to from objects within the package. Each reference base defines the prefix to be used for relative references that are associated with the reference base. For example, assume a package defines the following reference base:

```
<REFERENCE-BASE>
  <SHORT-LABEL>types</SHORT-LABEL>
  <IS-DEFAULT>false</IS-DEFAULT>
  <PACKAGE-REF DEST='AR-PACKAGE'>/autosar_types</PACKAGE-REF>
</REFERENCE-BASE>
```

Subsequently within the package relative references can be used that are associated with base "`types`", for example, the relative reference within the package that defines reference base "`types`" above:

```
<TYPE-TREF BASE='types'>my_type</TYPE-TREF>
```

This is equivalent to the absolute reference:

```
<TYPE-TREF>/autosar_types/my_type</TYPE-TREF>
```

At most one reference base can be marked as the default for the package. The default reference base is used when a relative reference does not explicitly define the associated base, e.g.:

```
<TYPE-TREF>my_type</TYPE-TREF>
```

4.2.3 Instance

An instance reference is a collection of absolute references that together define an instance. When resolving an instance reference RTA-RTE resolves each absolute reference in turn until all encapsulated references have been processed.

The set of encapsulated references is dependent on context, for example an instance reference from a “data received” event to a data element contains both a reference to the required port and a reference to the data element within the interface categorizing the port, for example:

```
<DATA-ELEMENT-IREF>
  <R-PORT-PROTOTYPE-REF>...
  <DATA-ELEMENT-PROTOTYPE-REF>...
</DATA-ELEMENT-IREF>
```

In contrast, a SWC instance reference contains an absolute reference to the top-level composition, zero or more component prototype references (each of which references one level of the composition hierarchy when nested compositions are used) and a final target component prototype reference:

```
<COMPONENT-IREF>
  <SOFTWARE-COMPOSITION-REF>...
  <COMPONENT-PROTOTYPE-REF>...
  <TARGET-COMPONENT-PROTOTYPE-REF>...
</COMPONENT-IREF>
```

The set of references necessary for each particular type of instance reference is specified whenever relevant in the following sections.

4.2.4 Referencing an ECU Instance

The `--rte` command line argument accepts a single argument identifying the starting point of the RTE configuration. Usually this should be set to `auto`.

In some circumstances, e.g. if the supplied model is not an ECU Extract, then it is necessary to indicate a suitable starting point for the RTE configuration. In this case, the argument value can be a reference to an `<ECU-INSTANCE>` or `<ECUC-VALUE-COLLECTION>` that references a system extract.

The reference must satisfy the following constraints:

- If referencing an `<ECU-INSTANCE>` then the reference must be an absolute reference to an ECU instance element and there must be exactly one ECU configuration element in the input.
- If referencing an `<ECUC-VALUE-COLLECTION>` then the ECU configuration must contain an `<ECU-EXTRACT-REF>` that identifies the associated `<SYSTEM>` element as well as references to the Rte, OS and COM configurations.

  If the `<SYSTEM>` element has category “ECU_EXTRACT” then all component prototypes within the root software composition are used for RTE generation and any

SwcToEcuMappings present in the System element are ignored and can be omitted if desired.

Otherwise the <SYSTEM> element must contain exactly one <SWC-TO-ECU-MAPPING> element that references the associated <ECU-INSTANCE>.

The --rte option can be passed a parameter of "auto" to cause the RTE generator to search for and use the single EcucValueCollection present in the input.

4.3 Packages

The configuration of software-components, types, ECUs, etc. within an AUTOSAR configuration is contained within one or more AUTOSAR package elements.

A package element, specified using the <AR-PACKAGE> tag, can contain either one or more sub-packages or one or more elements. Since sub-packages can contain AUTOSAR packages the relationship is recursive and a hierarchical tree of packages—akin to a file system—can be constructed.

The <AUTOSAR> element is the XML root node and must be present in all input files. Within the <TOP-LEVEL-PACKAGES> node one or more AUTOSAR packages can be defined. Each AUTOSAR package defines either sub-packages or elements.

```
ar_package ::=
    <AR-PACKAGE>
    short_name
    ( elements )
    ( sub-packages)
    </AR-PACKAGE>
```

Each <SUB-PACKAGES> element defines one or more AUTOSAR packages. They contain AUTOSAR sub-packages, which can themselves define either elements or further <SUB-PACKAGES>.

```
sub_packages ::=
    <SUB-PACKAGES>
    + ar_package
    </SUB-PACKAGES>
```

An AUTOSAR package can contain both sub-packages and elements.

The + in the above XML indicates there can be one or more of the items appearing next to the +.

4.3.1 Package Merging

An AUTOSAR package is an open set and therefore the RTA-RTE RTE generator "merges" elements (SWC types, interfaces, etc) and containers (OS tasks, runnable entity mappings, etc) within packages with the same name when the XML files are loaded.

The merge rules implemented by RTA-RTE depend on the file type:

ECU Configuration —The ECU Configuration defines the OS and RTE configuration for an ECU in terms of multiple <CONTAINER> elements each of which describes one aspect of the configuration such as a task or runnable entity mapping.

Containers can be split across multiple files and those with the same name at the same location within the package hierarchy merged when loaded.

Model —The model includes all SWC type definitions, system elements, etc.

The `atpSplitable` pattern is specified by AUTOSAR on certain aggregations. Not all of these are supported by RTA-RTE; in most cases two elements with the same path are regarded as a collision. However RTA-RTE does support merging for the following application configuration elements:

- <MODULE-CONFIGURATION>
- <ECU-CONFIGURATION>
- <SYSTEM>
- <COMPOSITION-TYPE>

Note that subelements of splitable model elements, such as SWC mappings, cannot typically be split and must be uniquely named.

The merging of elements and containers is illustrated in Figure 4.1.

4.4 Software Components

Atomic software component types are defined using one of the following tags:

- <APPLICATION-SW-COMPONENT-TYPE> — component that is part of an AUTOSAR application.
- <SENSOR-ACTUATOR-SW-COMPONENT-TYPE> — A component that is part of an AUTOSAR application and accesses sensors and/or actuators.
- <SERVICE-SW-COMPONENT-TYPE>.
- <ECU-ABSTRACTION-COMPONENT-TYPE>
- <COMPLEX-DEVICE-DRIVER-SW-COMPONENT-TYPE>

Whichever tag is used to declare a software component type, the element defines two things: the component's type name (necessary so that it can be referenced) and the component's port prototypes.

```
component_type ::=
        ( <APPLICATION-SW-COMPONENT-TYPE>
        | <SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
        | <SERVICE-SW-COMPONENT-TYPE>
        | <ECU-ABSTRACTION-SW-COMPONENT-TYPE>
        | <COMPLEX-DEVICE-DRIVER-SW-COMPONENT-TYPE>
        | <NV-BLOCK-SW-COMPONENT-TYPE> )
```

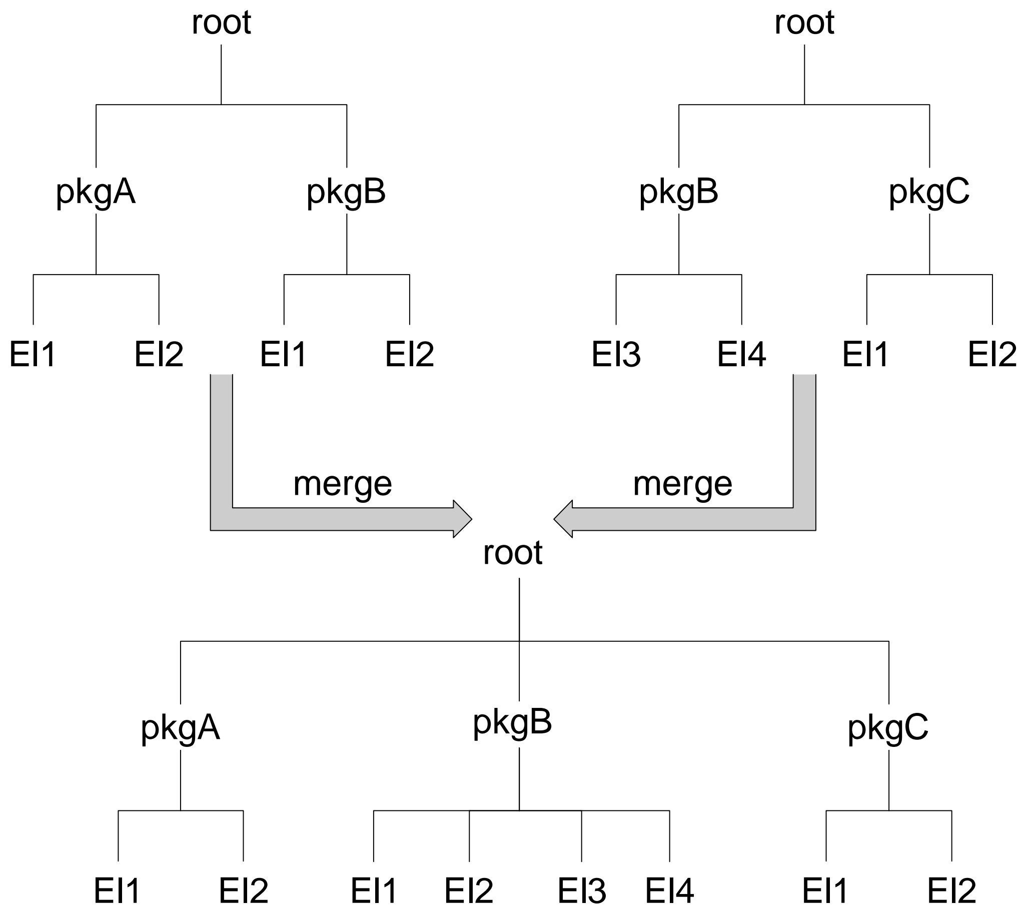


Figure 4.1: AUTOSAR Package Merge

```
short_name
port_prototypes
nv_block_descriptors
( </APPLICATION-SW-COMPONENT-TYPE>
    | </SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
    | </SERVICE-SW-COMPONENT-TYPE>
    | </ECU-ABSTRACTION-SW-COMPONENT-TYPE>
    | </COMPLEX-DEVICE-DRIVER-SW-COMPONENT-TYPE>
    | </NV-BLOCK-SW-COMPONENT-TYPE> )
```

The opening and closing tags for a component_type definition must match.

4.4.1 Port Prototypes

Port prototypes are defined using either the <P-PORT-PROTOTYPE> or <R-PORT-PROTOTYPE> element depending on whether the containing software-component type *provides* or *requires* the port. All port prototypes, whether they are required or provided, are encapsulated within a <PORTS> element:

```
port_prototypes ::=
```

```
<PORTS>
+ port_prototype
</PORTS>
```

A port prototype defines either a required port or a provided port; a port cannot be both required and provided.

```
port_prototype ::=
    ( r_port_prototype | p_port_prototype )
```

A port prototype is named and the name is used to reference instances of the port prototype once the containing software component has been instantiated.

```
r_port_prototype ::=
    <R-PORT-PROTOTYPE>
    short_name
    ( communication_specification )
    required_interface_reference
    </R-PORT-PROTOTYPE>

p_port_prototype ::=
    <P-PORT-PROTOTYPE>
    short_name
    ( communication_specification )
    provided_interface_reference
    </P-PORT-PROTOTYPE>
```

Each port prototype definition contains a reference to a categorizing interface, which can be a sender-receiver interface, client-server interface, a calibration interface, a mode switch interface or an Nv-data interface.

Multiple port prototypes, even if located in different software components, can reference the same interface.

```
required_interface_reference ::=
    <REQUIRED-INTERFACE-TREF>
    ref
    </REQUIRED-INTERFACE-TREF>

provided_interface_reference ::=
    <PROVIDED-INTERFACE-TREF>
    ref
    </PROVIDED-INTERFACE-TREF>
```

4.4.2 NV Blocks

AUTOSAR R4.0 introduced the <NV-BLOCK-SW-COMPONENT-TYPE> SWC type (Section 4.4). An NvBlockSwComponentType SWC can declare one or more <NV-BLOCK-DESCRIPTOR> elements that enable configuration and access to mirrors of data managed by the AUTOSAR NVRAM manager.

The NvBlockDescriptor elements are declared within an encapsulating <NV-BLOCK-DESCRIPTORS> element.

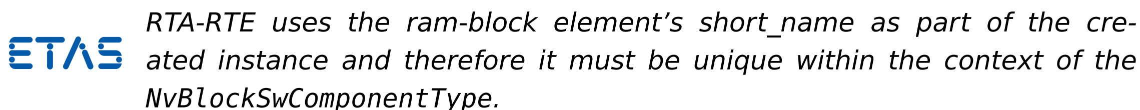
```
nv_block_descriptors ::=
        <NV-BLOCK-DESCRIPTORS>
        + nv_block_descriptor
        </NV-BLOCK-DESCRIPTORS>

nv_block_descriptor ::=
        <NV-BLOCK-DESCRIPTOR>
        short_name
        ( cs_ports )
        ( nvblock_data_mappings )
        ram_block
        ( rom_block )
        </NV-BLOCK-DESCRIPTOR>
```

Each NvBlockDescriptor element declares a ram block.

```
ram_block ::=
        <RAM-BLOCK>
        short_name
        ( sw_addr_method_ref )
        type_ref
        ( init_value )
        </RAM-BLOCK>
```

The <RAM-BLOCK> serves as the RAM mirror for the NVRAM manager's non-volatile data. The element is named and can therefore be referenced by other elements within the configuration.



RTA-RTE uses the ram-block element's short_name as part of the created instance and therefore it must be unique within the context of the NvBlockSwComponentType.

In addition to the ram-block, each NvBlockDescriptor element can optionally declare a rom-block, which can be used for initializing the ram-block.

```
rom_block ::=
        <ROM-BLOCK>
        short_name
        type_ref
        init_value
        </ROM-BLOCK>
```

The types used by the ram-block and rom-block must be compatible. RTA-RTE does not apply data conversion when initializing the ram-block from the rom-block.

The configuration of cs_ports and nvblock_data_mappings is considered in detail in Section 4.8.

4.4.3 Communication Specification

The communication_specification element (ComSpec) defines additional attributes of a port-prototype. For example:

- For a server, the queue length
- For a sender, whether or not transmission acknowledgment is enabled.

ComSpecs for Servers, ModeSwitchSenders and QueuedReceivers are mandatory. They must have a queue length greater than zero, and this is enforced by RTA-RTE.

```
communication_specification ::=
    ( provided_com_spec | required_com_spec )

provided_com_spec ::=
    <PROVIDED-COM-SPECS>
    ( server_com_spec
    | sender_com_spec
    | modemanager_com_spec
    | parameter_provide_com_spec )
    </PROVIDED-COM-SPECS>

required_com_spec ::=
    <REQUIRED-COM-SPECS>
    ( receiver_com_spec
    | client_com_spec
    | parameter_require_com_spec )
    </REQUIRED-COM-SPECS>
```

Sender

A data sender can be optionally configured to provide:

- An acknowledgment when a transmission is complete.
- An initial value to be used when the data is transmitted before it is written.
- Enabling (or disabling) data invalidation.

Transmission Acknowledgement

Acknowledgment is enabled using the PROVIDED-COM-SPECS element defined within the providing port prototype.

```
sender_com_spec ::=
    ( event_sender_com_spec | data_sender_com_spec )
```

Both event and data sender com specs define whether or not transmission acknowledgment is required and the data element to be acknowledged. In addition, a com spec element for a data sender can define whether or not the datum can be invalidated.

```
event_sender_com_spec ::=
    <QUEUED-SENDER-COM-SPEC>
    <DATA-ELEMENT-REF>ref</DATA-ELEMENT-REF>
    ( ack_request )
    </QUEUED-SENDER-COM-SPEC>
```

```
data_sender_com_spec ::=
    <NONQUEUED-SENDER-COM-SPEC>
    <DATA-ELEMENT-REF>ref</DATA-ELEMENT-REF>
    ( ack_request )
    ( invalidation_specification )
    ( init_value_specification )
    </NONQUEUED-SENDER-COM-SPEC>
```

Communication specifications for queued and non-queued ports both include a <DATA-ELEMENT-REF> reference that defines the data element to which this communication specification applies.

Both queued and non-queued communication specifications can optionally define one or more transmission acknowledgment request(s):

```
ack_request ::=
    <TRANSMISSION-ACKNOWLEDGE>
        <TIMEOUT>float</TIMEOUT>
    </TRANSMISSION-ACKNOWLEDGE>
```

A non-queued sender can optionally provide an initial value to be used when transmitting the data (e.g. in a network frame) before it has been updated by the application.

```
init_value_specification ::=
    <INIT-VALUE-REF>
        ref
    </INIT-VALUE-REF>
```

The <INIT-VALUE-REF> reference defines the constant to be used for the data element's initial value. The reference should identify the constant's value specification and not the constant specification element itself.

RTA-RTE uses AUTOSAR COM for inter-ECU communication. AUTOSAR COM supports initial values only for integer types and therefore RTA-RTE will raise an error if an initial value is specified for another type.

A non-queued data sender can optionally enable or disable data invalidation using the <CAN-INVALIDATE> element. If omitted the default is to disable data invalidation.

```
invalidation_specification ::=
    <CAN-INVALIDATE>( true | false )</CAN-INVALIDATE>
```

The "invalid value" used when a data item is invalidated is set within the type specification.

Mode Manager

A mode manager can be optionally configured to provide:

- An acknowledgment when a mode switch is complete.
- The length of the queue for mode switch requests.

Both the enabling of acknowledgment and the queue length are enabled using the <MODE-SWITCH-COM-SPEC> element defined within the providing port prototype's <PROVIDED-COM-SPEC>.

A ModeSwitchSenderComSpec must have a queue length greater than zero, and this is enforced by RTA-RTE.

```
modemanager_com_spec ::=
    <MODE-SWITCH-COM-SPEC>
    ( mode_ack_timeout )
    ( mode_switch_queuesize )
    </MODE-SWITCH-COM-SPEC>
```

Mode switch acknowledgment is enabled by the specification of a <TIMEOUT> element within the <MODE-SWITCHED-ACK>:

```
mode_ack_timeout ::=
    <MODE-SWITCHED-ACK>
    <TIMEOUT>time</TIMEOUT>
    </MODE-SWITCHED-ACK>
```

The time specified within the <MODE-SWITCHED-ACK> determines the timeout applied when the Rte_Feedback API is invoked; if no switch occurs within the specified timeout then runnable entities associated with the acknowledgment are activated.

A timeout specification of 0 enables acknowledgment without a timeout.

The length of the queue for mode switch requests is enabled by the specification of a <QUEUE-LENGTH> element within the <MODE-SWITCH-COM-SPEC> element:

```
mode_switch_queuesize ::=
    <QUEUE-LENGTH>int</QUEUE-LENGTH>
```

A queue length of zero is invalid.

Mode user

A mode user can request asynchronous mode switch using a <REQUIRED-COM-SPECS> element defined within the providing component prototype.

```
server_com_spec ::=
    <MODE-SWITCH-RECEIVER-COM-SPEC>
    support_async_modeswitch
    </MODE-SWITCH-RECEIVER-COM-SPEC>

support_async_modeswitch ::=
    <SUPPORTS-ASYNCHRONOUS-MODE-SWITCH>
    boolean
    </SUPPORTS-ASYNCHRONOUS-MODE-SWITCH>
```

All mode users must explicitly enable asynchronous mode switch for it to take effect. No warning is issued if a subset of mode users enable asynchronous mode switch.

Parameter Provide

A parameter provider com spec can be optionally configured to provide initial values for calibration parameters. A calibration com-spec contains a pair of values; the first is a reference to a constant containing the initial value and the second a reference to the associated calibration prototype:

```
parameter_provide_com_spec ::=
    <PARAMETER-PROVIDE-COM-SPEC>
    init_value
        <PARAMETER-REF>ref</PARAMETER-REF>
    </PARAMETER-PROVIDE-COM-SPEC>
```

The referenced calibration parameter must be defined within the interface categorizing the port. The referenced constant and the calibration parameter must have the same underlying type.

Parameter Require

A parameter requirer com spec can be optionally configured to provide initial values for calibration parameters for **unconnected** require ports.

Parameter require com specs are only used for unconnected RPorts.

A calibration com-spec contains a pair of values; the first is a reference to a constant containing the initial value and the second a reference to the associated calibration prototype:

```
parameter_require_com_spec ::=
    <PARAMETER-REQUIRE-COM-SPEC>
    init_value
        <PARAMETER-REF>ref</PARAMETER-REF>
    </PARAMETER-REQUIRE-COM-SPEC>
```

The referenced calibration parameter must be defined within the interface categorizing the port. The referenced constant and the calibration parameter must have the same underlying type.

Receiver

A receiver can use either 'data' (non-queued) or 'event' (queued) semantics. The <REQUIRED-COM-SPECS> element can contain a communication specification for either a 'data' or an 'event' receiver.

```
receiver_com_spec ::=
    ( event_receiver_com_spec
    | data_receiver_com_spec )
```

Event (Queued) Receivers

A 'QUEUED' receiver com_spec is used whenever a data element within the interface categorizing the port is specified as using queued reception.

```
event_receiver_com_spec ::=
    <QUEUED-RECEIVER-COM-SPEC>
    <DATA-ELEMENT-REF>ref</DATA-ELEMENT-REF>
    ( filter )
    ( queue_length )
    </QUEUED-RECEIVER-COM-SPEC>
```

Queue Length

A 'QUEUED' receiver can configure the length of the queue used to hold received data before it is processed. The queue length is set for each received data element using the using queue_length within the <EVENT-RECEIVER-COM-SPECS> or <QUEUED-RECEIVER-COM-SPECS> element.

A QueuedReceiverComSpec must have a queue length greater than zero, and this is enforced by RTA-RTE.

```
queue_length ::=
    <QUEUE-LENGTH>integer</QUEUE-LENGTH>
```

Data (Non-Queued) Receivers

A 'non-queued' receiver com_spec is used whenever a data element within the interface categorizing the port is specified as using non-queued reception.

```
data_receiver_com_spec ::=
    <UNQUEUED-RECEIVER-COM-SPEC>
    <DATA-ELEMENT-REF>ref</DATA-ELEMENT-REF>
    ( alive_timeout )
    ( filter )
    ( init_value_specification )
    ( invalid_data_handling )
    </UNQUEUED-RECEIVER-COM-SPEC>
```

For both the 'queued' and 'non-queued' receiver specifications the <DATA-ELEMENT-IREF> instance reference defines the data element to which this communication specification applies and requires a single context reference (the port prototype within the software-component type) and a target reference (the data element within the interface that categorizes the port prototype).

A 'non-queued' receiver can be optionally configured to define:

- The timeout used to verify that the sender is "alive".
- An initial value to be used when the data is transmitted before it is written.
- The method to be used to handle invalidated data. If invalidation is enabled, the RTE generator will respond to invalid data it encounters by using either the method of "keep" or that of "replace".

- A filter applied to received values by the generated RTE before they are forwarded to the receiver.

Alive Timeout

A 'non-queued' receiver can optionally configure the minimum acceptable inter-arrival period for signals received via COM.

```
alive_timeout ::=
    <ALIVE-TIMEOUT>time</ALIVE-TIMEOUT>
```

If a non-zero "alive timeout" is specified and the inter-arrival period drops below the specified value then any user-configured `DataReceiveErrorEvent` will be raised.

Alive timeout applies only to inter-ECU communication.

Initial Value

A 'non-queued' receiver can optionally provide an initial value to be used when the data is read before a value has been received from the sender. If omitted the default is to disable data invalidation.

```
init_value_specification ::=
    <INIT-VALUE-REF>ref</INIT-VALUE-REF>
```

The `<INIT-VALUE-REF>` reference defines the constant that defines the data element's initial value. The reference should identify a constant's value specification and not to the constant specification element itself.

RTA-RTE uses AUTOSAR COM for inter-ECU communication. AUTOSAR COM supports initial values only for integer types and therefore RTA-RTE will raise an error if an initial value is specified for another type.

Invalidation

The actions taken when a receiver receives invalid data can be set using the `<HANDLE-INVALID>` element. This can be either "keep" to retain the invalid value and pass it to the receiver or "replace" to substitute the data's initial value.

```
invalid_data_handling ::=
        <HANDLE-INVALID>(KEEP|REPLACE)</HANDLE-INVALID>
```

Filter

Finally a receiver can configure a filter that is applied to values before they are passed to the receiver.

```
filter ::=
    <FILTER>filter_condition</FILTER>
```

A wide range of filter conditions are available:

```
filter_condition ::=
    ( ALWAYS
    | MASKED-NEW-DIFFERS-MASKED-OLD
    | MASKED-NEW-DIFFERS-X
    | MASKED-NEW-EQUALS-MASKED-OLD
    | MASKED-NEW-EQUALS-X
    | NEVER
    | NEW-IS-DIFFERENT
    | NEW-IS-EQUAL
    | NEW-IS-GREATER
    | NEW-IS-GREATER-OR-EQUAL
    | NEW-IS-LESS
    | NEW-IS-LESS-OR-EQUAL
    | NEW-IS-OUTSIDE
    | NEW-IS-WITHIN
    | ONE-EVERY-N )
```

The available filter conditions are summarized in the following table.

Condition Description	XML

Masked new value differs from masked old value.	`<MASKED-NEW-DIFFERS-MASKED-OLD>`
	`  <MASK>int</MASK>`
	`</MASKED-NEW-DIFFERS-MASKED-OLD>`

Masked new value differs from X.	`<MASKED-NEW-DIFFERS-X>`
	`  <MASK>int</MASK>`
	`  <X>int</X>`
	`</MASKED-NEW-DIFFERS-X>`

Masked new value equals masked previous value.	`<MASKED-NEW-EQUALS-MASKED-OLD>`
	`  <MASK>int</MASK>`
	`</MASKED-NEW-EQUALS-MASKED-OLD>`

Masked new value equals X.	`<MASKED-NEW-EQUALS-X>`
	`  <MASK>int</MASK>`
	`  <X>int</X>`
	`</MASKED-NEW-EQUALS-X>`

Condition Description	XML
New value is different from previous value.	<NEW-IS-DIFFERENT> </NEW-IS-DIFFERENT>
New value is equal to previous value.	<NEW-IS-EQUAL> </NEW-IS-EQUAL>
New value is greater than previous.	<NEW-IS-GREATER> </NEW-IS-GREATER>
New value is greater than or equal to previous.	<NEW-IS-GREATER-OR-EQUAL> </NEW-IS-GREATER-OR-EQUAL>
New value is less than previous.	<NEW-IS-LESS> </NEW-IS-LESS>
New value is less than or equal to previous.	<NEW-IS-LESS-OR-EQUAL> </NEW-IS-LESS-OR-EQUAL>
New value is outside specified range.	<NEW-IS-OUTSIDE> <MIN>int</MIN> <MAX>int</MAX> </NEW-IS-OUTSIDE>
New value is within specified range.	<NEW-IS-WITHIN> <MIN>int</MIN> <MAX>int</MAX> </NEW-IS-WITHIN>

Table 4.1: Supported Signal Conditions for Value-based Filtering

Client

The “client” communication specification is a place-holder—it does not specify any information used by RTA-RTE.

Server

A non re-entrant server contains a queue used to hold client requests before they are processed by the server. The queue length is set for each server using the `<PROVIDED-COM-SPECS>` element defined within the providing component prototype.

4.5 AUTOSAR Types and Data Conversion

4.5.1 ApplicationPrimitiveDataType

An ApplicationPrimitiveDataType allows the modelling of a single data value relevant to a SWCT in terms of physical quantities, without the modeller having to decide on the implementation details of the type, such as the width, signedness and encoding. The choice of representation for the quantity can then be made later in the development cycle.

```
application_primitive_data_type ::=
    <APPLICATION-PRIMITIVE-DATA-TYPE>
        short_name
        <CATEGORY>VALUE</CATEGORY>
        <SW-DATA-DEF-PROPS>
            <SW-DATA-DEF-PROPS-VARIANTS>
                <SW-DATA-DEF-PROPS-CONDITIONAL>
                    <SW-CALIBRATION-ACCESS>sw_calibration_access</SW-
                        CALIBRATION-ACCESS>
                    ( <COMPU-METHOD-REF DEST='COMPU-METHOD'>ref</COMPU-
                        METHOD-REF> )
                    ( <UNIT-REF>ref</UNIT-REF> )
                </SW-DATA-DEF-PROPS-CONDITIONAL>
            </SW-DATA-DEF-PROPS-VARIANTS>
        </SW-DATA-DEF-PROPS>
    </APPLICATION-PRIMITIVE-DATA-TYPE>
```

The ShortName of an ApplicationPrimitiveDataType is only used to identify the model element. An ApplicationPrimitiveDataType does not result in a C type in the generated code for the RTE. Instead, every ApplicationDataType that is used in the model must, by the RTE generation time, be mapped to a compatible ImplementationDataType (see 4.5.15 DataTypeMappingSet) that will be used in the generated code to represent the physical quantity (or quantities, for complex ApplicationDataTypes, see below) using the chosen encoding.

Category must contain the literal text VALUE.

CompuMethodRef references a CompuMethod that describes how this ApplicationDataType relates to the physical world (CompuMethod of Category IDENTICAL or LINEAR). (See 4.5.11 CompuMethod).

UnitRef is optional because it is possible that the CompuMethod references a Unit. If the CompuMethod references a Unit then the ApplicationPrimitiveDataType must

- not reference any Unit, or
- reference the same Unit (same path), or
- reference an exactly similar Unit (different path to a Unit with same relevant properties).

When Interfaces containing ApplicationDataTypes are connected by a connector that references a PortInterfaceMapping then the Data Conversion feature is activated (see the User Guide) for an ApplicationPrimitiveDataType or for the ApplicationPrimitiveDataTypes within a complex ApplicationDataType.

4.5.2 ApplicationRecordDataType

An ApplicationRecordDataType is a compound data type describing a group of values in the physical world, analogous to a C struct.

```
application_record_data_type ::=
    <APPLICATION-RECORD-DATA-TYPE>
        short_name
        <CATEGORY>STRUCTURE</CATEGORY>
        <SW-DATA-DEF-PROPS>
            <SW-DATA-DEF-PROPS-VARIANTS>
                <SW-DATA-DEF-PROPS-CONDITIONAL>
                    <SW-CALIBRATION-ACCESS>sw_calibration_access</SW-
                        CALIBRATION-ACCESS>
                </SW-DATA-DEF-PROPS-CONDITIONAL>
            </SW-DATA-DEF-PROPS-VARIANTS>
        </SW-DATA-DEF-PROPS>
        <ELEMENTS>
        (
            <APPLICATION-RECORD-ELEMENT>
                short_name
                <CATEGORY>category</CATEGORY>
                <TYPE-TREF DEST='dest'>ref</TYPE-TREF>
            </APPLICATION-RECORD-ELEMENT>
        ... )
        </ELEMENTS>
    </APPLICATION-RECORD-DATA-TYPE>
```

As for ApplicationPrimitiveDataType, the ShortName is only used to identify the model element and no C type in the generated code results. Instead a DataTypeMappingSet (see 4.5.15) is required to specify a mapping to an ImplementationDataType of category STRUCTURE. Likewise the element ShortName is only used to identify the model element and does not influence any generated C code.

Category must contain the literal text STRUCTURE.

Each element can be an ApplicationPrimitiveDataType, ApplicationRecordDataType or ApplicationArrayDataType. The element Category and the reference Dest should match the class of the referenced data type.

4.5.3 ApplicationArrayDataType

An ApplicationArrayDataType is a compound data type describing a group of values in the physical world of the same type, analogous to a C array.

```
application_array_data_type ::=
    <APPLICATION-ARRAY-DATA-TYPE>
```

```
        short_name
        <CATEGORY>ARRAY</CATEGORY>
        <SW-DATA-DEF-PROPS>
            <SW-DATA-DEF-PROPS-VARIANTS>
                <SW-DATA-DEF-PROPS-CONDITIONAL>
                    <SW-CALIBRATION-ACCESS>sw_calibration_access</SW-
                        CALIBRATION-ACCESS>
                 </SW-DATA-DEF-PROPS-CONDITIONAL>
            </SW-DATA-DEF-PROPS-VARIANTS>
         </SW-DATA-DEF-PROPS>
         <ELEMENT>
             short_name
             <CATEGORY>category</CATEGORY>
             <TYPE-TREF DEST='dest'>ref</TYPE-TREF>
             <ARRAY-SIZE-SEMANTICS>FIXED-SIZE</ARRAY-SIZE-SEMANTICS>
             <MAX-NUMBER-OF-ELEMENTS>int</MAX-NUMBER-OF-ELEMENTS>
         </ELEMENT>
     </APPLICATION-ARRAY-DATA-TYPE>
```

As for ApplicationPrimitiveDataType, the ShortName is only used to identify the model element and no C type in the generated code results. Instead a DataTypeMappingSet (see 4.5.15) is required to specify a mapping to an `ImplementationDataType` of category ARRAY. Likewise the element ShortName is only used to identify the model element and does not influence any generated C code.

Category must contain the literal text ARRAY. RTA-RTE does not support dynamic arrays so ArraySizeSemantics must be the literal text FIXED-SIZE.

The element type can be an ApplicationPrimitiveDataType, ApplicationRecordDataType or ApplicationArrayDataType. The element Category and the reference Dest should match the class of the referenced data type.

4.5.4 ImplementationDataType — General

An `ImplementationDataType` describes a C type. Typically, an `ImplementationDataType` results in a `typedef` being written to the generated code (but see 4.5.5).

All `ImplementationDataTypes` contain a ShortName and a Category. The ShortName of an `ImplementationDataType` becomes the name of the C type in the generated code. The rules about what other sub-elements may be configured change according to the Category supplied.

```
implementation_data_type::=
    ( implementation_data_type_type_reference
    | implementation_data_type_value
    | implementation_data_type_array
    | implementation_data_type_structure )
```

4.5.5 TypeEmitter

In AUTOSAR 3.x and early 4.x, `ImplementationDataTypes` that reference a SwBaseType result in a `typedef` only if the corresponding `SwBaseType.nativeDeclaration` is not empty. For `ImplementationDataTypes` that are defined in external header files, the `nativeDeclaration` should be empty in order to suppress generation of a definition by RTA-RTE that might conflict with the externally-supplied definition.

From AUTOSAR 4.0.3, `ImplementationDataTypes` contain a `typeEmitter` attribute. If the attribute is present and its value is RTE then RTA-RTE will generate the corresponding `typedef` for the type. If the `typeEmitter` is present but has a value other than RTE, then RTA-RTE assumes that the type is defined somewhere in a header file outside of RTA-RTE's control and does not generate a `typedef`.

Additionally, in AUTOSAR 4.x, if the `typeEmitter` attribute is not present, then RTA-RTE uses the legacy method in which the presence or absence of `nativeDeclaration` is used to decide.

4.5.6 ImplementationDataType — Category TYPE_REFERENCE

An `ImplementationDataType` of Category TYPE_REFERENCE expresses a C type that is an alias for another C type.

```
<IMPLEMENTATION-DATA-TYPE>
    short_name
    <CATEGORY>TYPE_REFERENCE</CATEGORY>
    <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
            <SW-DATA-DEF-PROPS-CONDITIONAL>
                <SW-ADDR-METHOD-REF DEST="SW-ADDR-METHOD">ref</SW-ADDR-
                    METHOD-REF>
                <IMPLEMENTATION-DATA-TYPE-REF DEST='IMPLEMENTATION-DATA
                    -TYPE'>ref</IMPLEMENTATION-DATA-TYPE-REF>
                <INVALID-VALUE>int</INVALID-VALUE>
            </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
    <TYPE-EMITTER>RTE</TYPE-EMITTER>
</IMPLEMENTATION-DATA-TYPE>
```

If used in the RTE then it is expressed in the generated code as:

```
typedef type-specifier typedef-name;
```

The ShortName of `ImplementationDataType` becomes *typedef-name*.

The referenced `ImplementationDataType` represents the *type-specifier* in the typedef. This means that the ShortName of the referenced type becomes *type-specifier*.

The recommended way to define new `ImplementationDataTypes` that represent simple values is to set Category to TYPE_REFERENCE and reference a Platform Type in ImplementationDataTypeRef. The Platform Types are defined by AUTOSAR and help to

protect the model from differences between hardware targets. (See *AUTOSAR Specification of Platform Types*).

```
<IMPLEMENTATION-DATA-TYPE>
    <SHORT-NAME>myImplType</SHORT-NAME>
    <CATEGORY>TYPE_REFERENCE</CATEGORY>
    <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
            <SW-DATA-DEF-PROPS-CONDITIONAL>
                <IMPLEMENTATION-DATA-TYPE-REF DEST='IMPLEMENTATION-DATA-
                   TYPE'>/AUTOSAR_Platform/ImplementationDataTypes/uint16<
                   /IMPLEMENTATION-DATA-TYPE-REF>
            </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
    <TYPE-EMITTER>RTE</TYPE-EMITTER>
</IMPLEMENTATION-DATA-TYPE>
```

```
typedef uint16 myImplType;
```

Note that an `ImplementationDataTypeElement` of Category TYPE_REFERENCE has a different meaning. See 4.5.8 ImplementationDataType — Category ARRAY and 4.5.9 ImplementationDataType — Category STRUCTURE.

4.5.7 ImplementationDataType — Category VALUE

An `ImplementationDataType` of category VALUE describes a simple C data type.

```
<IMPLEMENTATION-DATA-TYPE>
    short_name
    <CATEGORY>VALUE</CATEGORY>
    <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
            <SW-DATA-DEF-PROPS-CONDITIONAL>
                <BASE-TYPE-REF DEST='SW-BASE-TYPE'>ref</BASE-TYPE-REF>
                <DATA-CONSTR-REF DEST='DATA-CONSTR'>ref</DATA-CONSTR-REF>
            </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
    <TYPE-EMITTER>RTE</TYPE-EMITTER>
</IMPLEMENTATION-DATA-TYPE>
```

If used in the RTE then the C type is expressed in the generated code as:

```
typedef type-specifier typedef-name;
```

Note: Normally you should use Category TYPE_REFERENCE for this purpose as types defined in that way are more easily portable between hardware targets.

The ShortName of `ImplementationDataType` becomes *typedef-name*.

BaseTypeRef references a SwBaseType that specifies the width of the type in bits and the encoding (which defines whether or not the type is signed). The SwBaseType's

nativeDeclaration contains native C that will be emitted as the *type-specifier* in the typedef. If the nativeDeclaration is blank or missing then no typedef is emitted. [rte_sws 7104 in, e.g. AUTOSAR_SWS_RTE R3.1.0]

In the case of an ImplementationDataTypeElement (See 4.5.8 ImplementationDataType — Category ARRAY and 4.5.9 ImplementationDataType — Category STRUCTURE) no typedef is emitted. An ImplementationDataTypeElement of Category VALUE is only rarely required (better to use TYPE_REFERENCE). If such an ImplementationDataTypeElement is used then its referenced SwBaseType must have a valid nativeDeclaration. If the nativeDeclaration is missing or blank then RTA-RTE rejects the configuration, because a member of the complex type has no type-specifier:

```
typedef struct {
    uint8 element1;          /* using a TYPE_REFERENCE to Platform
        Type uint8 */
    unsigned char element2; /* with nativeDeclaration "unsigned char
        ". */
    /*ERROR*/ element3;     /* using a VALUE referencing a
        SwBaseType without a nativeDeclaration */
} myStructType;
```

4.5.8 ImplementationDataType – Category ARRAY

An ImplementationDataType of category ARRAY describes a C array data type.

```
<IMPLEMENTATION-DATA-TYPE>
    short_name
    <CATEGORY>ARRAY</CATEGORY>
    <SUB-ELEMENTS>
        <IMPLEMENTATION-DATA-TYPE-ELEMENT>
            short_name
            <CATEGORY>TYPE_REFERENCE</CATEGORY>
            <ARRAY-SIZE>int</ARRAY-SIZE>
            <ARRAY-SIZE-SEMANTICS>FIXED-SIZE</ARRAY-SIZE-SEMANTICS>
            <SW-DATA-DEF-PROPS>
                <SW-DATA-DEF-PROPS-VARIANTS>
                    <SW-DATA-DEF-PROPS-CONDITIONAL>
                        <IMPLEMENTATION-DATA-TYPE-REF DEST='
                            IMPLEMENTATION-DATA-TYPE'>ref</
                            IMPLEMENTATION-DATA-TYPE-REF>
                    </SW-DATA-DEF-PROPS-CONDITIONAL>
                </SW-DATA-DEF-PROPS-VARIANTS>
            </SW-DATA-DEF-PROPS>
        </IMPLEMENTATION-DATA-TYPE-ELEMENT>
    </SUB-ELEMENTS>
    <TYPE-EMITTER>RTE</TYPE-EMITTER>
</IMPLEMENTATION-DATA-TYPE>
```

If used in the RTE then the C type is expressed in the generated code as:

```
typedef type-specifier typedef-name [ constant ];
```

The ShortName of the `ImplementationDataType` becomes *typedef-name*.

The ArraySize of the `ImplementationDataTypeElement` becomes *constant*.

The ShortName of the `ImplementationDataTypeElement` is not used except to identify the configuration element.

The `ImplementationDataTypeElement` specifies the type of the array element, which in turn can be of Category TYPE_REFERENCE, VALUE, STRUCTURE or ARRAY.

4.5.9 ImplementationDataType — Category STRUCTURE

An `ImplementationDataType` of category STRUCTURE describes a C `struct` data type.

```
<IMPLEMENTATION-DATA-TYPE>
    short_name
    <CATEGORY>STRUCTURE</CATEGORY>
    <SUB-ELEMENTS>
    (
        <IMPLEMENTATION-DATA-TYPE-ELEMENT>
            short_name
            <CATEGORY>TYPE_REFERENCE</CATEGORY>
            <SW-DATA-DEF-PROPS>
                <SW-DATA-DEF-PROPS-VARIANTS>
                    <SW-DATA-DEF-PROPS-CONDITIONAL>
                        <IMPLEMENTATION-DATA-TYPE-REF DEST='IMPLEMENTATION-
                            DATA-TYPE'>ref</IMPLEMENTATION-DATA-TYPE-REF>
                    </SW-DATA-DEF-PROPS-CONDITIONAL>
                </SW-DATA-DEF-PROPS-VARIANTS>
            </SW-DATA-DEF-PROPS>
        </IMPLEMENTATION-DATA-TYPE-ELEMENT>
    ... )
    </SUB-ELEMENTS>
    <TYPE-EMITTER>RTE</TYPE-EMITTER>
</IMPLEMENTATION-DATA-TYPE>
```

If used in the RTE then the C type is expressed in the generated code as:

```
typedef struct {
    member-type-specifier member-name;
    ...
} type-name
```

The ShortName of the `ImplementationDataType` becomes *type-name*.

The ShortName of the `ImplementationDataTypeElement` becomes *member-name*.

The `ImplementationDataTypeElement` specifies *member-type-specifier*, which in turn can be of Category TYPE_REFERENCE, VALUE, STRUCTURE or ARRAY.

4.5.10 ImplementationDataTypeElement

This is used inside an `ImplementationDataTypeElement` when specifying a complex data type (See 4.5.8 ImplementationDataType — Category ARRAY and 4.5.9 ImplementationDataType — Category STRUCTURE). The `ImplementationDataTypeElement` itself does not define a C type but is a type specifier either for an array element type or a structure member type.

`ImplementationDataTypeElements` are specified in nearly the same way as an `ImplementationDataType`. To configure an `ImplementationDataTypeElement`, refer to the section for the `ImplementationDataType` of the category you are interested in, but use the XML tag `IMPLEMENTATION-DATA-TYPE-ELEMENT`.

It is recommended that `ImplementationDataTypeElements` should be of category TYPE_REFERENCE, though STRUCTURE, ARRAY, and VALUE are supported too. When using VALUE there is a special caution that you cannot then reference a SwBaseType without nativeDeclaration. (See 4.5.7 ImplementationDataType — Category VALUE.)

4.5.11 CompuMethod — Category IDENTICAL

A CompuMethod of category `IDENTICAL` represents the linear data conversion $y = 0 + 1x/1$. Because it is linear it can be used in data conversion paths that also include LINEAR CompuMethods. On its own it does not result in any conversion code.

```
<COMPU-METHOD>
    short_name
    <CATEGORY>IDENTICAL</CATEGORY>
    <UNIT-REF DEST='UNIT'>ref</UNIT-REF>
</COMPU-METHOD>
```

The UnitRef is optional.

4.5.12 CompuMethod — Category LINEAR

A CompuMethod of category LINEAR describes a linear data conversion $y = num0 + num1 * x/denom$.

```
<COMPU-METHOD>
    short_name
    <CATEGORY>LINEAR</CATEGORY>
    <UNIT-REF DEST='UNIT'>ref</UNIT-REF>
    ( <COMPU-INTERNAL-TO-PHYS> | <COMPU-PHYS-TO-INTERNAL> )
        <COMPU-SCALES>
            <COMPU-SCALE>
                <COMPU-RATIONAL-COEFFS>
                    <COMPU-NUMERATOR>
                        <V>int</V>
                        <V>int</V>
                    </COMPU-NUMERATOR>
                    <COMPU-DENOMINATOR>
                        <V>int</V>
                    </COMPU-DENOMINATOR>
```

```
                </COMPU-RATIONAL-COEFFS>
            </COMPU-SCALE>
        </COMPU-SCALES>
    ( </COMPU-INTERNAL-TO-PHYS> | </COMPU-PHYS-TO-INTERNAL> )
</COMPU-METHOD>
```

The ShortName is used as a reference target.

The UnitRef is optional.

CompuInternalToPhys is used if x is the numerical value and y is the physical value.

CompuPhysToInternal is used if x is the physical value and y is the internal value.

It is permitted to specify both CompuInternalToPhys and CompuPhysToInternal. Each must be the inverse of the other. It is not necessary to specify both, as RTA-RTE can derive the inverse of a LINEAR CompuMethod as needed.

There must be exactly two V elements under `COMPU-NUMERATOR`. The first V element expresses $num0$ and the second V element expresses $num1$.

There must be exactly one V element under `COMPU-DENOMINATOR`. It expresses $denom$.

4.5.13 CompuMethod — Category RAT_FUNC

A CompuMethod of category `RAT_FUNC` is not supported by AUTOSAR R4.0. However if you configure a data path having the same or exactly similar CompuMethod at each end then RTA-RTE assumes that the CompuMethods cancel out and treats them as if they compose to the `IDENTICAL` transformation.

In cases where unsupported CompuMethod categories do not trivially match, RTA-RTE rejects the configuration.

4.5.14 CompuMethod — Category TEXTTABLE

A CompuMethod of category `TEXTTABLE` models a C enumerated type. It is not referenced by an `ApplicationPrimitiveDataType` like other CompuMethods but by an `ImplementationDataType`.

```
<COMPU-METHOD>
    short_name
    <CATEGORY>TEXTTABLE</CATEGORY>
    <COMPU-INTERNAL-TO-PHYS>
        <COMPU-SCALES>
        (
            <COMPU-SCALE>
                <LOWER-LIMIT>int</LOWER-LIMIT>
                <UPPER-LIMIT>int</UPPER-LIMIT>
                <COMPU-CONST>
                  <VT>identifier</VT>
                </COMPU-CONST>
            </COMPU-SCALE>
```

```
...)
            </COMPU-SCALES>
        </COMPU-INTERNAL-TO-PHYS>
    </COMPU-METHOD>
```

The CompuMethod may contain as many CompuScales as needed. For each CompuScale RTA-RTE generates code similar to the following:

```
#define symbol       ( ( type ) literal-constant )
```

LowerLimit and UpperLimit must be the same value, i.e. the CompuScale must express a point range. This value becomes *literal-constant*.

CompuConst must contain exactly one VT element. The contents of the VT element become *symbol*.

The *type* in the example is derived from the `ImplementationDataType` that references the CompuMethod.

4.5.15 DataTypeMappingSet

A `DataTypeMappingSet` maps `ApplicationDataTypes` used in a SWCT to `ImplementationDataTypes` with which those physical quantities should be represented in the generated code. It is referenced by a Software Component Type so it is possible that different SWCTs implement the same `ApplicationDataType` using different `ImplementationDataTypes`.

```
<DATA-TYPE-MAPPING-SET>
    short_name
    <DATA-TYPE-MAPS>
    (
        <DATA-TYPE-MAP>
        <APPLICATION-DATA-TYPE-REF DEST='APPLICATION-PRIMITIVE-DATA-TYPE'>
            ref</APPLICATION-DATA-TYPE-REF>
        <IMPLEMENTATION-DATA-TYPE-REF DEST='IMPLEMENTATION-DATA-TYPE'>ref</
            IMPLEMENTATION-DATA-TYPE-REF>
        </DATA-TYPE-MAP>
    ... )
    </DATA-TYPE-MAPS>
</DATA-TYPE-MAPPING-SET>
```

The `DataTypeMappingSet` can contain as many `DataTypeMap` elements as needed and indeed may contain mappings for `ApplicationDataTypes` not used by the referencing SWCT(s), allowing sharing of `DataTypeMappingSets` between SWCTs and between projects; a `DataTypeMap` is only considered if the `ApplicationDataType` is used directly in the model by a SWCT that references the `DataTypeMappingSet`.

The `ApplicationDataTypeRef` and `ImplementationDataTypeRef` reference the two types to be mapped together. The `ImplementationDataType` must be compatible with the `ApplicationDataType`. In particular, for primitive types the `ImplementationDataType` must have sufficient range to represent the range of

the ApplicationDataType, if specified, i.e. if it has DataConstrs. For structures, the ImplementationDataType must have the same number of members as the ApplicationDataType and the pairs of member types at each position (the member names need not be the same) must be compatible. For arrays the ImplementationDataType must have the same number of elements as the ApplicationDataType and the element types must be compatible.

4.5.16 TextTableMapping

A TextTableMapping causes RTA-RTE to generate data conversion code to transform discrete values between a Sender and Receiver or a Client and a Server.

```
<TEXT-TABLE-MAPPINGS>
    <TEXT-TABLE-MAPPING>
        <MAPPING-DIRECTION>BIDIRECTIONAL</MAPPING-DIRECTION>
        <VALUE-PAIRS>
        (
            <TEXT-TABLE-VALUE-PAIR>
                <FIRST-VALUE>int</FIRST-VALUE>
                <SECOND-VALUE>int</SECOND-VALUE>
            </TEXT-TABLE-VALUE-PAIR>
        ... )
        </VALUE-PAIRS>
    </TEXT-TABLE-MAPPING>
</TEXT-TABLE-MAPPINGS>
```

TextTableMappings appear within a DataPrototypeMapping. The DataPrototypeMapping contains the context of “First” and “Second” Data Prototype.

MappingDirection indicates whether the mapping is valid when data moves from First to Second Data Prototype or from Second To First Data Prototype, or whether it will work in both directions.

If the mapping of value pairs is 1:1 (i.e. no two FirstValue elements are alike and no two SecondValue elements are alike) then MappingDirection of BIDIRECTIONAL is recommended.

If the mapping is $n : 1$ (i.e. some two FirstValue elements are alike, then MappingDirection must be FIRST-TO-SECOND.

If the mapping is $1 : n$ (i.e. some two SecondValue elements are alike, then MappingDirection must be SECOND-TO-FIRST.

If the mapping is $n : m$ then it cannot be used by RTA-RTE and must be split into more than one mapping. RTA-RTE does not reject the configuration but the generated code will resolve the ambiguities of the $n : m$ mapping in an unspecified way.

4.5.17 Unit

Unit represents a Unit of measurement of some physical quantity. Data conversion is possible between data values typed by ApplicationDataTypes having linear Com-

puMethods (includes IDENTICAL) and having matching or compatible Units. Units are compatible if they reference the same or exactly similar PhysicalDimension elements.

```
<UNIT>
    short_name
    <FACTOR-SI-TO-UNIT>float</FACTOR-SI-TO-UNIT>
    <OFFSET-SI-TO-UNIT>float</OFFSET-SI-TO-UNIT>
    <PHYSICAL-DIMENSION-REF DEST='PHYSICAL-DIMENSION'>ref</PHYSICAL-
        DIMENSION-REF>
</UNIT>
```

ShortName is used as a reference target.

FactorSiToUnit is a floating-point decimal number expressing the number of units per SI unit.

OffsetSiToUnit is a floating-point decimal number expressing the offset to be added to convert from SI units to the specified unit.

4.5.18 Expression of Constant Values

Constant values are needed in the input model for the initial values of DataPrototypes and the reserved values of DataTypes used as the “invalid values” for DataPrototypes with invalidation enabled. These are modelled with ValueSpecifications, either directly where needed or within reusable ConstantSpecifications that can be referenced where needed.

4.5.19 NumericalValueSpecification

A NumericalValueSpecification provides a numeric constant value for direct use in the generated C. No data conversion is performed, even when the DataPrototype is typed by an ApplicationDataType; the value supplied is assumed to already be in the correct numerical representation for the mapped ImplementationDataType.

```
numerical_value_specification ::=
    <NUMERICAL-VALUE-SPECIFICATION>
        <VALUE>int</VALUE>
    </NUMERICAL-VALUE-SPECIFICATION>
```

4.5.20 ApplicationValueSpecification

An ApplicationValueSpecification provides a physical constant value in a given unit for a DataPrototype typed by an ApplicationPrimitiveDataType. Static (generation-time) data conversion is performed to obtain the corresponding numerical representation of the value, encoded for the mapped ImplementationDataType.

```
application_value_specification ::=
    <APPLICATION-VALUE-SPECIFICATION>
        <CATEGORY>VALUE</CATEGORY>
        <SW-VALUE-CONT>
            <UNIT-REF DEST='UNIT'>ref</UNIT-REF>
            <SW-VALUES-PHYS>
```

```
                <V>int</V>
            </SW-VALUES-PHYS>
        </SW-VALUE-CONT>
    </APPLICATION-VALUE-SPECIFICATION>
```

The referenced unit need not be the same as the unit for the ApplicationPrimitive-DataType of the DataPrototype since data conversion occurs as necessary. Of course the units need to reference the same physical dimension so that data conversion is possible.

4.5.21 RecordValueSpecification

A RecordValueSpecification provides a container for the constant values for each element of a structure, whether of ApplicationRecordDataType or of `ImplementationDataType` with category STRUCTURE. The value for each element is specified by an aggregated ValueSpecification of a class suitable for the data type of the corresponding element. The values are associated with the structure elements in declaration order.

```
record_value_specification ::=
    <RECORD-VALUE-SPECIFICATION>
        <FIELDS>
        (
            value_specification
        ...)
        </FIELDS>
    </RECORD-VALUE-SPECIFICATION>
```

4.5.22 ArrayValueSpecification

An ArrayValueSpecification provides a container for the constant values for each element of an array, whether or ApplicationArrayDataType or of `ImplementationDataType` with category ARRAY. The value for each element is specified by an aggregated Value-Specification of a class suitable for the array element data type.

```
array_value_specification ::=
    <ARRAY-VALUE-SPECIFICATION>
        <ELEMENTS>
        (
            value_specification
        ...)
        </ELEMENTS>
    </ARRAY-VALUE-SPECIFICATION>
```

For an array of elements of ApplicationPrimitiveDataType it is permitted to use a mix of NumericalValueSpecifications and ApplicationValueSpecifications.

4.5.23 ConstantSpecification

A ConstantSpecification provides a free-standing constant that can be referenced from anywhere that a ValueSpecification can be used.

```
constant_specification ::=
    <CONSTANT-SPECIFICATION>
        short_name
        <VALUE-SPEC>
            value_specification
        </VALUE-SPEC>
    </CONSTANT-SPECIFICATION>
```

4.5.24 ConstantReference

A ConstantReference can be used in place of a ValueSpecification to take the value from a free-standing ConstantSpecification.

```
constant_reference ::=
    <CONSTANT-REFERENCE>
      <CONSTANT-REF DEST='CONSTANT-SPECIFICATION'>ref</CONSTANT-REF>
    </CONSTANT-REFERENCE>
```

4.5.25 IncludedDataTypeSet

`IncludedDataTypeSets` can be used to add prefixes to the names of enumerated values in the generated C code (for example see 4.5.14).

```
included_data_type_set ::=
    <INCLUDED-DATA-TYPE-SET>
    <DATA-TYPE-REFS>
    data_type_ref+
    </DATA-TYPE-REFS>
    <LITERAL-PREFIX>prefix</LITERAL-PREFIX>
    </INCLUDED-DATA-TYPE-SET>
```

There may be one or more `data_type_ref` items each of which is a `<DATA-TYPE-REF>` element referencing an `ImplementationDataType` or an `ApplicationDataType`. The prefix `prefix` is added to the names of enumerated values generated from the referenced types.

When an `IncludedDataTypeSet` references an `ApplicationDataType`, RTA-RTE adds prefixes to enumerated value names generated from the `ApplicationDataType` and from any mapped `ImplementationDataType`.

If an `ImplementationDataType` or an `ApplicationDataType` is referenced by multiple `IncludedDataTypeSets` then each enumeration value name will be generated once for each `IncludedDataTypeSet` prefix.

If an `ImplementationDataType` or `ImplementationDataTypeElement` of Category TYPE_REFERENCE references a `CompuMethod` that results in enumerated value generation and the target of the TYPE_REFERENCE references a `CompuMethod` that also results in enumerated value generation then RTA-RTE will use the `CompuMethod` in the TYPE_REFERENCE rather than the target.

4.6 Interfaces

An interface element defines the data elements (sender-receiver), calibration elements or operations (client-server) that apply to the interface. A sender-receiver interface is declared using the <SENDER-RECEIVER-INTERFACE> element, a calibration interface using the <CALPRM-INTERFACE> and a client-server interface defined using the <CLIENT-SERVER-INTERFACE> element.

4.6.1 Sender-Receiver

A sender-receiver interface is defined using the <SENDER-RECEIVER-INTERFACE> element. The element must be named to enable it to be referenced from other elements, such as port prototypes.

```
sender_receiver_interface ::=
    <SENDER-RECEIVER-INTERFACE>
    short_name
    ( data_element_prototypes )
    ( mode_groups )
    </SENDER-RECEIVER-INTERFACE>
```

Data Elements

A sender-receiver interface element encapsulates zero or more data element definitions each defined using the <DATA-ELEMENT-PROTOTYPE> element.

```
data_element_prototypes ::=
    <DATA-ELEMENTS>
    +data_element
    </DATA-ELEMENTS>

data_element ::=
    <VARIABLE-DATA-PROTOTYPE>
    short_name
        ( sw_data_def_props_for_measurement )
    <TYPE-TREF>ref</TYPE-TREF>
    </VARIABLE-DATA-PROTOTYPE >
```

Each data element references a data type. The reference must refer to an AUTOSAR data type declared within the AUTOSAR XML configuration.

Mode Declarations

In addition to the (optional) data element prototypes, a sender-receiver interface can define zero or more mode declaration group prototypes within a <MODE-GROUPS> element.

```
mode_groups ::=
    <MODE-GROUPS>
    +mode_declaration_group_prototype
    </MODE-GROUPS>

mode_declaration_group_prototype ::=
```

```
        <MODE-DECLARATION-GROUP-PROTOTYPE>
        short_name
        <TYPE-TREF>ref</TYPE-TREF>
        </MODE-DECLARATION-GROUP-PROTOTYPE>
```

Each Mode Declaration Group Prototype element references a mode declaration group type. The reference must refer to an AUTOSAR <MODE-DECLARATION-GROUP> declared within the AUTOSAR XML configuration.

4.6.2 Nv-Data

A Nv-Data interface is defined using the <NV-DATA-INTERFACE> element. The element must be named to enable it to be referenced from other elements, such as port prototypes.

```
nv_data_interface ::=
    <NV-DATA-INTERFACE>
    short_name
    ( nv_data_prototypes )
    </NV-DATA-INTERFACE>
```

Nv-Data Elements

A Nv-Data interface element encapsulates zero or more Nv-data element definitions each defined using the <VARIABLE-DATA-PROTOTYPE> element.

```
nv_data_prototypes ::=
    <NV-DATAS>
    +variable_data_prototype
    </NV-DATAS>
```

Each variable data prototype references a data type. The reference must refer to an AUTOSAR data type declared within the AUTOSAR XML configuration.

4.6.3 Calibration

A calibration interface is defined using the <CALPRM-INTERFACE> element. The element must be named to enable it to be referenced from other elements, such as port prototypes.

```
calprm_interface ::=
    <CALPRM-INTERFACE>
    short_name
    ( calibration_element_prototypes )
    </CALPRM-INTERFACE>
```

Calibration Elements

A calibration interface element encapsulates zero or more calibration element prototypes each defined using the <DATA-ELEMENT-PROTOTYPE> element.

```
calibration_element_prototypes ::=
    <CALPRM-ELEMENTS>
```

```
        +calprm_element
        </CALPRM-ELEMENTS>

    calprm_element ::=
        <CALPRM-ELEMENT-PROTOTYPE>
        short_name
        ( sw_addr_method_ref )
        <TYPE-TREF>ref</TYPE-TREF>
        </CALPRM-ELEMENT-PROTOTYPE >
```

Each calibration element prototype references a data type. The reference must refer to an AUTOSAR data type declared within the AUTOSAR XML configuration.

A calibration element prototype contains an optional reference to a <SW-ADDR-METHOD> element:

```
    sw_addr_method_ref ::=
        <SW-DATA-DEF-PROPS>
        <SW-ADDR-METHOD-REF>ref</SW-ADDR-METHOD-REF>
        </SW-DATA-DEF-PROPS>
```

The <SW-ADDR-METHOD-REF> element must refer to an AUTOSAR <SW-ADDR-METHOD> element. RTA-RTE packs all elements within an interface that reference the same <SW-ADDR-METHOD> element into a structure to minimise the size of the required data-structures and to ensure that related elements are allocated to the same region of memory.

Calibration Data

RTA-RTE can allocate initial values for Calibration data. RTA-RTE creates a data structure for each calibration element group and when initial values are specified for all members of a group declares a suitable static initializer containing the values within `Rte.c`.

See Section 10.5 for details on labels and datastructures created by RTA-RTE.

4.6.4 Client-Server

A client-server interface follows a similar form to a sender-receiver interface with the exception that the data element definitions are replaced by operation definitions.

The client-server interface element must be named to enable it to be referenced from other elements, such as port prototypes.

```
    client_server_interface ::=
        <CLIENT-SERVER-INTERFACE>
        short_name
        operation_prototypes
        ( application_errors )
        </CLIENT-SERVER-INTERFACE>

    operation_prototypes ::=
        <OPERATIONS>
```

```
    +operation
    </OPERATIONS>
```

Operation Prototypes

Each operation within the client-server interface element is defined using the <OPERATION-PROTOTYPE> element that defines the operation name as well as encapsulating the definitions of the formal argument list.

```
operation ::=
    <OPERATION-PROTOTYPE>
    short_name
    <ARGUMENTS>
    +argument
    </ARGUMENTS>
    ( error_references )
    </OPERATION-PROTOTYPE>
```

An operation prototype must define one or more formal parameters using the <ARGUMENT-PROTOTYPE> element.

```
argument ::=
        <ARGUMENT-PROTOTYPE>
        short_name
        ( sw_data_def_props_for_measurement )
        <TYPE-REF>ref</TYPE-REF>
        direction
        </ARGUMENT-PROTOTYPE>
```

The specification within an argument prototype of sw_data_def_props is optional. If present it describes whether the argument is measureable. For more details see Section 4.7.

The short name of the <ARGUMENT-PROTOTYPE> defines the name of the formal parameter.

The type of the parameter is defined by the type reference and must refer to a type defined in the XML input. Each parameter also has a direction that specifies whether the parameter is an “in”, “in/out” or “out” parameter.

```
direction ::=
    <DIRECTION>(IN | INOUT | OUT)</DIRECTION>
```

An operation can be marked as ‘pure’ in which case invocation of the server runnable entity are not queued but are instead invoked by a direction function call from the client SWC. This is achievied by marking the runnable entity that implements the server as “CanBeInvokedConcurrently”, see Section 4.10.9.

A client-server interface can, optionally, define one or more application errors that can be returned (using the standard C return mechanism) from a runnable entity implementing the server.

```
application_errors ::=
    <POSSIBLE-ERRORS>
    +application_error
    </POSSIBLE-ERRORS>

application_error ::=
    <APPLICATION-ERROR>
    short_name
    <ERROR-CODE>int</ERROR-CODE>
    </APPLICATION-ERROR>
```

Declared application errors are defined in the software-component's application header file using the template:

```
RTE_E_<interface_name>_<error_name>
```

Once an application error has been defined it must be referenced from the associated operation prototype using an <POSSIBLE-ERROR-REF> element:

```
error_references ::=
        <POSSIBLE-ERROR-REFS>
        +error_reference
        </POSSIBLE-ERROR-REFS>

error_reference ::=
        <POSSIBLE-ERROR-REF>ref</POSSIBLE-ERROR-REF>
```

The data type for application errors is always Std_ReturnType and thus the return value for a runnable whose operation invoked event references one or more application errors is also Std_ReturnType.

4.7 Measurement

Measurement permits communication within the RTE to be monitored by an external tool.

4.7.1 Enabling Measurement

To enable measurement it must be both globally enabled in the RTE module configuration(see Section 4.19.2) and the individual item must be configured as measurable by specifying a <SW-DATA-DEF-PROPS> element:

```
sw_data_def_props_for_measurement ::=
        <SW-DATA-DEF-PROPS>
        <SW-CALIBRATION-ACCESS>READ-ONLY</SW-CALIBRATION-ACCESS>
        </SW-DATA-DEF-PROPS>
```

Measurement can be enabled for data elements (sender-receiver), arguments (client-server) and inter-runnable variables.

Data element Prototype —A data element within sender-receiver communication is enabled for measurement by specifying a <SW-DATA-DEF-PROPS> element either

in the data element prototype within an interface or, for primitive types, within data type's definition. If measurement is enabled/disabled in both places then the definition within the data element prototype takes precedence.

Argument Prototype —An argument within client-server communication is enabled for measurement by specifying a <SW-DATA-DEF-PROPS> element either in the argument prototype within an interface or, for primitive types, within data type's definition. If measurement is enabled/disabled in both places then the definition within the argument prototype takes precedence.

Inter-runnable variable —Measurement can be enabled for an inter-runnable variable (see Section 4.10.3) by specifying a <SW-DATA-DEF-PROPS> element either in the inter-runnable variable declaration within an internal behavior or, for primitive types, within relevant type's definition. If measurement is enabled/disabled in both places then the definition within the inter-runnable variable declaration prototype takes precedence.

4.7.2 RTA-RTE output

RTA-RTE outputs an XML file that describes each measured item's RTE-created buffer variable and associated configuration element (e.g. port and data item).

The XML output format is not defined by AUTOSAR and is therefore RTA-RTE specific.

The XML root element is <MEASURABLE-INFO>.

```
measurable ::=
        <MEASURABLE-INFO>
        ( measurable_data )
        ( measurable_arguments )
        ( measurable_irvs )
        </MEASURABLE-INFO>
```

Data Element Prototype

Measurable data element prototypes are described by the <MEASURABLE-DATUM> element.

```
measurable_data ::=
        <MEASURABLE-DATA>
        *measurable_datum
        </MEASURABLE-DATA>

measurable_datum ::=
        <MEASURABLE-DATUM>
        short-name
        <INSTANCE-DATUM>instance_ref</INSTANCE-DATUM>
        <SYMBOL>
        <NAME>string</NAME>
        <TYPE>string</TYPE>
        </SYMBOL>
        </MEASURABLE-DATA>
```

The <SYMBOL> element describes the RTE allocated name and data type of the variable to be measured. The <INSTANCE-DATUM> element references the associated component prototype, port and data element.

Argument Prototype

Measurable argument prototypes are described by the <MEASURABLE-ARG> element.

```
measurable_arguments ::=
        <MEASURABLE-ARGS>
        *measurable_arg
        </MEASURABLE-ARGS>

measurable_arg ::=
        <MEASURABLE-ARG>
        short-name
        <INSTANCE-ARG>instance_ref</INSTANCE-ARG>
        <SYMBOL>
        <NAME>string</NAME>
        <TYPE>string</TYPE>
        </SYMBOL>
        </MEASURABLE-ARG>
```

The <SYMBOL> element describes the RTE allocated name and data type of the variable to be measured. The <INSTANCE-ARG> element references the associated component prototype, port and data element.

Inter-runnable variables

Measurable inter-runnable variables are described by the <MEASURABLE-IRV> element.

```
measurable_arguments ::=
        <MEASURABLE-IRVS>
        *measurable_irv
        </MEASURABLE-IRVS>

measurable_irv ::=
        <MEASURABLE-IRV>
        short-name
        <INSTANCE-IRV>instance_ref</INSTANCE-IRV>
        <SYMBOL>
        <NAME>string</NAME>
        <TYPE>string</TYPE>
        </SYMBOL>
        </MEASURABLE-IRV>
```

The <SYMBOL> element describes the RTE allocated name and data type of the variable to be measured. The <INSTANCE-ARG> element references the associated component prototype and variable.

Example

The following XML describes a measured data element, argument and inter-runnable variable.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<MEASURABLE-INFO xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
    noNamespaceSchemaLocation="measurement.xsd">
  <MEASURABLE-DATA>
    <MEASURABLE-DATUM>
      <SHORT-NAME>SWCI0_pa_value</SHORT-NAME>
      <INSTANCE-DATUM>
        <SOFTWARE-COMPOSITION-REF>/TMeasurement/Compo</SOFTWARE-COMPOSITION
            -REF>
        <TARGET-COMPONENT-PROTOTYPE-REF>/TMeasurement/Compo/producer</
            TARGET-COMPONENT-PROTOTYPE-REF>
        <PORT-PROTOTYPE-REF>/TMeasurement/swc_tx/pa</PORT-PROTOTYPE-REF>
        <DATA-ELEMENT-PROTOTYPE-REF>/TMeasurement/if1/value</DATA-ELEMENT-
            PROTOTYPE-REF>
      </INSTANCE-DATUM>
      <SYMBOL>
        <NAME>Rte_RxBuf_0</NAME>
        <TYPE>VAR(SInt16, RTE_DATA)</TYPE>
      </SYMBOL>
    </MEASURABLE-DATUM>
  </MEASURABLE-DATA>
  <MEASURABLE-ARGS>
    <MEASURABLE-ARG>
      <SHORT-NAME>SWCI0_sc1_a</SHORT-NAME>
      <INSTANCE-ARG>
        <SOFTWARE-COMPOSITION-REF>/TMeasurement/Compo</SOFTWARE-COMPOSITION
            -REF>
        <TARGET-COMPONENT-PROTOTYPE-REF>/TMeasurement/Compo/client</TARGET-
            COMPONENT-PROTOTYPE-REF>
        <PORT-PROTOTYPE-REF>/TClientServer/swc_cli/sc1</PORT-PROTOTYPE-REF>
        <ARGUMENT-REF>/TClientServer/ifs/operation/a</ARGUMENT-REF>
      </INSTANCE-ARG>
      <SYMBOL>
        <NAME>Rte_MsBuf_0</NAME>
        <TYPE>VAR(SInt16, RTE_DATA)</TYPE>
      </SYMBOL>
    </MEASURABLE-ARG>
  </MEASURABLE-ARGS>
  <MEASURABLE-IRVS>
    <MEASURABLE-IRV>
      <SHORT-NAME>SWCI0_irvex1</SHORT-NAME>
      <INSTANCE-IRV>
        <SOFTWARE-COMPOSITION-REF>/TMeasurement/Compo</SOFTWARE-COMPOSITION
            -REF>
        <TARGET-COMPONENT-PROTOTYPE-REF>/TMeasurement/Compo/consumer</
            TARGET-COMPONENT-PROTOTYPE-REF>
        <VARIABLE-REF DEST="INTER-RUNNABLE-VARIABLE">/
            TInterRunnableVariables/IBswc_ex/irvex1</VARIABLE-REF>
      </INSTANCE-IRV>
      <SYMBOL>
        <NAME>Rte_Var_SWCI0_irvex1</NAME>
        <TYPE>VAR(SInt32, RTE_DATA)</TYPE>
```

```
      </SYMBOL>
    </MEASURABLE-IRV>
  </MEASURABLE-IRVS>
</MEASURABLE-INFO>
```

RTA-RTE includes an XML schema describing the XML file created by RTA-RTE, see {install-fldr}\Auxiliary.

4.8 NVRAM

AUTOSAR R4.0 introduced NvBlockSwComponentTypes for the configuration of RAM-based mirrors of non-volatile data managed by the NVRAM manager (see Section 4.4.2).

RTA-RTE supports two forms of access to the ram-blocks declared within NvBlockSwComponentTypes; from application SWCs using NvBlockDataMappings and from the NVRAM manager using ClientServerPorts.

4.8.1 Nv-Block Data Mappings

<NV-BLOCK-DATA-MAPPING> elements configure read and/or write access to ram-blocks (or to sub-elements of ram-blocks) declared by NvBlockDescriptor elements through ports categorized by <NV-DATA-INTERFACE>s.

```
nvblock_data_mappings ::=
        <NV-BLOCK-DATA-MAPPINGS>
        + nvblock_data_mapping
        </NV-BLOCK-DATA-MAPPINGS>

nvblock_data_mapping ::=
        <NV-BLOCK-DATA-MAPPING>
        nvram_element
        ( written_var )
        ( read_var )
        </NV-BLOCK-DATA-MAPPING>
```

A single <NV-BLOCK-DATA-MAPPING> can declare both read and write access to the same nvram_element.

NVRAM Element

The nvram_element references the ram-block within the NvBlockDescriptor via a <LOCAL-VARIABLE-REF>, <AUTOSAR-VARIABLE-IN-IMPL-DATATYPE> or <AUTOSAR-VARIABLE-IREF> element.

```
nvram_element ::=
        <NV-RAM-BLOCK-ELEMENT>
        ( local_variable_ref |
          autosar_variable_ref |
          autosar_variable_iref )
        </NV-RAM-BLOCK-ELEMENT>

local_variable_ref ::=
        <LOCAL-VARIABLE-REF>
```

```
        ref
        </LOCAL-VARIABLE-REF>

autosar_variable_ref ::=
        <AUTOSAR-VARIABLE-IN-IMPL-DATATYPE>
        <ROOT-VARIABLE-DATA-PROTOTYPE-REF>
        ref
        </ROOT-VARIABLE-DATA-PROTOTYPE-REF>
        ( impl_sub_element_mapping )
        </AUTOSAR-VARIABLE-IN-IMPL-DATATYPE>

autosar_variable_iref ::=
        <AUTOSAR-VARIABLE-IREF>
        <ROOT-VARIABLE-DATA-PROTOTYPE-REF>
        ref
        </ROOT-VARIABLE-DATA-PROTOTYPE-REF>
        ( appl_sub_element_mapping )
        </AUTOSAR-VARIABLE-IREF>
```

The `impl_sub_element_mapping` (or `appl_sub_element_mapping` as appropriate) is optional; if present it defines a mapping from the port's Nv-data element to a sub-element of the ram-block's data type.

```
impl_sub_element_mapping ::=
        ( <CONTEXT-DATA-PROTOTYPE-REFS>
          + context_ref
          </CONTEXT-DATA-PROTOTYPE-REFS> )
        <TARGET-DATA-PROTOTYPE-REF>
        ref
        </TARGET-DATA-PROTOTYPE-REF>

appl_sub_element_mapping ::=
        ( + context_ref )
        <TARGET-DATA-PROTOTYPE-REF>
        ref
        </TARGET-DATA-PROTOTYPE-REF>

context_ref ::=
        <CONTEXT-DATA-PROTOTYPE-REF>
        ref
        </CONTEXT-DATA-PROTOTYPE-REF>
```

If present the `impl_sub_element_mapping` (or `appl_sub_element_mapping`) defines zero or more context references and one target reference. Each reference defines one data type element and, when combined, identify the mapped sub-element of the ram-block.

ETAS *This release of RTA-RTE does not support mappings for individual elements of array types.*

Read Access by SWCs

Read access by an application SWC occurs through a port **required** by the SWC and **provided** by the NvBlockSwComponentType.

An NvBlockDataMapping declares read access to a ram-block through a <READ-NV-DATA> element (the element is "read" since the ram-block is provided by the NvBlockSwComponentType). The instance references within the <READ-NV-DATA> element must reference a **provide** port ON THE NvBlock SWC.

```
written_var ::=
        <READ-NV-DATA>
        <AUTOSAR-VARIABLE-IREF>
        iref
        </AUTOSAR-VARIABLE-IREF>
        </READ-NV-DATA>
```

The iref must reference a provide port and a variable data prototype within the port's interface. The port must be categorized by an NvDataInterface and the type of the variable data prototype must be compatible with the referenced ram-block type.

A single provider port on an NvBlockSwComponentType can have multiple connected require ports and thus a ram-block mirror can be read by multiple application SWCs. RTA-RTE uses interrupt blocking to prevent simultaneous writes corrupting reads.

It is not possible to use multiple mappings to associate more than one ram-block with the same provide port.

Write Access by SWCs

Write access by an application SWC occurs through a port **provided** by the SWC and **required** by the NvBlockSwComponentType.

An NvBlockDataMapping declares write access to a ram-block through an <WRITTEN-NV-DATA> element. The port instance reference within the <WRITTEN-NV-DATA> element must reference a require port.

```
read_var ::=
        <WRITTEN-NV-DATA>
        <AUTOSAR-VARIABLE-IREF>
        iref
        </AUTOSAR-VARIABLE-IREF>
        </WRITTEN-NV-DATA>
```

The <AUTOSAR-VARIABLE-IREF> element must reference a require port and a variable data prototype within the port's interface. The port must be categorized by an NvDataInterface and the type of the variable data prototype must be compatible with the referenced ram-block type.

A single require port can have multiple providers and thus a ram-block mirror can be updated by multiple application SWCs.

RTA-RTE does not support fan-out from a single require port to multiple ram-blocks.

To configure a single Write API generated for an application SWC to write to more than one ram-block within a NvBlockSwComponentType then multiple require ports (and multiple NvBlockDataMapping elements) must be configured and connected to the provide port.

4.8.2 Access by NVRAM manager

For each NvBlockDescriptor the RTE generator creates two API call-backs for the NVRAM manager. The Rte_GetMirror and Rte_SetMirror APIs do not require any specific configuration.

4.8.3 Client-Server Ports

Client-server port specifications within an NvBlockDescriptor (Section 4.4.2) cause RTA-RTE to create call-back API functions to be invoked by the NVRAM manager.

```
cs_ports ::=
        <CLIENT-SERVER-PORTS>
        + ( client_assignment | server_assignment )
        </CLIENT-SERVER-PORTS>
```

A client_assignment references a require port and a server_assignment references a provide port.

Client (require) ports

Each client_assignment that references a *require* port categorized by a client-server interface declares a call-back API function intended to be invoked by the NVRAM manager. The name of this function is based on the <ROLE> declared within the <ROLE-BASED-PORT-ASSIGNMENT> element:

```
client_assignment ::=
        <ROLE-BASED-PORT-ASSIGNMENT>
        <PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">
        ref
        </PORT-PROTOTYPE-REF>
        <ROLE>
        string
        </ROLE>
        </ROLE-BASED-PORT-ASSIGNMENT>
```

If multiple role_based_assignment share the same declared “role” then their actions are agregated by RTA-RTE.

RTA-RTE recognizes that standardized NvM roles NvMNotifyJobFinished and NvMNotifyInitBlock and establishes the call-back parameters according to the AUTOSAR RTE and NvM specifications. For other roles the parameters of the generated call-back function, if any, are taken from the first operation in the referenced require

port's interface.

It is not possible to reference just a single operation in the require port therefore the referenced interface should have a single operation.

For each declared role, RTA-RTE creates a function `Rte_<role>_<b>_<d>` that includes the servers referenced via the require ports in role based assignments that use role `<role>`. The generated function name includes the component prototype name <b> and the NvBlockDescriptor name <d>.

If the server invoked through the referenced require port is mapped to a task then RTA-RTE will generate code that writes the request to the server's queue and waits for the result. This will mean that the calling task, i.e. the NVRAM manager's task, will need to be declared as accessing the events. It is therefore recommended **not** to map the server to a task, i.e. omit the `TaskRef`, in which case RTA-RTE will use a direct function call to invoke the server.

Server (provide) ports

An `NvBlockSwComponentType` can also configure *provided* Client-Server ports. These ports are allow the C-based API of the MVRAM manager to be invoked via ports on application SWCs.

A `server_assignment` references a **provide** port categorized by a client-server interface:

```
server_assignment ::=
        <ROLE-BASED-PORT-ASSIGNMENT>
        <PORT-PROTOTYPE-REF DEST="P-PORT-PROTOTYPE">
        ref
        </PORT-PROTOTYPE-REF>
        <ROLE>
        string
        </ROLE>
        </ROLE-BASED-PORT-ASSIGNMENT>
```

For each **connected** provide port referenced from a `server_assignment` for which an `OperationInvokedEvent` is configured RTA-RTE will generate a call to NVRAM manager's C API.

The runnable entity referenced from the `OperationInvokedEvent` must be marked as concurrently invocable (see Section 4.10.9) and the runnable's symbol must be the name of the NvM API to be invoked. The parameters passed to the NvM API are formed from relevant port-defined arguments and the arguments to the relevant client-server operation. In addition the `OperationInvokedEvent` should **not** be mapped to a task to ensure direct function invocation of the NvM API from within the generated `Rte_Call` function.

4.9 AUTOSAR Modes

Modes within an AUTOSAR system are declared within a <MODE-DECLARATION-GROUP> element.

```
mode_group ::=
    <MODE-DECLARATION-GROUP>
    short_name
    <INITIAL-MODE-REF>ref</INITIAL-MODE-REF>
    <MODE-DECLARATIONS>
    +mode_declaration
    </MODE-DECLARATIONS>
    </MODE-DECLARATION-GROUP>

mode_declaration ::=
    <MODE-DECLARATION>
    short_name
    </MODE-DECLARATION>
```

The <MODE-DECLARATION-GROUP> element includes a reference to the group's initial mode. This must be a reference to a mode declared within the group.

ETAS *RTA-RTE supports up to 32 modes per mode-declaration group.*

4.10 Internal Behavior

The internal behavior description of a component defines the runnable entities present and is separate from the description of the component type.

The <INTERNAL-BEHAVIOR> element defines one or more runnable entities using the <RUNNABLE-ENTITY> element. Each runnable entity is named and the name is used to reference the entity from other elements in the XML.

The <INTERNAL-BEHAVIOR> also defines events associated with the runnable entities using the RTE-EVENTS element. Events can include time-triggers for the runnable entity in which case the period at which the entity is invoked is specified.

```
internal_behavior ::=
    <INTERNAL-BEHAVIOR>
    short_name
    swc_type_ref
    rte_events
    ( interrunnable_variables )
    ( exclusive_areas )
    ( nvram_mappings )
    ( per_instance_calprms )
    ( per_instance_memorys )
    ( port_api_options )
    ( runnable_entitys )
    ( shared_calprms )
    ( multiple_instances )
    </INTERNAL-BEHAVIOR>
```

The software component type reference indicates for which software component the internal behavior is defined. All software-component types use the same form of component reference:

```
swc_type_ref ::=
    <COMPONENT-REF>ref</COMPONENT-REF>
```

4.10.1 RTE Events

RTE Events indicate the action the RTE should take in response to certain stimuli.

```
rte_events ::=
    <EVENTS>
    *timing_event
    *data_received_event
    *data_receive_error_event
    *data_send_completed_event
    *asynchronous_server_call_returns_event
    *operation_invoked_event
    *mode_switch_event
        *mode_switched_ack_event
    </EVENTS>
```

It is permitted (though of limited utility) for a software-component to define no RTE Events. However in this case an empty <EVENTS> element must still be included.

An internal behavior for an NvBlockSwComponentType (see Section 4.4.2) can declare only OperationInvokedEvents and the associated runnable triggered when the event occurs. The activated runnable may only configure a <SYMBOL> and a <CAN-BE-INVOKED-CONCURRENTLY> flag – no other attributes are permitted.

Timing Event

A timing event defines the response to a time stimulus and is used to set the period for a time-triggered runnable entity.

A <TIMING-EVENT> element includes a reference to a runnable entity to start when the RTE event occurs.

```
timing_event ::=
    <TIMING-EVENT>
    short_name
    ( mode_dependency_list )
    <START-ON-EVENT-REF>ref</START-ON-EVENT-REF>
    <PERIOD>time</PERIOD>
    </TIMING-EVENT>
```

The <START-ON-EVENT-REF> reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The period of a TIMING-EVENT element must be expressed in seconds. No trailing text indicating the units is required or permitted. Fractions of a section can be expressed using floating point format with a period (".") as the decimal separator.

A Timing Event cannot be the target of a wait point.

A timing event can have an optional activation offset specified in the ECU configuration description. A different offset can be specified for each runnable entity instance (i.e. a different offset for each timing event in each SWC instance). For details, see the specification of `ActivationOffset` in Section 4.19.2.

Data Received Event

A data received event defines the response to reception of data or events on a receiver port.

A `<DATA-RECEIVED-EVENT>` element includes an instance reference to the data item received on the port and, optionally, a reference to a runnable entity to start when the RTE event occurs.

```
data_received_event ::=
    <DATA-RECEIVED-EVENT>
    short_name
    ( mode_dependency_list )
    ( <START-ON-EVENT-REF>ref</START-ON-EVENT-REF> )
    <DATA-IREF>instance_ref</DATA-IREF>
    </DATA-RECEIVED-EVENT>
```

The `<START-ON-EVENT-REF>` reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The `<DATA-IREF>` instance reference defines the data element to which this event applies and requires a single context reference (the port prototype) and a target reference (the data element within the interface that categorizes the port).

Data Receive Error Event

A data receive error event defines the runnable entity to activate when one of the following situations occurs:

- An invalidated data item is received (whether via a notification from COM or by intra-ECU communication implemented by RTA-RTE), and invalid reception handling for this data item is not set to REPLACE (in which case a data received event will occur instead).
- COM notifies RTA-RTE that a periodic data item has not been received within the configured time and has therefore timed out.

A <DATA-RECEIVE-ERROR-EVENT> element includes an instance reference to the data item received on the port and a reference to a runnable entity to start when the RTE event occurs.

```
data_receive_error_event ::=
    <DATA-RECEIVE-ERROR-EVENT>
    short_name
    ( mode_dependency_list )
    ( <START-ON-EVENT-REF>ref</START-ON-EVENT-REF> )
    <DATA-IREF>instance_ref</DATA-IREF>
    </DATA-RECEIVE-ERROR-EVENT>
```

The <START-ON-EVENT-REF> reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The <DATA-IREF> instance reference defines the data element to which this event applies and requires a single context reference (the port prototype) and a target reference (the data element within the interface that categorizes the port).

The activated runnable entity must use the Rte_IStatus API to read the error state—therefore the activated runnable should declare read access to the datum (see Section 4.10.10) to gain access to the status.

A Data Receive Error Event cannot be the target of a wait point.

Data Send Completed Event

A data send completed event defines the action taken when data or events have been transmitted on a provided sender-receiver port.

A <DATA-SEND-COMPLETED-EVENT> element includes a reference to the data send point and an optional reference to a runnable entity to start when the RTE event occurs.

```
data_send_completed_event ::=
    <DATA-SEND-COMPLETED-EVENT>
    short_name
    ( mode_dependency_list )
    ( <START-ON-EVENT-REF>ref</START-ON-EVENT-REF> )
    <EVENT-SOURCE-REF>ref</EVENT-SOURCE-REF>
    </DATA-SEND-COMPLETED-EVENT>
```

The <START-ON-EVENT-REF> reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The <EVENT-SOURCE-REF> reference is a simple reference that defines the data send point to which this event applies. The send point must be declared within the same

internal behavior as the RTE event.

Asynchronous Server Call Returns Event

An asynchronous server call returns event defines how the software component client port handles the return of an asynchronous server call.

An <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT> element includes an optional reference to a runnable entity to start when the RTE event occurs, and a reference to the <ASYNCHRONOUS-SERVER-CALL-POINT> within the runnable entity referenced.

```
asynchronous_server_call_returns_event ::=
    <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
    short_name
    ( mode_dependency_list )
    ( <START-ON-EVENT-REF>ref</START-ON-EVENT-REF> )
    <EVENT-SOURCE-REF>ref</EVENT-SOURCE-REF>
    </ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
```

The <START-ON-EVENT-REF> reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The <EVENT-SOURCE-REF> reference is a simple reference that defines the server call point to which this event applies. The call point must be declared within the same internal behavior as the RTE event.

Operation Invoked Event

An operation invoked event defines what the software component server port does in response to a server request.

An <OPERATION-INVOKED-EVENT> element includes a reference to the server operation that triggers the event and the runnable entity that acts as the server.

```
operation_invoked_event ::=
    <OPERATION-INVOKED-EVENT>
    short_name
    ( mode_dependency_list )
    <START-ON-EVENT-REF>ref</START-ON-EVENT-REF>
    <OPERATION-IREF>instance_ref</OPERATION-IREF>
    </OPERATION-INVOKED-EVENT>
```

The <START-ON-EVENT-REF> reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The <OPERATION-IREF> instance reference defines the operation to which this event applies and requires a single context reference (the port prototype) and a target reference (the operation within the interface that categorizes the port).

An Operation Invoked Event cannot be the target of a wait point.

Mode Switch Event

A Mode Switch event defines the actions of the software component when a mode manager switches modes using the `Rte_Switch` API.

A MODE-SWITCH-EVENT element includes a reference to the runnable entity that is triggered on either ENTRY to or EXIT from the referenced mode.

```
mode_switch_event ::=
    <MODE-SWITCH-EVENT>
    short_name
    ( mode_dependency_list )
    <START-ON-EVENT-REF>ref</START-ON-EVENT-REF>
    <ACTIVATION>(ENTRY|EXIT)</ACTIVATION>
    <MODE-IREF>instance_ref</MODE-IREF>
    </MODE-SWITCH-EVENT>
```

The `<START-ON-EVENT-REF>` reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The `<ACTIVATION>` element defines whether a Mode Switch Event applies to ENTRY to a mode or EXIT from a mode. It is not possible for a single mode switch event to apply to both entry and exit—if runnable activation is required for both then two Mode Switch Events should be defined.

The `<MODE-IREF>` within a `<MODE-SWITCH-EVENT>` element must contain two context references (respectively, the port prototype and the mode declaration group prototype within the interface categorizing the port prototype) and one target reference (the mode within the mode declaration group that types the declaration group prototype).

A Mode Switch Event cannot be the target of a wait point.

Mode Switched Acknowledge Event

A ModeSwitchedAck event defines the actions of the software component when a mode switch is complete.

A `<MODE-SWITCHED-ACK-EVENT>` element includes an optional reference to the runnable entity that is triggered after the mode switch defined by the elements reference to a `<MODE-SWITCH-POINT>` has completed.

```
mode_switched_ack_event ::=
    <MODE-SWITCHED-ACK-EVENT>
    short_name
    ( mode_dependency_list )
    <START-ON-EVENT-REF>ref</START-ON-EVENT-REF>
```

```
    <EVENT-SOURCE-REF>ref</EVENT-SOURCE-REF>
    </MODE-SWITCH-EVENT>
```

The <START-ON-EVENT-REF> reference is an absolute reference to a runnable entity declared within the same internal behavior as the RTE event. Activation of the referenced runnable entity by RTA-RTE in response to the RTE event can be disabled using one or more mode dependencies—see Section 4.10.2.

The <EVENT-SOURCE-REF> defines the Mode Switch Point to which the event applies.

A Mode Switched Ack Event occurs for the mode instance, not for a particular mode. Therefore the same RTE Event will occur after each mode switch irrespective of the requested mode.

A ModeSwitchedAckEvent that is the target of a wait point's trigger reference produces a blocking `Rte_Feedback` API call. It is not valid for a ModeSwitchedAckEvent element that is the target of a wait point's trigger reference to also trigger runnable entity activation.

4.10.2 Mode Dependency

An RTE event can define zero or more mode dependencies to control activation of a referenced runnable entity by RTA-RTE.

```
mode_dependency_list ::=
    <MODE-DEPENDENCY>
    <DEPENDENT-ON-MODE-IREFS>
    +mode_dependency
    </DEPENDENT-ON-MODE-IREFS>
    </MODE-DEPENDENCY>
```

Each mode dependency disables activation of the runnable entity by RTA-RTE when the specified mode is active.

```
mode_dependency ::=
    <DEPENDENT-ON-MODE-IREF>
    instance_ref
    </DEPENDENT-ON-MODE-IREF>
```

The <DEPENDENT-ON-MODE-IREF> within a mode dependency list must contain two context references (respectively, the port prototype and the mode declaration group prototype within the interface categorizing the port prototype) and one target reference (the mode within the mode declaration group that types the declaration group prototype).

When a mode dependency disables an event that would have written to a queue (e.g. data receive when isQueued is true) RTA-RTE suppresses both the write and the runnable entity activation.

4.10.3 Inter-Runnable Variables

Inter-runnable variables support communication between runnable entities within the same instance of an AUTOSAR SW-C. An internal behavior can declare zero or more inter-runnable variables using the <INTER-RUNNABLE-VARIABLE> tag:

```
interrunnable_variables ::=
    <INTER-RUNNABLE-VARIABLES>
    +interrunnable_variable
    </INTER-RUNNABLE-VARIABLES>

interrunnable_variable ::=
    <INTER-RUNNABLE-VARIABLE>
    short_name
        ( sw_data_def_props_for_measurement )
    <TYPE-TREF>ref</TYPE-TREF>
    <COMMUNICATION-APPROACH>
    (EXPLICIT|IMPLICIT)
    </COMMUNICATION-APPROACH>
    <INIT-VALUE-REF>ref</INIT-VALUE-REF>
    </INTER-RUNNABLE-VARIABLE>
```

Access to an inter-runnable variable, whether read or write, is atomic. If an atomic read-modify-write operation is required then an exclusive area must be used instead.

The <INIT-VALUE-REF> reference defines the inter-runnable variable's initial value. The reference should identify a constant's value specification and not to the constant specification element itself.

The specification within an inter-runnable variable of sw_data_def_props is optional. If present it describes whether the variable is measurable. For more details see Section 4.7.

4.10.4 Exclusive Areas

An internal behavior can declare zero or more exclusive areas that are used to provide mutual exclusive access to state shared between runnable entities.

The scope of an exclusive area is restricted to a software-component instance and thus an exclusive area cannot be used to control access to state shared between software-component instances.

```
exclusive_areas ::=
    <EXCLUSIVE-AREAS>
    +exclusive_area
    </EXCLUSIVE-AREAS>

exclusive_areas ::=
    <EXCLUSIVE-AREA>
    short_name
    </EXCLUSIVE-AREA>
```

An <EXCLUSIVE-AREA> can be declared with an optional hint to the RTE generator as to how the exclusive area should be implemented. See Section 4.19.2.

4.10.5 NVRAM Mapping

An internal behavior can define zero or more NVRAM mappings.

```
nvram_mappings ::=
   <NVRAM-MAPPINGS>
   +nvram_mapping
   </NVRAM-MAPPINGS>
```

Each nvram_mapping defines a single mapping for the behavior.

```
nvram_mapping ::=
    <NVRAM-MAPPING>
    short_name
    <MIRROR-BLOCK-REF>
    ref
    </MIRROR-BLOCK-REF>
    </NVRAM-MAPPINGS>
```

The <MIRROR-BLOCK-REF> associates an NVRAM mapping with a per-instance memory. When a mirror block reference is defined, the RTA-RTE RTE generator uses the RamBlockLocationSymbol from the NvRamAllocation within the ECUC as the name of the instantiated per-instance memory.

4.10.6 Calibration

An internal behavior element can define zero or more per-instance and shared calibration parameters.

Calibration Element

A calibration element prototype declares a single calibratable parameter.

```
calprm_element::=
    <CALPRM-ELEMENT-PROTOTYPE>
    short_name
    ( sw_addr_method_ref )
    type_ref
    </CALPRM-ELEMENT-PROTOTYPE>
```

A calibration element within an internal behavior[2] can be either per-instance or shared depending on where it is declared within the internal behavior.

The sw_addr_method_ref is used to aggregate calibration element prototypes from the same SWC instance (and of the same type) into calibration element groups.

```
sw_addr_method_ref ::=
```

[2] In addition to declaring calibration parameters within an internal behavior they can also be declared within a calibration component type. Parameters declared within a calibration component type can be used by multiple SWC instances.

```
    <SW-DATA-DEF-PROPS>
    <SW-ADDR-METHOD-REF>ref</SW-ADDR-METHOD-REF>
    </SW-DATA-DEF-PROPS>
```

The <SW-ADDR-METHOD-REF> must refer to a <SW-ADDR-METHOD> element.

Per-instance

The <PER-INSTANCE-CALPRMS> element aggregates all calibration elements that are assigned unique values for each referencing SWC instance.

```
per_instance_calprms ::=
    <PER-INSTANCE-CALPRMS>
    +calprm_element
    </PER-INSTANCE-CALPRMS>
```

Each calprm_element definition defines the name, data type, and optionally the swAddrMethod, of a single calibration parameter. The definition of a calprm_element is described in Section 4.6.3.

Shared

The <SHARED-CALMRMS> element aggregates all calibration elements that have common values for each referencing SWC instance.

```
shared_calprms ::=
    <SHARED-CALPRMS>
    +calprm_element
    </SHARED-CALPRMS>
```

Each calprm_element definition defines the name, data type, and optionally the swAddrMethod, of a single calibration parameter.

Initial Values

Initial value assignment for shared and per-instance calibration parameters are contained within instances of the <LOCAL-PARAMETER-INIT-VALUE-ASSIGNMENT> element:

```
init_values ::=
    <INIT-VALUES>
        *init_value_assignment
    </INIT-VALUES>
```

An initial value assignment element contains a pair of references; the first to a constant containing the initial value and the second the associated calibration prototype:

```
init_value_assignment ::=
    <LOCAL-PARAMETER-INIT-VALUE-ASSIGNMENT>
    <INIT-VALUE-REF>ref</INIT-VALUE-REF>
        <PARAMETER-REF>ref</PARAMETER-REF>
    </LOCAL-PARAMETER-INIT-VALUE-ASSIGNMENT>
```

The referenced calibration parameter must be in the same InternalBehavior as the LocalParameterInitValueAssignment. The referenced constant and the calibration parameter must have the same underlying type.

4.10.7 Per-Instance Memories

An internal behavior element can define zero or more per-instance memories (PIM) that are instantiated by RTA-RTE once for each instance of the software-component.

The per-instance memory sections within an AUTOSAR software-component are declared within the <PER-INSTANCE-MEMORYS> element:

```
per_instance_memorys ::=
    <PER-INSTANCE-MEMORYS>
    +pim
    </PER-INSTANCE-MEMORYS>
```

Each per-instance memory definition defines the name and type of a single per-instance memory.

```
pim ::=
    <PER-INSTANCE-MEMORY>
    short_name
    <TYPE>string</TYPE>
    <TYPE-DEFINITION>string</TYPE-DEFINITION>
    </PER-INSTANCE-MEMORY>
```

RTA-RTE uses the <TYPE> and <TYPE-DEFINITION> elements to form the data type of the per-instance memory through a type definition:

```
typedef <TYPE-DEFINITION> <TYPE>;
```

Where <TYPE-DEFINITION> and <TYPE> are extracted from the input. Therefore the <TYPE-DEFINITION> must be the required C-type and the <TYPE> the data type name. The <TYPE-DEFINITION> is used without interpretation by RTA-RTE and is not checked for syntactic correctness.

The <TYPE> of a per-instance memory is visible as a C typedef to a software-component. Therefore all defined per-instance memories of a single software-component type must have unique short names and types.

4.10.8 Port options

The <PORT-API-OPTION> element defines port-defined argument values and offers control over whether or not the port-API (indirect-API) is generated for a port.

```
port_api_options ::=
    <PORT-API-OPTIONS>
    *port_api_option
    </PORT-API-OPTIONS>
```

Each port_api_option element defines the indirect_api options and/or the port argument values:

```
port_api_option ::=
    <PORT-API-OPTION>
    ( indirect_api )
    ( port_arg_values )
    ( enable_take_address )
    <PORT-REF>ref</PORT-REF>
    </PORT-API-OPTION>
```

The <PORT-REF> element must reference a port within the SWC type associated with the encapsulating internal behavior.

Indirect-API (Port-API) Control

Support for the indirect API can be enabled/disabled for individual ports within a SWC type using the <INDIRECT-API> element.

```
indirect_api ::=
    <INDIRECT-API>(true|false)</INDIRECT-API>
```

Disabling the indirect-API for a port will reduce the ROM usage for a SWC instance.

The indirect-API is always generated if an SWC is declared as supporting multiple instances irrespective of the <PORT-API-OPTION> settings.

Enable Take Address

When “enable take address” is specified for a port the API mapping generated within the application header file will permit the address of an API function to be taken. Without this option the mapping may be implemented as a macro that does not support the address operator.

```
enable_take_address ::=
    <ENABLE-TAKE-ADDRESS>(true|false)</ENABLE-TAKE-ADDRESS>
```

Port-Defined Argument Values

Port-defined argument values support the interaction between SWCs and Basic Software Modules by the automatic adaptation of server invocation by RTA-RTE depending on the port used to invoke the server.

Port-defined arguments can only be applied to provided (server) ports.

Each port_arg_values list defines the valid argument values for the port referenced by the Port-API options element:

```
port_argument_list ::=
    <PORT-ARG-VALUES>
    *value_spec
    </PORT-ARG-VALUES>
```

One port-defined argument is passed to the operations within the client-server interface that categorizes the referenced port for each “value” defined within the argument list.

```
value_spec ::=
    <xyz-LITERAL>
    short_name
    <TYPE-TREF>ref</TYPE-TREF>
    <VALUE>string</VALUE>
    </xyz-LITERAL>
```

4.10.9 Runnable Entities

The runnable entities within an AUTOSAR software-component are declared within the <RUNNABLES> element:

```
runnable_entitys ::=
    <RUNNABLES>
    +runnable_entity
    </RUNNABLES>
```

The only mandatory information for a runnable entity is its short name and symbol. However, if the runnable entity needs to interact with the interfaces in the associated software component then this must be stated explicitly.

```
runnable_entity ::=
    <RUNNABLE_ENTITY>
    short_name
    ( pure )
    ( data_read_points )
    ( data_receive_points
    ( data_send_points )
    ( data_write_points )
    ( runnable_entity_runs_in_exclusive_areas )
    ( mode_switch_points )
    ( read_variables )
    ( minimum_start_interval )
    ( server_call_points )
    symbol
    ( runnable_entity_can_enter_exclusive_areas )
    ( wait_points )
    ( written_variables )
    </RUNNABLE_ENTITY>
```

Concurrent execution

The runnable entity responsible for servicing an Operation Invoked Event for a client-server operation can be marked as subject to concurrent execution (previous releases of RTA-RTE referred to such runnable entities as ‘pure’) to enable un-queued execution.

```
pure ::=
    <CAN-BE-INVOKED-CONCURRENTLY>
    (true|false)
```

```
    </CAN-BE-INVOKED-CONCURRENTLY>
```

For intra-ECU client-server communication, a runnable entity that is marked as concurrently executable will be directly invoked by RTA-RTE irrespective of whether intra-task or inter-task communication is involved provided all clients access the server synchronously.

Concurrent execution affects runnable entities activated as a result of an `OperationInvoked` RTE event as well as allowing runnables activated by multiple RTEEvents to be mapped to preemptable tasks.

Write Accesses

When the runnable is to be used to send data on a software component interface using implicit RTE API it must define the data write points where the data is written by the runnable entity. Each data write access point needs to be named and must reference a data item in a sender-receiver interface.

```
data_write_points ::=
    <DATA-WRITE-ACCESSS>
    +data_write_point
    </DATA-WRITE-ACCESSS>

data_write_point ::=
    <DATA-WRITE-ACCESS>
    short_name
    <DATA-ELEMENT-IREF>
    instance_ref
    </DATA-ELEMENT-IREF>
    </DATA-WRITE-ACCESS>
```

A data write point is necessary for RTA-RTE to create the `Rte_IWrite` and `Rte_IInvalidate` API calls.

The `<DATA-ELEMENT-IREF>` instance reference defines the data element to which this event applies and requires a single context reference (the port prototype) and a target reference (the data element within the interface that categorizes the port).

4.10.10 Read Accesses

When the runnable is used to read data on a software component interface using implicit RTE API it must define the data read points where the data is read by the runnable entity. Each data read access point needs to be named and must reference a data item in a sender-receiver interface.

```
data_read_points ::=
    <DATA-READ-ACCESSS>
    +data_read_point
    </DATA-READ-ACCESSS>

data_read_point ::=
    <DATA-READ-ACCESS>
```

```
    short_name
    <DATA-ELEMENT-IREF>
    instance_ref
    </DATA-ELEMENT-IREF>
    </DATA-READ-ACCESS>
```

A data read point is necessary for RTA-RTE to create the `Rte_IRead` and `Rte_IStatus` API calls.

The <DATA-ELEMENT-IREF> instance reference defines the data element to which this event applies and requires a single context reference (the port prototype) and a target reference (the data element within the interface that categorizes the port).

Receive Points

When the runnable is used to receive data on a software component interface using the explicit RTE API it must define the data receive points where the data is received. Each data receive point needs to be named and reference a data item in a sender-receiver interface.

```
data_receive_points ::=
    <DATA-RECEIVE-POINTS>
    +data_receive_point
    </DATA-RECEIVE-POINTS>

data_receive_point ::=
    <DATA-RECEIVE-POINT>
    short_name
    <DATA-ELEMENT-IREF>
    instance_ref
    </DATA-ELEMENT-IREF>
    </DATA-RECEIVE-POINT>
```

A data receive point is necessary for RTA-RTE to create the `Rte_Receive` API calls.

The <DATA-ELEMENT-IREF> instance reference defines the data element to which this event applies and requires a single context reference (the port prototype) and a target reference (the data element within the interface that categorizes the port).

Send Points

When the runnable is used to send data on a software component interface using the explicit RTE API it must define which data items are sent. Each data send point needs to be named and must reference a data item in a sender-receiver interface.

```
data_send_points ::=
    <DATA-SEND-POINTS>
    +data_send_point
    </DATA-SEND-POINTS>

data_send_point ::=
    <DATA-SEND-POINT>
```

```
        short_name
        <DATA-ELEMENT-IREF>
        instance_ref
        </DATA-ELEMENT-IREF>
        </DATA-SEND-POINT>
```

A data send point is necessary for RTA-RTE to create the Rte_Send or Rte_Write API calls.

The <DATA-ELEMENT-IREF> instance reference defines the data element to which this event applies and requires a single context reference (the port prototype) and a target reference (the data element within the interface that categorizes the port).

Mode Switch Points

If a runnable is used as a mode manager that will make an Rte_Switch API call over a sender-receiver interface, then the runnable needs to define a mode switch point.

A mode switch point is necessary for RTA-RTE to create the Rte_Switch API call.

Each mode switch point needs to be named and must reference a mode declaration group prototype in a sender-receiver interface.

```
    mode_switch_points ::=
        <MODE-SWITCH-POINTS>
        +mode_switch_point
        </MODE-SWITCH-POINTS>

    mode_switch_point ::=
        <MODE-SWITCH-POINT>
        short_name
        <MODE-GROUP-IREF>
        <P-PORT-PROTOTYPE-REF>
        ref
        </P-PORT-PROTOTYPE-REF>
        <MODE-DECLARATION-GROUP-PROTOTYPE-REF>
        ref
        </MODE-DECLARATION-GROUP-PROTOTYPE-REF>
        </MODE-GROUP-IREF>
        </MODE-SWITCH-POINT>
```

The <MODE-SWITCH-IREF> instance reference defines the mode declaration group prototype to which the switch point applies. The instance reference requires a port prototype reference and a mode declaration group prototype reference (which must be within the interface that categorizes the port).

Server Call Points

If a runnable is used as a client that will make a server call over a client-server interface, then the runnable needs to define the server interface operation(s) that it calls.

```
    server_call_points ::=
```

```
    <SERVER-CALL-POINTS>
    + ( asynchronous_server_call_point |
    synchronous_server_call_point )
    </SERVER-CALL-POINTS>

asynchronous_server_call_point ::=
    <ASYNCHRONOUS-SERVER-CALL-POINT>
    short_name
    <OPERATION-IREFS>
    +operation_iref
    </OPERATION-IREFS>
    </ASYNCHRONOUS-SERVER-CALL-POINT>

synchronous_server_call_point ::=
    <SYNCHRONOUS-SERVER-CALL-POINT>
    short_name
    <OPERATION-IREFS>
    +operation_iref
    </OPERATION-IREFS>
    <TIMEOUT>time</TIMEOUT>
    </SYNCHRONOUS-SERVER-CALL-POINT>
```

The <TIMEOUT> defines the maximum time an inter-ECU call will block before returning. No trailing text indicating the units is required or permitted. Fractions of a second can be expressed using floating point format with a period (“.”) as the decimal separator.

```
operation_iref ::=
    <OPERATION-IREF>instance_ref</OPERATION-IREF>
```

The <OPERATION-IREF> instance reference defines the operation to which this event applies and requires a single context reference (the port prototype) and a target reference (the operation within the interface that categorizes the port).

RTA-RTE supports at most one operation_iref reference per server call point.

Symbol

The SYMBOL element provides the C name of the function that implements the runnable entity.

```
symbol ::=
    <SYMBOL>C-Ident</SYMBOL>
```

RTA-RTE expects that the symbol will be defined in an implementation of a software component. A protyotype for the runnable entity is declared in the component’s application header file.

Blocking API calls

When a runnable needs to block (i.e. wait on events) it must name the points at which it waits and, for each point, define the associated <RTE-EVENT>.

```
wait_points ::=
    <WAIT-POINTS>
    +wait_point
    </WAIT-POINTS>

wait_point ::=
    <WAIT-POINT>
    short_name
    <TRIGGER-REFS>
    +( <TRIGGER-REF>ref</TRIGGER-REF> )
    </TRIGGER-REFS>
    <TIMEOUT>time</TIMEOUT>
    </WAIT-POINT>
```

The <TIMEOUT> elements defines the maximum time, in seconds, a blocking API call will wait before returning. No trailing text indicating the units is required or permitted. Fractions of a second can be expressed using floating point format with a period (".") as the decimal separator.

When a blocking API call is required the <RTE-EVENT> referenced within a trigger reference must ***not*** *also reference a runnable entity.*

A Wait Point is only permitted to reference the following RTE events:

- Data Received
- Data Send Completed
- Asynchronous Server Call Returns
- Mode Switched Acknowledge

Access to Exclusive Areas

Each runnable can specify whether it can get access to a critical section at runtime, either implicitly when invoked by the generated RTE, or explicitly by making an appropriate RTE call.

Explicit exclusive areas are defined by a "runnable can enter" element:

```
runnable_entity_can_enter_exclusive_areas ::=
    <CAN-ENTER-EXCLUSIVE-AREA-REFS>
    +exclusive_area_reference
    </CAN-ENTER-EXCLUSIVE-AREA-REFS>

exclusive_area_reference ::=
    <CAN-ENTER-EXCLUSIVE-AREA-REF>
    ref
    </CAN-ENTER-EXCLUSIVE-AREA-REF>
```

Implicit exclusive areas are defined within a "runs inside exclusive area" element:

```
runnable_entity_runs_in_exclusive_areas ::=
    <RUNS-INSIDE-EXCLUSIVE-AREA-REFS>
    +implicit_area_reference
    </RUNS-INSIDE-EXCLUSIVE-AREA-REFS>

implicit_area_reference ::=
    <RUNS-INSIDE-EXCLUSIVE-AREA-REF>
    ref
    </RUNS-INSIDE-EXCLUSIVE-AREA-REF>
```

Exclusive areas can only be used for concurrency control within a software-component instance.

Read Variables

Each runnable can specify whether it can get “read” access to an inter-runnable variable at runtime, either implicitly or explicitly, by making an appropriate RTE call.

```
read_variables ::=
    <READ-VARIABLE-REFS>
    +read_variable
    </READ-VARIABLE-REFS>

read_variable ::=
    <READ-VARIABLE-REF>ref</READ-VARIABLE-REF>
```

Written Variables

Each runnable can specify whether it can get “write” access to an inter-runnable variable at runtime, either implicitly or explicitly, by making an appropriate RTE call.

```
written_variables ::=
    <WRITTEN-VARIABLE-REFS>
    +written_variable
    </WRITTEN-VARIABLE-REFS>

written_variable ::=
    <WRITTEN-VARIABLE-REF>ref</WRITTEN-VARIABLE-REF>
```

Minimum Start Interval

The `<MINIMUM-START-INTERVAL>` element specifies the time (in seconds) between which any two executions of the runnable are guaranteed to be separated.

```
minimum_start_interval ::=
    <MINIMUM-START-INTERVAL>
    time
    </MINIMUM-START-INTERVAL>
```

A minimum start interval specification is typically used with runnable entities triggered by `DataReceivedEvents` to limit the rate at which events are processed. However a minimum start interval can be applied to any runnable entity except for those triggered by an `OperationInvokedEvent`.

4.10.11 Multiple Instantiation

The <SUPPORTS-MULTIPLE-INSTANTIATION> element determines whether or not multiple instances of a software component are permitted.

```
multiple_instances ::=
    <SUPPORTS-MULTIPLE-INSTANTIATION>
    ( true | false )
    </SUPPORTS-MULTIPLE-INSTANTIATION>
```

A value of “true” indicates that multiple instances of a software-component are permitted whereas “false” indicates that multiple instances are forbidden.

RTA-RTE is able to apply higher levels of optimization, especially at the “Contract” generation stage, when multiple instantiation is forbidden.

4.11 Implementation

The implementation description of a software component defines its implementation characteristics.

This release of RTA-RTE uses the <SWC-IMPLEMENTATION> element to determine whether or not a software-component is delivered as source code or object-code.

```
implementation ::=
    <SWC-IMPLEMENTATION>
    short_name
    ( code_descriptor )
    <BEHAVIOR-REF>ref</BEHAVIOR-REF>
    <SWC-IMPLEMENTATION>
```

An <IMPLEMENTATION> or <SWC-IMPLEMENTATION> element references an internal behavior element that in turn references a SWC type. This relationship permits multiple implementation elements to be “associated” with a particular SWC type, e.g. one implementation for each supported processor.

RTE generation occurs for a specific implementation and therefore RTA-RTE needs additional information in the form of an implementation selection element to determine which <IMPLEMENTATION> or <SWC-IMPLEMENTATION> element to use for a specific SWC instance. See Section 4.19.2

4.11.1 Code Descriptor

The <CODE-DESCRIPTOR> element determines whether the SWC type for which the implementation is associated is delivered as source-code or object-code.

```
code_descriptor ::=
    <CODE-DESCRIPTORS>
        <CODE>
    short_name
        <ARTIFACT-DESCRIPTORS>
        <AUTOSAR-ENGINEERING-OBJECT>
```

```
        <CATEGORY>(SWSRC|SWOBJ)</CATEGORY>
        </AUTOSAR-ENGINEERING-OBJECT>
        </ARTIFACT-DESCRIPTORS>
        </CODE>
    </CODE-DESCRIPTORS>
```

When an SWC instance is delivered as “SWSRC”, RTA-RTE will elide the generation of functions in the generated RTE wherever possible; for example a client-server `Rte_Call` with an optimized API mapping that directly invokes the server runnable does not require the body of the RTE `Call` API to be generated.

4.12 Signals

A signal, or message, is the basic unit of communication between ECUs, usually carrying a single logical item of data, such as a value representing engine temperature. AUTOSAR distinguishes between three kinds of signal: system signals, interaction layer signals (i-signals) and COM signals, all of which may refer to the same data item. To understand the configuration of inter-ECU commumication within RTA-RTE, it is important to understand what is referred to by the different types of signal.

System Signal —the most abstract representation of a message used to carry data, and is the kind of signal referred to when data mappings are configured.

I-Signal —an instance of a system signal in a particular interaction layer PDU. A separate class is used in order to support “fan-out”, enabling a single system signal to be referred to in multiple PDU instances, sent to different destinations. Consequently, there may be n i-signals corresponding to a single system signal.

Com signal —an element of the COM module configuration, representing a concrete instance of a signal. Com signals correspond directly to i-signals, i.e. for every i-signal in the system definition that is required for the ECU being generated, there should be a Com signal. The configuration of Com signals includes detailed information about the communication stack that is not included in the high-level system definition, such as repetition counts, filters, etc.

The `<SYSTEM-SIGNAL>` element defines an AUTOSAR signal.

```
system_signal ::=
    <SYSTEM-SIGNAL>
    short_name
    <LENGTH>int</LENGTH>
    </SYSTEM-SIGNAL>
```

The signal length is defined in bits. All other properties of the signal, such as its bit position, are set when a signal type is mapped into a frame type.

An I-Signal describes an instance of a system signal in a particular interaction layer PDU and hence the `<I-SIGNAL>` references a `<SYSTEM-SIGNAL>`.

```
i_signal ::=
    <I-SIGNAL>
    short_name
    <SYSTEM-SIGNAL-REF>ref</SYSTEM-SIGNAL-REF>
    </I-SIGNAL>
```

An I-Signal is referenced from an <I-SIGNAL-TO-I-PDU-MAPPING> when the signal is packed into an I-PDU—see Section 4.14.

An I-Signal describes an instance of a system signal in a particular interaction layer PDU and hence the <I-SIGNAL> references a <SYSTEM-SIGNAL>.


```
i_signal ::=
    <I-SIGNAL>
    short_name
    <SYSTEM-SIGNAL-REF>ref</SYSTEM-SIGNAL-REF>
    </I-SIGNAL>
```

An I-Signal is referenced from an <I-SIGNAL-TO-I-PDU-MAPPING> when the signal is packed into an I-PDU—see Section 4.14.

4.13 System Signal Group

The <SYSTEM-SIGNAL-GROUP> element defines a group of AUTOSAR signals that can be sent and received atomically.

```
signal_group_type ::=
    <SYSTEM-SIGNAL-GROUP>
    short_name
    <SYSTEM-SIGNALS-REFS>
    +signal_ref
    </SYSTEM-SIGNALS-REFS>
    </SYSTEM-SIGNAL-GROUP>

signal_ref ::=
    <SYSTEM-SIGNALS-REF>ref</SYSTEM-SIGNALS-REF>
```

4.14 PDU Type

The <SIGNAL-I-PDU> element captures the definition of an I-PDU including the position of all signals within the frame.

```
pdu_type ::=
    <SIGNAL-I-PDU>
    short_name
    <LENGTH>int</LENGTH>
    <SIGNAL-TO-PDU-MAPPINGS>
    +signal_to_pdu_mapping
    </SIGNAL-TO-PDU-MAPPINGS>
    </SIGNAL-I-PDU>
```

The PDU length is specified in bits.

Each <I-SIGNAL-TO-I-PDU-MAPPING> element defines the name, packing order and start position in bits of a referenced I-signal:

```
signal_to_pdu_mapping ::=
    <I-SIGNAL-TO-I-PDU-MAPPING>
    short_name
    <PACKING-BYTE-ORDER>
    ( MOST-SIGNIFICANT-BYTE-FIRST |
      MOST-SIGNIFICANT-BYTE-LAST )
    </PACKING-BYTE-ORDER>
    <SIGNAL-REF>ref</SIGNAL-REF>
    <START-POSITION>int</START-POSITION>
    </I-SIGNAL-TO-I-PDU-MAPPING>
```

The signal start position within the frame type is specified in bits.

The <SIGNAL-REF> element must refer to a <I-SIGNAL> defined in the input.

4.15 ECU Types

The <ECU> element captures the definition of an ECU type.

```
ecu ::=
    <ECU>
    short_name
    ( ecu_abstraction_ref )
    </ECU>
```

The communication buffer definition is not used by RTA-RTE but must be specified for the XML to be valid.

4.16 Composition

A composition defines instances of component types and the connections between ports.

```
composition_type ::=
    <COMPOSITION-TYPE>
    short_name
    component_prototypes
    connector_definitions
    port_prototypes
    </COMPOSITION-TYPE>
```

4.16.1 Component Prototypes

A composition creates the component prototypes—in the XML input these are defined using the <COMPONENTS> element:

```
component_instances ::=
    <COMPONENTS>
    +component_prototype
    </COMPONENTS>
```

Each component prototype defines two pieces of information; a name so the prototype can be referenced from other XML elements and a reference to the relevant software-component type.

```
component_prototype ::=
    <COMPONENT-PROTOTYPE>
    short_name
    <TYPE-TREF>ref</TYPE-TREF>
    </COMPONENT-PROTOTYPE>
```

4.16.2 Connector Definitions

The connector definitions define the connections between provided and required ports.

```
connector_definitions ::=
    <CONNECTORS>
    +assembly_connector
    +delegation_connector
    </CONNECTORS>
```

Assembly Connector

An assembly connector connects provided and required ports within a composition (and therefore the connection should not span a composition). A delegation connector “exports” ports within a composition to enable compositions to be further composed into higher level compositions.

Each assembly connector defines two port instances; a provided and required port.

```
assembly_connector ::=
    <ASSEMBLY-CONNECTOR-PROTOTYPE>
    short_name
    <PROVIDER-IREF>instance_ref</PROVIDER-IREF>
    <REQUESTER-IREF>instance_ref</REQUESTER-IREF>
    </ASSEMBLY-CONNECTOR-PROTOTYPE>
```

The <PROVIDER-IREF> instance reference defines the providing port prototype to which this connector applies and requires a single context reference (the component prototype) and a target reference (the port prototype within the SWC type).

The <REQUIRER-IREF> instance reference defines the requiring port prototype to which this connector applies and requires a single context reference (the component prototype) and a target reference (the port prototype within the SWC type).

For both <PROVIDER-IREF> and <REQUIRER-IREF> instance references the referenced component prototype must be within the same composition as the assembly connector.

Delegation Connector

A delegation connector defines two port prototypes which must either both be provided or both be required.

```
delegation_connector ::=
```

```
<DELEGATION-CONNECTOR-PROTOTYPE>
short_name
<INNER-PORT-IREF>instance_ref</INNER-PORT-IREF>
<OUTER-PORT-REF>instance_ref</OUTER-PORT-IREF>
</DELEGATION-CONNECTOR-PROTOTYPE>
```

The instance reference within the <INNER-PORT-IREF> should refer to a port prototype on a component prototype within the composition, whereas the reference within the <OUTER-PORT-REF> should refer to a port prototype of the composition itself.

Delegation connectors must always connect ports of the same type. RTA-RTE flags attempts to connect PPorts to RPorts using delegation connectors as invalid configurations.

Service Connector

In addition to assembly and delegation connectors, RTA-RTE supports a third type of connector, the “SERVICE-CONNECTOR-PROTOTYPE”, which is used to connect a software component to an AUTOSAR service. However, service connectors should not appear in a standard software composition, but are placed instead in the ECU’s software composition, which is also where services themselves are configured on an ECU (see Section **??**).

4.16.3 Port Prototypes

A delegation connector connects a port prototype on a component prototype within the composition to a port prototype defined within the composition.

```
port_prototypes::=
    <PORTS>
    +port_prototype
    </PORTS>
```

The definition of port prototypes is covered in Section 4.4.1.

The port prototype within the composition can be either the source or destination of the delegation connector.

4.17 ECU Instances

A system topology type defines the instances of ECU types:

```
ecu_instances ::=
     <ECU-INSTANCES>
     +ecu-instance
     </ECU-INSTANCES>
```

Each ECU instance is defined using an <ECU-INSTANCE> element. As with the <SYSTEM>, the ECU instance is named and therefore can be referenced from other elements:

```
ecu_instance ::=
     <ECU-INSTANCE>
```

```
        short_name
        <ECU-TREF>ref</ECU-TREF>
        ( ecu_port_instances )
        </ECU-INSTANCE>

    ecu_instance ::=
        <ECU-INSTANCE>
        short_name
        ( ecu_connectors )
        </ECU-INSTANCE>
```

The <ECU-TREF> element must refer to an ECU element defined within the input.

Before an ECU instance can be referenced, for example on the command line, a topology instance must be defined that references the encapsulating system topology type.

The ECU instance defines zero or more communication port instances:

```
    ecu_port_instances ::=
        <PORT-INSTANCES>
        +ecu_port_instance
        </PORT-INSTANCES>

    ecu_connectors ::=
        <CONNECTORS>
        +communication_connector
        </CONNECTORS>

    communication_connector ::=
        <COMMUNICATION-CONNECTOR>
        short_name
        <ECU-COMM-PORT-INSTANCES/>
        </COMMUNICATION-CONNECTOR>
```

RTA-RTE does not use the <ECU-COMM-PORT-INSTANCE> element.

A communication port instance defines the port speed and the relevant port type (within the ECU type):

```
    ecu_port_instance ::=
        <CAN-COMMUNICATION-PORT-INSTANCE>
        short_name
        <COMM-PORT-ID>int</COMM-PORT-ID>
        <COMMUNICATION-SPEED-PORT>
        int
        </COMMUNICATION-SPEED-PORT>
        <PORT-TREF>ref</PORT-TREF>
        <PROTOCOL-VERSION>
        ( L-2-0-A | L-2-0-B )
        </PROTOCOL-VERSION>
        </CAN-COMMUNICATION-PORT-INSTANCE>
```

The <PORT-TREF> element references an <ECU-COMMUNICATION-PORT> on the ECU type. It should references a communication port on the same ECU type as specified within the <ECU-INSTANCE>.

The <PROTOCOL-VERSION> element defines whether the CAN communication port is CAN 2.0a (L-2-0-A) or CAN 2.0b (L-2-0-B) compliant.

4.18 System Description

The system description defines the core elements of an AUTOSAR system including how SWC prototypes are mapped onto the available ECU instances and how AUTOSAR signals are mapped onto Com signals.

A system definition must be named so that other XML elements can refer to the system.

```
system ::=
    <SYSTEM>
    short_name
    ( mapping )
    sw_composition
    </SYSTEM>
```

4.18.1 Mapping Sets

The mapping of both component instances to ECU instances and, for inter-ECU communication, data element prototypes to system signals is performed by the mapping sets:

```
mapping ::=
    <MAPPING>
    short_name
    ( data_mappings )
    ( swc_mappings )
    </MAPPING>
```

The two mapping sets are responsible for mapping component interface communication to physical data transmission and mapping components to ECUs.

4.18.2 Data Mapping

The Data Mapping maps communication between software-components to physical (network) data transmission.

There are two categories of data mapping: those for sender-receiver and those for client-server communication. These may be freely mixed within the <DATA-MAPPINGS> block:

```
data_mappings ::=
    <DATA-MAPPINGS>
    *( sender_receiver_mapping | client_server_mapping )
    <DATA-MAPPINGS>
```

Sender-Receiver Communication

Sender-receiver mappings map a data element prototype, part of a sender-receiver interface, either to a single signal (if the data type is primitive), or to a signal group (if the data type is complex).

```
sender_receiver_mapping ::=
    ( sender_receiver_signal_mapping |
    sender_receiver_signal_group_mapping )

sender_receiver_signal_mapping ::=
    <SENDER-RECEIVER-TO-SIGNAL-MAPPING>
    data_element_iref
    signal_ref
    </SENDER-RECEIVER-TO-SIGNAL-MAPPING>

sender_receiver_signal_group_mapping ::=
    <SENDER-RECEIVER-TO-SIGNAL-GROUP-MAPPING>
    data_element_iref
    signal_group_ref
    type_mapping
    </SENDER-RECEIVER-TO-SIGNAL-GROUP-MAPPING>
```

Data element instances consist of references to a software component instance within a composition, a port, and a data element itself, which appears within the description of the sender-receiver interface that characterizes the port.

```
data_element_iref ::=
    <DATA-ELEMENT-IREF>
    +component_prototype_ref
    <PORT-PROTOTYPE-REF>ref<PORT-PROTOTYPE-REF>
    <DATA-ELEMENT-REF>ref</DATA-ELEMENT-REF>
    </DATA-ELEMENT-IREF>
```

The `<DATA-ELEMENT-IREF>` element includes one `<COMPONENT-PROTOTYPE-REF>` for each level of the composition hierarchy—when nested compositions are present multiple `<COMPONENT-PROTOTYPE-REF>` are necessary.

```
component_prototype_ref ::=
    <COMPONENT-PROTOTYPE-REF>
    ref
    </COMPONENT-PROTOTYPE-REF>

signal_ref ::=
    <SIGNAL-REF>ref</SIGNAL-REF>

signal_group_ref ::=
    <SIGNAL-GROUP-REF>ref</SIGNAL-GROUP-REF>
```

Complex types are mapped to signal groups: these types may be either records or arrays.

```
type_mapping ::=
    <TYPE-MAPPING>
    ( sr_record_type_mapping | sr_array_type_mapping )
    </TYPE-MAPPING>
```

Records are mapped using the <SENDER-REC-RECORD-TYPE-MAPPING> element:

```
sr_record_type_mapping ::=
    <SENDER-REC-RECORD-TYPE-MAPPING>
    <RECORD-ELEMENT-MAPPINGS>
    +sr_record_element_mapping
    </RECORD-ELEMENT-MAPPINGS>
    </SENDER-REC-RECORD-TYPE-MAPPING>
```

Each record element, whether a primitive or complex type, is mapped using a <SENDER-REC-RECORD-ELEMENT-MAPPING> element.

```
sr_record_element_mapping ::=
    <SENDER-REC-RECORD-ELEMENT-MAPPING>
    ( sr_simple_record_element_mapping | sr_complex_record_element_mapping
       )
    </SENDER-REC-RECORD-ELEMENT-MAPPING>
```

Record elements can be primitive or complex types. For an element which is a primitive type the mapping references a single signal.

```
sr_simple_record_element_mapping ::=
    sr_record_element_iref
    signal_ref
```

For record elements that are themselves complex types a nested mapping element is used.

```
sr_complex_record_element_mapping ::=
    <COMPLEX-TYPE_MAPPING>
    ( sr_record_type_mapping | sr_array_type_mapping )
    </COMPLEX-TYPE_MAPPING>
    sr_record_element_ref
```

The sr_record_element_ref references the relevant record element.

```
sr_record_element_ref ::=
    <RECORD-ELEMENT-REF>
    ref
    </RECORD-ELEMENT-REF>
```

Arrays are mapped element-by-element, in a similar way to records, with each element being mapped to a separate signal:

```
sr_array_type_mapping ::=
    <SENDER-REC-ARRAY-TYPE-MAPPING>
    <ARRAY-ELEMENT-MAPPINGS>
    +sr_array_element_mapping
```

```
    </ARRAY-ELEMENT-MAPPINGS>
    </SENDER-REC-ARRAY-TYPE-MAPPING>
```

Each element of the array is mapped using a <SENDER-REC-ARRAY-ELEMENT-MAPPING> element.

```
sr_array_element_mapping ::=
    <SENDER-REC-ARRAY-ELEMENT-MAPPING>
    ( sr_simple_array_element_mapping | sr_complex_array_element_mapping )
    </SENDER-REC-ARRAY-ELEMENT-MAPPING>
```

Array elements can be primitive or complex types. For an element which is a primitive type the mapping references a single signal.

```
sr_simple_array_element_mapping ::=
    sr_array_element_ref
    signal_ref
```

For array elements that are themselves complex types a nested mapping element is used.

```
sr_complex_array_element_mapping ::=
    <COMPLEX-TYPE_MAPPING>
    sr_record_type_mapping
    </COMPLEX-TYPE_MAPPING>
    sr_array_element_ref
```

Note the restriction here that an array may not contain nested array-type elements—only records can be nested.

```
sr_array_element_ref ::=
    <INDEXED-ARRAY-ELEMENT>
    <ARRAY-ELEMENT-REF>ref</ARRAY-ELEMENT-REF>
    <INDEX>int</INDEX>
    </INDEXED-ARRAY-ELEMENT>
```

The <INDEX> gives the position of the mapped element within the array; the <ARRAY-ELEMENT-REF> refers to the type of the array element, and should therefore have the same value for all the mapped elements of a given array.

Client-Server Communication

The client-sever to protocol mappings maps client-sever interface communication to system signals via an embedded reference to a communication protocol

```
client_server_to_protocol_mappings ::=
    <CLIENT-SERVER-TO-PROTOCOL-MAPPING>
    <CS-PARAMETER-MAPPINGS>
    + client_server_parameter_mapping
    <CS-PARAMETER-MAPPINGS>
    <MAPPED-OPERATION-IREF>
    instance_ref
```

```
    </MAPPED-OPERATION-IREF>
    </CLIENT-SERVER-TO-PROTOCOL-MAPPING>
```

The <MAPPED-OPERATION-IREF> instance reference requires three context references (the component prototype, the port prototype within the component prototype and the operation within the interface categorizing the port) and a target reference (the argument within the operation).

```
client_server_parameter_mapping ::=
    <CLIENT-SERVER-PARAMETER-MAPPING>
    <MAPPED-PARAMETER-IREF>
    instance_ref
    </MAPPED-PARAMETER-IREF>
    <USED-COMM-PROTO-SIGNAL-IREF>
    instance_ref
    </USED-COMM-PROTO-SIGNAL-IREF>
    </CLIENT-SERVER-PARAMETER-MAPPING>
```

The <MAPPED-PARAMETER-IREF> instance reference requires three context references (the component prototype, the port prototype within the component prototype and the operation within the interface categorizing the port) and a target reference (the argument within the operation).

The <USED-COMM-PROTO-SIGNAL-IREF> instance reference merely requires a target reference; no context references are required. The target reference must refer to a <COMM-PROTOCOL-SIGNAL-ROLE> defined in the input.

Client-Server Communication

The client-sever to signal mappings map client-server interface communication to system signals in a way similar to the mapping of sender-receiver communications. In the client-server case, a signal group is always used, holding not only the signals for the parameters of the operation, but also those for various elements of meta-data, such as sequence counters.

```
client_server_mapping ::=
    <CLIENT-SERVER-TO-SIGNAL-GROUP-MAPPING>
    ( application_errors )
    ( composite_type_mappings )
    ( empty_signal )
    mapped_operation_ref
    ( primitive_type_mappings )
    ( request_group_ref | response_group_ref )
    ( sequence_counter )
    </CLIENT-SERVER-TO-SIGNAL-GROUP-MAPPING>

mapped_operation_ref ::=
    <MAPPED-OPERATION-IREF>
        component_prototype_ref
    <PORT-PROTOTYPE-REF>ref</PORT-PROTOTYPE-REF>
    <OPERATION-REF>ref</OPERATION-REF>
    </MAPPED-OPERATION-IREF>
```

The <MAPPED-OPERATION-IREF> instance reference contains elements: the component prototype, the port prototype within the component prototype and the operation within the interface categorizing the port.

```
application_errors ::=
    <APPLICATION-ERRORS>
    *application_error_mapping
    </APPLICATION-ERRORS >

application_error_mapping ::=
    <APPLICATION-ERROR-MAPPING>
    system_signal_ref
    </APPLICATION-ERROR-MAPPING>
```

The <APPLICATION-ERRORS> element is used to provide a signal to carry an error code from the server back to the client.

```
empty_signal ::=
    <EMPTY-SIGNAL>
    system_signal_ref
    </EMPTY-SIGNAL>

sequence_counter ::=
    <SEQUENCE-COUNTER>
    system_signal_ref
    </SEQUENCE-COUNTER >

request_group_ref ::=
    <REQUEST-GROUP-REF>ref</REQUEST-GROUP-REF >

response_group_ref ::=
    <RESPONSE-GROUP-REF>ref</RESPONSE-GROUP-REF >
```

Either a <REQUEST-GROUP-REF> or a <RESPONSE-GROUP-REF> must be specified in a mapping (but not both), and for any given operation, there must be mappings containing each of these elements. They identify the signal group used for atomic transmission of all the arguments and other data associated with a client-server call or response.

The <PRIMITIVE-TYPE-MAPPING> and <COMPOSITE-TYPE-MAPPING> elements are used to collect signal mappings for the simple and complex arguments, respectively, of a client-server operation. They include a range of XML elements very similar to those used for sender-receiver data mappings.

```
primitive_type_mappings ::=
    <PRIMITIVE-TYPE-MAPPING>
    *primitive_type_mapping
    </PRIMITIVE-TYPE-MAPPING>

primitive_type_mapping ::=
    argument_ref
    system_signal_ref
```

The argument_ref reference identifies the argument to which the mapping applies.

```
argument_ref ::=
    <ARGUMENT-REF>ref</ARGUMENT-REF>

system_signal_ref ::=
    <SYSTEM-SIGNAL-REF>ref</SYSTEM-SIGNAL-REF>
```

A <COMPOSITE-TYPE-MAPPING> element is used to collect signal mappings for complex arguments (records and arrays) of a client-server operation.

```
composite_type_mappings ::=
    <COMPOSITE-TYPE-MAPPINGS>
    *( cs_record_type_mapping | cs_array_type_mapping )
    </COMPOSITE-TYPE-MAPPINGS>
```

Since there can be multiple complex arguments the element encapsulates zero or more mapping elements—one for each argument.

The <CLIENT-SERVER-RECORD-TYPE-MAPPING> element references the client-server argument and contains the mapping to signals.

```
cs_record_type_mapping ::=
    <CLIENT-SERVER-RECORD-TYPE-MAPPING>
    argument_ref
    <RECORD-ELEMENT-MAPPINGS>
    +cs_record_element_mapping
    <RECORD-ELEMENT-MAPPINGS>
    </CLIENT-SERVER-RECORD-TYPE-MAPPING>
```

Each element of the record is mapped using a <CLIENT-SERVER-RECORD-ELEMENT-MAPPING> element.

```
cs_record_element_mapping ::=
    <CLIENT-SERVER-RECORD-ELEMENT-MAPPING>
    ( cs_simple_record_element_mapping |
    cs_complex_record_element_mapping )
    </CLIENT-SERVER-RECORD-ELEMENT-MAPPING>
```

Record elements can be primitive or complex types. For an element which is a primitive type the mapping references a single signal.

```
cs_simple_record_element_mapping ::=
    cs_record_element_ref
    signal_ref
```

For record elements that are themselves complex types a nested <CLIENT-SERVER-RECORD-ELEMENT-MAPPING> element is used. The argument reference for a nested element is ignored.

```
cs_complex_record_element_mapping ::=
    <COMPLEX-TYPE_MAPPING>
    ( cs_record_type_mapping | cs_array_type_mapping )
    </COMPLEX-TYPE_MAPPING>
    cs_record_element_ref
```

The relevant record element is identified by an <RECORD-ELEMENT-REF> element.

```
cs_record_element_ref ::=
    <RECORD-ELEMENT-REF>
    ref
    </RECORD-ELEMENT-REF>
```

Arrays are mapped element-by-element, in a similar way to record, with each element being mapped to a separate signal:

```
cs_array_type_mapping ::=
    <CLIENT-SERVER-ARRAY-TYPE-MAPPING>
        argument_ref
    <ARRAY-ELEMENT-MAPPINGS>
    +cs_array_element_mapping
    </ARRAY-ELEMENT-MAPPINGS>
    </CLIENT-SERVER-ARRAY-TYPE-MAPPING>

cs_array_element_mapping ::=
    <CLIENT-SERVER-ARRAY-ELEMENT-MAPPING>
    ( cs_simple_array_element_mapping |
    cs_complex_array_element_mapping )
    </CLIENT-SERVER-ARRAY-ELEMENT-MAPPING>

cs_simple_array_element_mapping ::=
    cs_array_element_ref
    signal_ref

cs_complex_array_element_mapping ::=
    <COMPLEX-TYPE_MAPPING>
    cs_record_type_mapping
    </COMPLEX-TYPE_MAPPING>
    cs_array_element_ref
```

Note the restriction here that an array may not contain nested array-type elements.

```
cs_array_element_ref ::=
    <CLIENT-SERVER-ARRAY-ELEMENT-MAPPING>
    <INDEXED-ARRAY-ELEMENT>
    <ARRAY-ELEMENT-REF>
    ref
    </ARRAY-ELEMENT-REF>
    <INDEX>int</INDEX>
    </INDEXED-ARRAY-ELEMENT>
    signal_ref
    </CLIENT-SERVER-ARRAY-ELEMENT-MAPPING>
```

An array element reference consists of a software component instance within a composition, a port, and an operation argument. The <INDEX> gives the position of the mapped element within the array; the <ARRAY-ELEMENT-REF> refers to the type of the array element, and should therefore have the same value for all the mapped elements of a given array.

4.18.3 Software Component to Implementation Mapping

A “software-component implementation mapping” maps software component prototypes to specific implementations. RTA-RTE uses the implementation to determine if a SWC is implemented as source or object code.

```
swc_to_impl_mapping ::=
    <SW-IMPL-MAPPINGS>
    <SWC-TO-IMPL-MAPPING>
    short_name
        COMPONENT-IMPLEMENTATION-REF>ref</COMPONENT-IMPLEMENTATION-REF>
    <COMPONENT-IREFS>
    +component_prototype_iref
    </COMPONENT-IREFS>
    </SWC-TO-IMPL-MAPPING>
    </SW-IMPL-MAPPINGS>
```

A single SWC-to-implementation mapping can associate multiple component instances with the same implementation.

4.18.4 Software Component to ECU Mapping

A “software-component to ECU” maps software component prototypes to ECU instances.

```
swc_mappings ::=
    <SW-MAPPINGS>
    *swc_to_ecu_mapping
    <SW-MAPPINGS>
```

An SWC mapping defines neither the component prototypes nor the ECU instances—it merely contains references to other elements that associate component prototypes with ECU instances.

```
swc_to_ecu_mapping ::=
    <SWC-TO-ECU-MAPPING>
    short_name
    <COMPONENT-IREFS>
    +component_prototype_iref
    </COMPONENT-IREFS>
    ecu_instance_iref
    </SWC-TO-ECU-MAPPING>
```

A single <SWC-TO-ECU-MAPPING> element can map multiple SWC prototypes to an ECU instance. Each mapped prototype requires a <COMPONENT-IREF> element.

```
component_prototype_iref ::=
    <COMPONENT-IREF>
    instance_ref
    </COMPONENT-IREF>
```

The <COMPONENT-IREF> instance reference defines the component prototype to be mapped and requires at least one <SOFTWARE-COMPOSITION-REF> reference to the

top level composition, zero or more <COMPONENT-PROTOTYPE-REF> references to component prototypes that form levels within the composition hierarchy and finally one <TARGET-COMPONENT-PROTOTYPE-REF> that specifies the SWC prototype.

The software-components are mapped to an ECU instance defined via the <ECU-INSTANCE-REF> element:

```
ecu_instance_iref ::=
    <ECU-INSTANCE-REF>
    instance_ref
    </ECU-INSTANCE-REF>
```

The <ECU-INSTANCE-REF> reference defines the ECU instance to which the component prototypes are mapped.

4.18.5 Software Composition Instance

A “software-component to ECU” mapping within a system description element identifies software component prototypes via a reference to a software composition instance.

```
sw_compositon ::=
    <SOFTWARE-COMPOSITION>
    short_name
    <SOFTWARE-COMPOSITION-TREF>
    ref
    </SOFTWARE-COMPOSITION-TREF>
    </SOFTWARE-COMPOSITION>
```

The reference within the <SOFTWARE-COMPOSITION-TREF> refers to the system’s top-level composition—this is the self-contained (i.e. does not delegate any ports) composition that contains all SWC instances mapped within the System element’s component mapping.

4.19 ECU Description

The ECU Description is used to define the module configuration for the RTE (as well as the configuration for many other AUTOSAR modules).

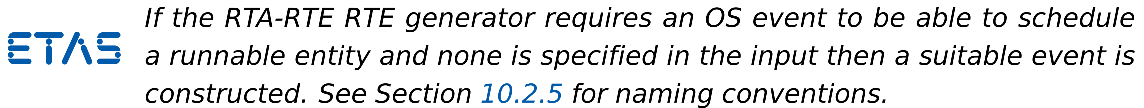
The top-level contain for module configuration within the ECU Description is an XML element called a <MODULE-CONFIGURATION>. within a module’s configuration an XML element called a <CONTAINER> is used to describe different configuration aspects.

The type of a module configuration or container element is identified by a <DEFINITION-REF> element that defines a “path”; for AUTOSAR defined definition references this element contains a path that always starts with /AUTOSAR. A container can encapsulate other (sub)containers and thus a hierarchy of configuration containers is formed.

For brevity, within the following section the ‘path’ of the encapsulated containers is abbreviated such that the common prefix (e.g. /AUTOSAR/Rte) is replaced with ‘...’ when describing containers located at the same hierarchy level.

4.19.1 OS Module

AUTOSAR OS Configuration occurs within an instance of the container type `/AUTOSAR/Os`. RTA-RTE can require access to the defined OS tasks, resources and events.



If the RTA-RTE RTE generator requires an OS event to be able to schedule a runnable entity and none is specified in the input then a suitable event is constructed. See Section 10.2.5 for naming conventions.

4.19.2 RTE Module

Configuration of the RTE module occurs within an instance of the container type `/AUTOSAR/Rte`. The RTE configuration container can contain one or more sub-containers:

- `.../CommonPublishedInformation`—Defines the AUTOSAR version and RTA-RTE version for which the information within the file is appropriate. The RTA-RTE RTE generator verifies that the supplied Common Published Information is appropriate.
- `.../RTEGeneration`—Defines the generation parameters for RTA-RTE including the operating mode, optimization strategy and whether or not VFB Trace Hooks are generated.
- `.../SwComponentInstance`—The parent container for describing configuration of a SW-C instance. Further sub-containers defines the mapping of one or more runnable entity instances to OS task(s) and the implementation method to be used for exclusive areas. The SW-C instance container can optionally describe the association between an software-component instance and its implementation.
- `.../CalprmComponentInstance`—Disables calibration support for a specified application software-component or calibration component type.

The relationship between containers in the Rte's module configuration container is described in Figure 4.2.

Container `CommonPublishedInformation`

Configuration of version information occurs within an instance of the container type `/AUTOSAR/Rte/CommonPublishedInformation`.

The Common Published Information container contains eight integer values:

- `.../ArMajorVersion`—An integer defining the 'major' part of the supported AUTOSAR version.
- `.../ArMinorVersion`—An integer defining the 'minor' part of the supported AUTOSAR version.
- `.../ArMajorVersion`—An integer defining the 'patch' revision of the supported AUTOSAR version.

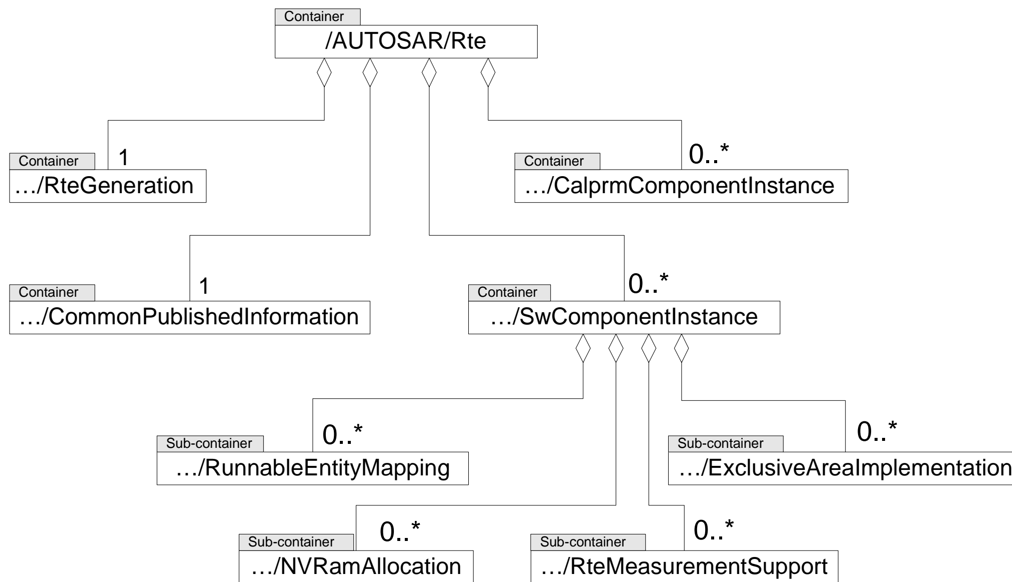


Figure 4.2: Top-level RTE Configuration Containers

- `.../ModuleId`—Not used by RTA-RTE.
- `.../SwMajorVersion`—An integer defining the 'major' part of the supported RTA-RTE release.
- `.../SwMinorVersion`—An integer defining the 'minor' part of the supported RTA-RTE release.
- `.../SwPatchVersion`—An integer defining the 'patch' revision of the supported RTA-RTE release.
- `.../VendorId`—Not used by the RTA-RTE RTE generator.

The supported values for each integer parameter are defined in Section 11.1. If the `CommonPublishedInformation` container is present in the input the contained values are compared against the expected values and an error emitted if inconsistent.

Container `RTEGeneration`

Configuration of RTE generation parameters occurs within an instance of the container type `/AUTOSAR/Rte/RteGeneration`. The RTE generation container can contain one or more containers:

- `.../RteGenerationMode`—Defines the generation mode used; acceptable enumeration values are `COMPATIBILITY_MODE` and `VENDOR_MODE`.
- `.../RteVfbTrace`—Enables (non-zero) or disables (zero) creation of VFB trace hook calls in the generated RTE. Note that even when creation of hook calls is enabled it remains necessary to enable/disable the use of individual hooks when the RTE is compiled.

The generation of VFB trace hooks can be enabled/disabled using either the `--vfb-trace` command-line option or via the RTEGeneration container. If the optimization mode is specified both within RTE parameters and on the command-line the latter specification takes precedence.

The specification of this parameter is optional; if omitted RTA-RTE will create VFB trace hook calls.

- `.../RteOptimizationMode`—Defines the optimization strategy for RTE generation; acceptable enumeration values are “`RUNTIME`” and “`MEMORY`”.

  The optimization mode can be selected using either the `--optimize` command-line option or via the RTEGeneration container. If the optimization mode is specified both within RTE parameters and on the command-line the latter specification takes precedence.

  The optimization strategy parameter is optional; if omitted RTA-RTE defaults to the “runtime” strategy.

In addition to the containers common to all AUTOSAR releases described above, the following additional containers can be included within the `RteGeneration` container.

- `.../RteMeasurementSupport`—Enables (“`true`”) or disables (“`false`”) measurement support within the generated RTE.

  The specification of this parameter is optional; if omitted measurement support within RTA-RTE is enabled.

- `.../RteCalibrationSupport`—Defines the global calibration method; acceptable enumeration values are “`NONE`”, “`SINGLE_POINTERED`”, “`DOUBLE_POINTERED`” and “`INITIALIZED_RAM`”.

  The calibration method can be selected using either the `--calibration-method` command-line option or via the RTEGeneration container. If the calibration method is specified both within RTE parameters and on the command-line the latter specification takes precedence.

  The calibration method parameter is optional; if omitted RTA-RTE defaults to the “single pointered” method.

The RTE module configuration must contain at most one RTE generation container.

Sub-container `RteForceBasicTask`

RTA-RTE uses a **sub-container** within the `RteGeneration` container to enable the selection of forced-basic semantics (see *RTA-RTE User Guide*) for individual tasks.

The specification of the `RteForceBasicTask` container is a custom RTA-RTE feature and is not part of the AUTOSAR standards.

The `RteForceBasicTask` sub-container has definition reference `/RTARTE/Rte/RteGeneration/RteForceBasicTask` and may occur zero or more times within the RTE’s module configuration

The RteForceBasicTask sub-container references an OsTask container within the application OS module configuration and switches on or off the forced-basic semantics for that task by means of a boolean parameter /RTARTE/Rte/RteGeneration/RteForceBasicTask/OverrideValue.

It is an error to specify multiple RteForceBasicTask sub-containers referring to the same task unless all the containers for the task also specify the same OverrideValue.

An example of this vendor-specific container is shown below. For clarity the RteGeneration parameters have been omitted and the definition refernces shortened.

```
<CONTAINER>
  <SHORT-NAME>...</SHORT-NAME>
  <DEFINITION-REF>/AUTOSAR/Rte/RteGeneration</DEFINITION-REF>
  <PARAMETER-VALUES>
    ...
  </PARAMETER-VALUES>
  <SUB-CONTAINERS>
    <CONTAINER>
      <SHORT-NAME>...</SHORT-NAME>
      <DEFINITION-REF>
        /RTARTE/Rte/RteGeneration/RteForceBasicTask
      </DEFINITION-REF>
      <PARAMETER-VALUES>
        <BOOLEAN-VALUE>
          <DEFINITION-REF>.../OverrideValue</DEFINITION-REF>
          <VALUE>true</VALUE>
        </BOOLEAN-VALUE>
      </PARAMETER-VALUES>
      <REFERENCE-VALUES>
        <REFERENCE-VALUE>
          <DEFINITION-REF>.../TaskRef</DEFINITION-REF>
          <VALUE-REF>/pkg/Os/example_task</VALUE-REF>
        </REFERENCE-VALUE>
      </REFERENCE-VALUES>
    </CONTAINER>
  </SUB-CONTAINERS>
</CONTAINER>
```

The names of RteForceBasicTask sub-containers are not used by RTA-RTE other than to ensure uniqueness.

Container SwComponentInstance

The /AUTOSAR/Rte/SwComponentInstance container defines the configuration of a single SW-C instance including the mapping of a runnable entity instance to OS tasks and the specification of exclusive area implementation method.

The SwComponentInstance container can specify one or more sub-containers that define runnable entity mapping, specify how an exclusive area should be implemented or define NVRAM allocation.

Component Prototype Selection

The `SwComponentInstance` container references the appropriate SW-C prototype using `SoftwareComponentInstanceRef`. This element contains a <`VALUE-IREF`> element refering to the software component prototype in the context of its composition.

RTA-RTE requires that the <`VALUE-IREF`> element contain a set of absolute references to the SW-C instance (i.e. the component prototype). When a single level exists in the composition hierachy then the SoftwareComponentInstanceRef can be a single <`VALUE-REF`>. However when nested compositions are used the reference **must** include each level of the nesting as a <`CONTEXT-REF`> and must terminate with a <`VALUE-REF`> that references a component-prototype.

When a `/AUTOSAR/Rte/SwComponentInstance` container references a service component it should use a `ServiceComponentPrototypeRef` rather than a `SoftwareComponentInstanceRef`.

The `ServiceComponentPrototypeRef` is specified as a reference to the component prototype. Service components cannot be present within a nested composition and therefore only a single <`XML-TAG`> is required.

The `SwComponentInstance` must specify exactly one `SoftwareComponentInstanceRef` or `ServiceComponentPrototypeRef`.

Implementation Selection

The `SwComponentInstance` container can optionally reference an implementation element using an `.../ImplementationRef`. when specified this selects an implementation element to associate with the SW-C instance. The implementation is specified as an absolute reference.

A SW-C instance can only be associated with a single implementation.

Runnable entity Mapping

Each runnable entity mapping sub-container maps a single RTE Event (instance and hence triggered runnable entity instance) to a task. Each `RunnableEntityMapping` defines the following parameter values:

- `.../PositionInTask`—The position of the runnable entity within the task (this must be specified as an unsigned integer).

  To prevent ambiguities in runnable ordering, all runnable entities mapped to a task should have unique positions within the task. The positions specified for runnable entities within a task need not be contiguous.

- `.../RTEEventRef`—The RTE Event responsible for activating the runnable entity name. This must be specified as an absolute reference to the RTE Event within the SW-C internal behavior definition

- `.../MappedToTaskRef`—The OS task to which the runnable entity is mapped. This must be specified as an absolute reference to the OS Task definition within the OS configuration container.
- `.../OsEventRef`—The OS event to be used by the RTA-RTE RTE generator (if required) to schedule the runnable entity.

  ETAS *The specification of an OsEvent is optional. If the RTA-RTE RTE generator does not require an `OsEvent` to be used to schedule the runnable then it will be ignored. If an OsEvent is required but not specified then the RTE generator will construct a suitable event.*
- `.../ActivationOffset`—The offset (in seconds) from the period. This parameter is only applicable for TimingEvents.

The input information must contain one runnable entity mapping container for each RTE Event that starts a runnable entity. There is no need to map RTE Events that do not trigger a runnable.

Exclusive Area Implementation

As well as runnable entity mapping, the `SwComponentInstance` container can specify one or more implementation method specifications.

Each `ExclusiveAreaImplementation` sub-container specifies the implementation of a single exclusive area in the context of a SwcInstance. The sub-container provides a mechanism to specify the implementation method (OS resource or interrupt blocking) and, if applicable, the OS resource, used for an exclusive area instance.

- `.../ExclusiveAreaRef`—The exclusive area to which the implementation method applies.

  The referenced exclusive area **must** be declared within the internal behavior associated with the SW-C instance.
- `.../ExclusiveAreaImplMechanism`—The implementation method to be used for the referenced exclusive area. Acceptable enumeration values are: "`INTERRUPT_BLOCKING`", "`NON_PREEMPTIVE_TASKS`", "`OS_RESOURCE`" and "`COOPERATIVE_RUNNABLE_PLACEMENT`".

  If the specification of the implementation method is omitted for a SW-C instance, RTA-RTE will assume that OS resources should be used.
- `.../ExclusiveAreaOsResourceRef`—This is an optional, vendor-specific, reference that specifies the `OsResource` to be used to implement the referenced Exclusive Area.

  When present, the `.../ExclusiveAreaOsResourceRef` **must** refer to an `OsResource` container. If absent, then the default OS resource created by RTA-RTE is used. If the exclusive area does not require an OS resource, for example it's use is optimized away by RTA-RTE, then the parameter is disregarded.

The DEFINITION-REF of the parameter is /RTARTE/Rte/SwComponentInstance/-ExclusiveAreaImplementation/ExclusiveAreaOsResourceRef

Because this is a vendor-specific parameter, its DEFINITION-REF starts /RTARTE/ not /AUTOSAR/.

An example of the ExclusiveAreaOsResourceRef parameter is shown below. For clarity the standard AUTOSAR definition references for the ExclusiveAreaImplementation have been abbreviated.

```
<CONTAINER>
  <SHORT-NAME>EAImplementation1</SHORT-NAME>
  <DEFINITION-REF>
    /AUTOSAR/.../ExclusiveAreaImplementation
  </DEFINITION-REF>
  <PARAMETER-VALUES>
    <ENUMERATION-VALUE>
      <DEFINITION-REF>/AUTOSAR/.../ExclusiveAreaImplMechanism</DEFINITION-
          REF>
      <VALUE>OS_RESOURCE</VALUE>
    </ENUMERATION-VALUE>
  </PARAMETER-VALUES>
  <REFERENCE-VALUES>
    <REFERENCE-VALUE>
      <DEFINITION-REF>/AUTOSAR/.../ExclusiveAreaRef</DEFINITION-REF>
      <VALUE-REF>/pkg/ibswc/cr1</VALUE-REF>
    </REFERENCE-VALUE>
    <REFERENCE-VALUE>
      <DEFINITION-REF>
        /RTARTE/Rte/SwComponentInstance/ExclusiveAreaImplementation/
            ExclusiveAreaOsResourceRef
      </DEFINITION-REF>
      <VALUE-REF>/pkg/Os/MyResource</VALUE-REF>
    </REFERENCE-VALUE>
  </REFERENCE-VALUES>
</CONTAINER>
```

NVRam Allocation

As well as runnable entity mapping, the SwComponentInstance container can specify one or more NVRAM allocation specifications.

Each NVRamAllocation sub-container specifies the implementation of a single exclusive area and defines the following parameter values:

- .../RamBlockLocationSymbol—The name (C identifier) associated with the NVRAM block.
- .../RomBlockLocationSymbol—The name (C identifier) of the ROM initializer associated with the NVRAM block.

This parameter is not used by this release of RTA-RTE.

- `.../SwNvRamMappingReference`—An absolute reference to the NVRAM mapping (within the SW-C instances internal behavior).
- `.../NvmBlockRef`—Associated NVRAM block definition (within the ECUC configuration for the NVRAM manager)

Calibration Disable

RTA-RTE uses the `CalprmComponentInstance` container to disable calibration support for a specified SW-C type.

Each `CalprmComponentInstance` container disables calibration support for a single AUTOSAR application SW-C or Calibration component type.

The container encapsulates a Boolean parameter and references a SW-C type. The parameter determines whether calibration support is enabled (“`true`”) or disabled (“`false`”) for the referenced SW-C.

- `.../CalprmComponentInstanceRef`—The SW-C type, specified as an absolute reference.
- `.../CalibrationSupportEnabled`—Boolean indicating whether or not calibration support is enabled.

Calibration support is enabled by default and therefore a `CalprmComponentInstance` container is only necessary to disable support to a specified SW-C type.

The RTE Generation parameter `RteCalibrationSupport` can be used to globally disable calibration support by selecting the calibration method “none”.

4.20 Vendor Specific XML Extensions

There are no vendor-specific XML extensions.

4.21 Post-build

The following configuration options are available after an RTE has been generated.

4.21.1 Atomicity

The generated RTE uses different mechanisms for ensuring that data reads and writes are atomic depending on the actual size of the data item. The atomicity mechanism can be configured, after the RTE is generated (post-build), to disable code generation for certain data types when it is known that the underlying hardware already provides the required atomicity, e.g. for 8-bit data types.

- `RTE_<SIZE>_ATOMIC`—if defined, the generated RTE assumes that integer variables of size <SIZE> can be read and/or written atomically.

- `RTE_<SIZE>_NONATOMIC`—if defined, the generated RTE assumes that integer variables of size <SIZE> cannot be read and/or written atomically.

Where <SIZE> corresponds to the data item size in bits. Valid values are `8BIT`, `16BIT`, `32BIT`, `64BIT`, `FLOAT32`, `FLOAT64` and `BOOLEAN`.

The default of the generated RTE is to assume that only 8-bit integers can be read/written atomically and that all other types must be protected. If the default behavior is not required then definitions can be placed on the command line when invoking the C compiler to modify compilation of `Rte.c` and SWC implementations, for example:

```
$(CC) --DRTE_16BIT_ATOMIC --DRTE_32BIT_ATOMIC Rte.c
```

4.22 Variability

Version 5.1 of RTA-RTE introduces support for *pre-build variability*. Variability allows elements of the configuration to be enabled or disabled, or values in the configuration to be written as expressions rather than literal numbers. The containers for which RTA-RTE supports variability are listed in subsection 4.22.1 whilst the AUTOSAR Formula Language (AFL) is described in subsection 4.22.2.

4.22.1 Containers Supporting Pre-Build Variability

The following table lists the containers which RTA-RTE checks for pre-build variation during the processing of input XML.

Container
APPLICATION-PRIMITIVE-DATA-TYPE
APPLICATION-RECORD-DATA-TYPE
APPLICATION-SW-COMPONENT-TYPE
ARRAY-SIZE
ASYNCHRONOUS-SERVER-CALL-POINT
ASYNCHRONOUS-SERVER-CALL-RESULT-POINT
ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT
BACKGROUND-EVENT
BSW-BACKGROUND-EVENT
BSW-CALLED-ENTITY
BSW-EXTERNAL-TRIGGER-OCCURRED-EVENT
BSW-IMPLEMENTATION
BSW-INTERNAL-TRIGGER-OCCURRED-EVENT
BSW-INTERNAL-TRIGGERING-POINT
BSW-INTERRUPT-ENTITY
BSW-MODE-RECEIVER-POLICY
BSW-MODE-SENDER-POLICY

Container
BSW-MODE-SWITCH-EVENT
BSW-MODE-SWITCHED-ACK-EVENT
BSW-MODULE-DESCRIPTION
BSW-MODULE-ENTRY
BSW-SCHEDULABLE-ENTITY
BSW-TIMING-EVENT
BSW-TRIGGER-DIRECT-IMPLEMENTATION
CALIBRATION-PARAMETER-VALUE-SET
CLIENT-SERVER-ARRAY-ELEMENT-MAPPING
CLIENT-SERVER-ARRAY-TYPE-MAPPING
CLIENT-SERVER-INTERFACE
CLIENT-SERVER-INTERFACE-MAPPING
CLIENT-SERVER-OPERATION
CLIENT-SERVER-OPERATION-MAPPING
CLIENT-SERVER-PRIMITIVE-TYPE-MAPPING
CLIENT-SERVER-RECORD-ELEMENT-MAPPING
CLIENT-SERVER-RECORD-TYPE-MAPPING
CLIENT-SERVER-TO-SIGNAL-GROUP-MAPPING
COMPLEX-DEVICE-DRIVER-SW-COMPONENT-TYPE
COMPLEX-TYPE-MAPPING
COMPOSITION-SW-COMPONENT-TYPE
COMPU-METHOD
CONSTANT-SPECIFICATION
CONSTANT-SPECIFICATION-MAPPING
CONSTANT-SPECIFICATION-MAPPING-SET
DATA-CONSTR
DATA-PROTOTYPE-MAPPING
DATA-RECEIVE-ERROR-EVENT
DATA-RECEIVED-EVENT
DATA-SEND-COMPLETED-EVENT
DATA-TYPE-MAPPING-SET
DATA-TYPE-POLICY
DATA-WRITE-COMPLETED-EVENT
ECU
ECU-ABSTRACTION-SW-COMPONENT-TYPE
ECU-INSTANCE

Container
ECU-SW-COMPOSITION
ECUC-CONTAINER-VALUE
ECUC-VALUE-COLLECTION
EXCLUSIVE-AREA
EXTERNAL-TRIGGER-OCCURRED-EVENT
EXTERNAL-TRIGGERING-POINT
FILTER
FLAT-INSTANCE-DESCRIPTOR
FLAT-MAP
I-SIGNAL
I-SIGNAL-GROUP
I-SIGNAL-TO-I-PDU-MAPPING
IMPL-INIT-VALUE
IMPLEMENTATION-DATA-TYPE
IMPLEMENTATION-DATA-TYPE-ELEMENT
INCLUDED-DATA-TYPE-SET
INDEX
INDEXED-ARRAY-ELEMENT
INTERNAL-TRIGGER-OCCURRED-EVENT
INTERNAL-TRIGGERING-POINT
IS-REENTRANT y?
IS-SERVICE y?
MODE-ACCESS-POINT
MODE-DECLARATION-GROUP
MODE-DECLARATION-GROUP-PROTOTYPE
MODE-GROUP
MODE-INTERFACE-MAPPING
MODE-MAPPING
MODE-REQUEST-TYPE-MAP
MODE-SWITCH-INTERFACE
MODE-SWITCH-POINT
MODE-SWITCHED-ACK
MODE-SWITCHED-ACK-EVENT
NETWORK-REPRESENTATION
NV-BLOCK-DATA-MAPPING
NV-BLOCK-DESCRIPTOR

Container
NV-BLOCK-SW-COMPONENT-TYPE
NV-DATA-INTERFACE
OPERATION-INVOKED-EVENT
P-PORT-PROTOTYPE
PACKING-BYTE-ORDER
PARAMETER-ACCESS
PARAMETER-DATA-PROTOTYPE
PARAMETER-INTERFACE
PARAMETER-SW-COMPONENT-TYPE
PER-INSTANCE-MEMORY
PHYSICAL-DIMENSION
PORT-API-OPTION
QUEUED-RECEIVER-COM-SPEC
R-PORT-PROTOTYPE
RAM-BLOCK
ROOT-SW-COMPOSITION-PROTOTYPE
RUNNABLE-ENTITY
SENDER-REC-ARRAY-ELEMENT-MAPPING
SENDER-REC-ARRAY-TYPE-MAPPING
SENDER-REC-RECORD-ELEMENT-MAPPING
SENDER-REC-RECORD-TYPE-MAPPING
SENDER-RECEIVER-INTERFACE
SENDER-RECEIVER-TO-SIGNAL-GROUP-MAPPING
SENDER-RECEIVER-TO-SIGNAL-MAPPING
SENSOR-ACTUATOR-SW-COMPONENT-TYPE
SERVER-ARGUMENT-IMPL-POLICY
SERVICE-COMPONENT-PROTOTYPE
SERVICE-SW-COMPONENT-TYPE
START-POSITION
SUPPORTS-ASYNCHRONOUS-MODE-SWITCH
SW-ADDR-METHOD
SW-BASE-TYPE
SW-COMPONENT-PROTOTYPE
SW-DATA-DEF-PROPS-CONDITIONAL
SW-SYSTEMCONST
SW-SYSTEMCONST-VALUE

Container
SW-SYSTEMCONSTANT-VALUE-SET
SWC-BSW-MAPPING
SWC-BSW-RUNNABLE-MAPPING
SWC-BSW-SYNCHRONIZED-MODE-GROUP-PROTOTYPE
SWC-BSW-SYNCHRONIZED-TRIGGER
SWC-IMPLEMENTATION
SWC-MODE-SWITCH-EVENT
SWC-SERVICE-DEPENDENCY
SWC-TO-ECU-MAPPING
SWC-TO-IMPL-MAPPING
SYNCHRONOUS-SERVER-CALL-POINT
SYSTEM
SYSTEM-MAPPING
SYSTEM-SIGNAL
SYSTEM-SIGNAL-GROUP
TASK
TEXT-TABLE-MAPPING
TEXT-TABLE-VALUE-PAIR
TIMING-EVENT
TRIGGER
TRIGGER-INTERFACE
TRIGGER-INTERFACE-MAPPING
TRIGGER-MAPPING
TYPE
TYPE-MAPPING
UNIT
USED-DATA-ELEMENT
V
VARIABLE-ACCESS
VARIABLE-AND-PARAMETER-INTERFACE-MAPPING
VARIABLE-DATA-PROTOTYPE

4.22.2 AUTOSAR Formula Language

Whilst the AUTOSAR Formula Language (AFL) is described fully in Section 4.8 of the AUTOSAR Generic Structure Template, the manner in which it is integrated into the XML is not. Here are some examples of how expressions are written.

Simple Number

```
...
  <VARIATION-POINT>
    <SHORT-LABEL>vpbe1Pct</SHORT-LABEL>
    <SW-SYSCOND BINDING-TIME='CODE-GENERATION-TIME'>1</SW-SYSCOND>
  </VARIATION-POINT>
...
```

In this example, the result of the <SW-SYSCOND> expression is simply the number *1*.

Simple Expression

```
...
  <VARIATION-POINT>
    <SHORT-LABEL>vpbe1Pct</SHORT-LABEL>
    <SW-SYSCOND BINDING-TIME='CODE-GENERATION-TIME'>1 + 4</SW-SYSCOND>
  </VARIATION-POINT>
...
```

In this example, the result of the <SW-SYSCOND> expression is the number *5* - the result of adding 1 and 4.

Other arithmetic operators are available (+, -, *, /).

Cross-Reference

```
...
<AR-PACKAGE>
  <SHORT-NAME>system_constants</SHORT-NAME>
  <ELEMENTS>
    <SW-SYSTEMCONST>
      <SHORT-NAME>switch_on</SHORT-NAME>
    </SW-SYSTEMCONST>
    <SW-SYSTEMCONSTANT-VALUE-SET>
      <SHORT-NAME>scvs</SHORT-NAME>
      <SW-SYSTEMCONSTANT-VALUES>
        <SW-SYSTEMCONST-VALUE>
          <SW-SYSTEMCONST-REF DEST='SW-SYSTEMCONST'>/system_constants/
              switch_on</SW-SYSTEMCONST-REF>
          <VALUE>1</VALUE>
        </SW-SYSTEMCONST-VALUE>
      </SW-SYSTEMCONSTANT-VALUES>
    </SW-SYSTEMCONSTANT-VALUE-SET>
  </ELEMENTS>
</AR-PACKAGE>
...
  <VARIATION-POINT>
    <SHORT-LABEL>vpbe1Pct</SHORT-LABEL>
    <SW-SYSCOND BINDING-TIME='CODE-GENERATION-TIME'>
      <SYSC-REF DEST='SW-SYSTEMCONST'>/system_constants/switch_on</SYSC-REF
          >
    </SW-SYSCOND>
  </VARIATION-POINT>
```

```
...
```

In this example, the result of the <SW-SYSCOND> expression is the number *1* - since that is the value of the system constant (`switch_on`) which it references.

<VALUE>s contained within <SW-SYSTEMCONST>s are themselves expressions, and can therefore reference other <SW-SYSTEMCONST>s.

Boolean Tests

```
...
<AR-PACKAGE>
  <SHORT-NAME>system_constants</SHORT-NAME>
  <ELEMENTS>
    <SW-SYSTEMCONST>
      <SHORT-NAME>num_cylinders</SHORT-NAME>
    </SW-SYSTEMCONST>
    <SW-SYSTEMCONSTANT-VALUE-SET>
      <SHORT-NAME>scvs</SHORT-NAME>
      <SW-SYSTEMCONSTANT-VALUES>
        <SW-SYSTEMCONST-VALUE>
          <SW-SYSTEMCONST-REF DEST='SW-SYSTEMCONST'>/system_constants/
              num_cylinders</SW-SYSTEMCONST-REF>
          <VALUE>6</VALUE>
        </SW-SYSTEMCONST-VALUE>
      </SW-SYSTEMCONSTANT-VALUES>
    </SW-SYSTEMCONSTANT-VALUE-SET>
  </ELEMENTS>
</AR-PACKAGE>
...
  <VARIATION-POINT>
    <SHORT-LABEL>vpbe1Pct</SHORT-LABEL>
    <SW-SYSCOND BINDING-TIME='CODE-GENERATION-TIME'>
      <SYSC-REF DEST='SW-SYSTEMCONST'>/system_constants/num_cylinders</SYSC
          -REF> == 4
    </SW-SYSCOND>
  </VARIATION-POINT>
...
```

In this example, the expression compares the value of the `num_cylinders` <SW-SYSTEMCONST> against the literal *4*. The result of this comparison is the number *0*, indicating FALSE.

Such an expression might be used to enable configuration which is only relevant for four-cylinder engines.

VALUE substitution

```
...
<AR-PACKAGE>
  <SHORT-NAME>system_constants</SHORT-NAME>
  <ELEMENTS>
```

```
      <SW-SYSTEMCONST>
        <SHORT-NAME>num_cylinders</SHORT-NAME>
      </SW-SYSTEMCONST>
      <SW-SYSTEMCONSTANT-VALUE-SET>
        <SHORT-NAME>scvs</SHORT-NAME>
        <SW-SYSTEMCONSTANT-VALUES>
          <SW-SYSTEMCONST-VALUE>
            <SW-SYSTEMCONST-REF DEST='SW-SYSTEMCONST'>/system_constants/
                num_cylinders</SW-SYSTEMCONST-REF>
            <VALUE>6</VALUE>
          </SW-SYSTEMCONST-VALUE>
        </SW-SYSTEMCONSTANT-VALUES>
      </SW-SYSTEMCONSTANT-VALUE-SET>
    </ELEMENTS>
  </AR-PACKAGE>
  ...
    <IMPLEMENTATION-DATA-TYPE>
      <SHORT-NAME>Array1</SHORT-NAME>
      <CATEGORY>ARRAY</CATEGORY>
      <SUB-ELEMENTS>
        <IMPLEMENTATION-DATA-TYPE-ELEMENT>
          <SHORT-NAME>Array1_element</SHORT-NAME>
          <CATEGORY>VALUE</CATEGORY>
          <ARRAY-SIZE BINDING-TIME="CODE-GENERATION-TIME"><SYSC-REF DEST="SW-
              SYSTEMCONST">/system_constants/num_cylinders</SYSC-REF></ARRAY-
              SIZE>
          <ARRAY-SIZE-SEMANTICS>FIXED-SIZE</ARRAY-SIZE-SEMANTICS>
          ...
```

In this example, the value of a <SW-SYSTEMCONST> is used as a <VALUE> in the definition of an array-type.

The result of the <SW-SYSTEMCONST> might be combined in an arithmetic expression:

```
  ...
    <IMPLEMENTATION-DATA-TYPE>
      <SHORT-NAME>Array1</SHORT-NAME>
      <CATEGORY>ARRAY</CATEGORY>
      <SUB-ELEMENTS>
        <IMPLEMENTATION-DATA-TYPE-ELEMENT>
          <SHORT-NAME>Array1_element</SHORT-NAME>
          <CATEGORY>VALUE</CATEGORY>
          <ARRAY-SIZE BINDING-TIME="PRE-COMPILE-TIME"><SYSC-REF
          DEST="SW-SYSTEMCONST">/system_constants/num_cylinders</SYSC-REF> *
              2</ARRAY-SIZE>
          <ARRAY-SIZE-SEMANTICS>FIXED-SIZE</ARRAY-SIZE-SEMANTICS>
          ...
```

Here, the number of elements in the array is twice the number of cylinders. Since the array size has a BINDING-TIME of PRE-COMPILE-TIME the value is also emitted as a

Condition-Value-Macro in the Configuration header file (rte_sws_6541).

4.23 Support for the atpSplitable Stereotype

Certain containers within the input XML contain aggregates which are marked as 'splittable' (i.e. elements of the aggregations can exist in multiple input files - marked as `atpSplitable` within the Autosar specifications). RTA-RTE now supports all relevant splittable aggregates.

The unsupported (and irrelevant to RTA-RTE) splittable containers are as follows:

- `AliasNameSet.AliasNames`
- `ApplicationSwComponentType.SwComponentDocumentations`
- `CanCluster.PhysicalChannels`
- `CanPhysicalChannel.FrameTriggerings`
- `CanPhysicalChannel.ISignalTriggerings`
- `CanPhysicalChannel.PduTriggerings`
- `ComplexDeviceDriverSwComponentType.SwComponentDocumentations`
- `EcuAbstractionSwComponentType.SwComponentDocumentations`
- `EndToEndProtection.EndToEndProfile`
- `EndToEndProtection.EndToEndProtectionISignalIPdus`
- `EndToEndProtection.EndToEndProtectionVariablePrototypes`
- `EndToEndProtectionSet.EndToEndProtections`
- `EthernetPhysicalChannel.FrameTriggerings`
- `EthernetPhysicalChannel.ISignalTriggerings`
- `EthernetPhysicalChannel.PduTriggerings`
- `FlexrayCluster.PhysicalChannels`
- `FlexrayPhysicalChannel.FrameTriggerings`
- `FlexrayPhysicalChannel.ISignalTriggerings`
- `FlexrayPhysicalChannel.PduTriggerings`
- `LinCluster.PhysicalChannels`
- `LinPhysicalChannel.FrameTriggerings`
- `LinPhysicalChannel.ISignalTriggerings`

- LinPhysicalChannel.PduTriggerings
- NvBlockSwComponentType.SwComponentDocumentations
- ParameterSwComponentType.SwComponentDocumentations
- SensorActuatorSwComponentType.SwComponentDocumentations
- ServiceProxySwComponentType.SwComponentDocumentations
- ServiceSwComponentType.SwComponentDocumentations
- TtCanPhysicalChannel.FrameTriggerings
- TtCanPhysicalChannel.ISignalTriggerings
- TtCanPhysicalChannel.PduTriggerings
- TtcanCluster.PhysicalChannels

5 RTE Conventions

5.1 Name Space

All symbols (e.g. function names, global variables, etc.) created by the RTA-RTE RTE generator that are visible within the global namespace use either the RTE prefix (for definitions), `Rte` (variables, functions names and other symbols) or `rte` (some generated files).

To prevent clashes with symbols created by the RTE generator, application software components should not create symbols in the global namespace using the prefix `rte` (irrespective of case).

The generated RTE is designed to work with different components written in different source languages and therefore does not use language specific features, such as C++ namespaces, to ensure symbol name uniqueness. A component can, however, use language specific features to ensure unique local symbols.

5.2 Software-Component Naming

RTA-RTE does not impose any naming convention on software-components over and above the standard AUTOSAR restrictions:

- The RTE/`Rte` namespace is reserved exclusively for use by RTA-RTE generated symbols.
- Software-component names must only contain characters that are valid both for C-identifiers and for filenames.

  This restriction exists since the name is used to construct symbols within the generated RTE (e.g. component data structure instance name) and the name of the application header file for the software component.
- The names of software component names must be unique.

  An application header file is created for each software component type and therefore type names must be globally unique.

6 RTE API Reference

The functions described in this section are organized by RTE API name as used by C and C++ software components. The API mapping implemented in the application header file used by a software component hides from the software component programmer the need to be aware of the steps taken by the RTE generator to ensure that the generated API functions have unique names.

6.1 API Parameter Passing

The mechanism by which the parameters are passed to the RTE by API calls and from the RTE to runnable entities depends on the parameter class and the type of the parameter being passed.

Parameters are divided into two types:

- **Primitive Types** – AUTOSAR primitive types excluding strings, user defined enumeration types
- **Complex Types** – AUTOSAR string types, user defined record and array types,

Class	Semantics	Primitive Type	Complex Type
IN	Read-only	by Value	by Reference
IN/OUT	Read-Write	by Reference	by Reference
OUT	Write-only	by Reference	by Reference

Table 6.1: RTA-RTE Parameter Passing Conventions

6.2 Data Types

This section defines the RTA-RTE specific data types used by the RTE API. This section does not describe the pre-defined AUTOSAR basic software data types nor does it define how additional AUTOSAR data types can be configured in the XML.

6.2.1 Rte_Instance

A software component instance within an ECU is an instance of a particular component type. Multiple instances of the same component type can be mapped to an ECU instance and hence a mechanism is required to disambiguate access – for which the RTE API uses an instance handle.

The RTE API uses component-specific instance handles defined using the typedef `Rte_Instance` to provide an identifier for a component instance. All components of the same type can share the same `Rte_Instance` type definition.

For ease of use the name of the instance handle type remains the same for all components, even though the underlying types are different, since the actual type definition is private to the component. This feature relies on the direct equivalence in C/C++ be-

tween the typedef and the underlying type.

The instance handle is omitted from the RTE API when an AUTOSAR software-component is declared as singly instantiated in the SW-C description.

6.2.2 Std_ReturnType

The `Std_ReturnType` type defines the “status” and “error” values returned by RTE API functions. The following values are defined:

- RTE_E_OK
- RTE_E_INVALID
- RTE_E_LOST_DATA
- RTE_E_MAXAGE_EXCEEDED
- RTE_E_COM_STOPPED
- RTE_E_TIMEOUT
- RTE_E_LIMIT
- RTE_E_NO_DATA
- RTE_E_TRANSMIT_ACK
- RTE_E_NEVER_RECEIVED
- RTE_E_UNCONNECTED
- RTE_E_IN_EXCLUSIVE_AREA
- RTE_E_SEG_FAULT
- SCHM_E_OK
- SCHM_E_LIMIT
- SCHM_E_NO_DATA
- SCHM_E_TRANSMIT_ACK
- SCHM_E_IN_EXCLUSIVE_AREA

6.2.3 Port Handles

RTA-RTE creates a port handle type for each interface that is used to categorize a port on a software-component where the port has the indirect API enabled[1].

[1]The enabling can be explicit through use of a port API option or implicit when the SWC type supports multiple instantiation

A port handle type is used in conjunction with the Rte_Ports and Rte_NPorts APIs to support iteration over similarly typed ports.

The name of the port handle type is formed as follows:

```
Rte_PortHandle_<interface>_<P/R>
```

Where <interface> is the interface name typing the port and "P"/"R" is literal text indicating whether the port requires or provides the interface.

If the same interface is used to type both required and provided ports then two port handle types are created. The different port handle types are distinguished by the P/R suffix.

The port handle type is written to the software-components application header file and therefore is uniquely defined for each component type.

6.3 Rte_Call

```
Std_ReturnType
Rte_Call_<p>_<o>([IN Rte_Instance <instance>,]
                 [IN|INOUT|OUT] param_1, ...
                 [IN|INOUT|OUT] param_n)
```

Where <p> is the port name and <o> the operation within the client-server interface that categorizes the port.

6.3.1 Description

API call used by a client to initiate a client-server communication. The Rte_Call API is used for both synchronous and asynchronous calls.

The Rte_Call API includes zero or more IN, IN/OUT and OUT parameters:

- IN parameters are passed by value for primitive data types and by reference for all other types.
- OUT parameters are always passed by reference.
- IN/OUT parameters are passed by value when they are primitive data types and the call is asynchronous and by reference for all other cases.

6.3.2 Return Values

The return value is used to indicate errors detected by the RTE during execution of the Rte_Call call and, for synchronous communication, application errors returned from execution of the server.

RTE_E_OK – The call completed successfully.

RTE_E_TIMEOUT (Synchronous inter-task and inter-ECU only) – No reply was received within the configured timeout.

RTE_E_COM_STOPPED – A communications error occurred – the data has not been successfully passed to the communication service.

RTE_E_LIMIT – Multiple asynchronous client-server communications to the same server are attempted.

Application error – A synchronous client-server can, optionally, return an application error. The name and values of applications errors are defined within the port's interface.

6.3.3 Notes

For intra-task communication, or inter-task communication to a "pure" server, the RTE API `Rte_Call` may be elided completely and the call further mapped to a direct function call of the server's runnable entity.

A synchronous `Rte_Call` API is generated if a runnable entity within a software-component's internal behavior includes a SynchronousCallPoint specification.

An asynchronous `Rte_Call` API is generated if a a runnable entity within a software-component's internal behavior includes a AsynchronousCallPoint specification.

The OUT parameters are omitted for an asynchronous call since they are only required to collect the result (when using `Rte_Result`).

6.3.4 Example

Consider a required port, `ra`, containing an operation op1 that takes a single IN integer argument. The following calls can then be made:

```
SInt16 a = 23;
Std_ReturnType e = Rte_Call_ra_op1(self, a);
if (e == RTE_E_OK)
{
  /* call succeeded */
}
```

6.4 Rte_Prm

```
<data type> const
Rte_Prm_<port>_<name>([IN Rte_Instance <Instance>])
```

Where <name> is the name of the calibration element to access and <port> the name of the require port name(which must be categorized by a calibration interface).

6.4.1 Description

The Rte_Prm API provides access to calibration data within calibration components. Multiple software-component instances can access the same calibration data provided by an instance of a calibration component.

6.4.2 Return Values

The return value is typed according to the type of the configuration data. For primitive data types the value is returned however for complex types a pointer to the calibration data is returned.

From the perspective of an application software component calibration data is considered to be read-only and should not be altered.

6.4.3 Notes

A Rte_Prm API is created for each calibration prototype in the calibration interface.

The --calibration-instantiation option means that RTA-RTE can either import calibration data declared by the user in an external module or can instantiate memory for the calibration data. Due to deficiencies in the AUTOSAR XML input RTA-RTE does not create the calibration data – see Section 10.5 and therefore the initialization of the instantiated memory must be performed by the user before access by software-components.

6.4.4 Example

```
FUNC(void, RTE_APPL_CODE)
tt1(Rte_Instance self)
{
  UInt16 abc = Rte_Prm_port_abc(self);
}
```

6.5 Rte_CData

```
<data type> const
Rte_CData_<name>([IN Rte_Instance <Instance>])
```

Where <name> is the name of the calibration element.

6.5.1 Description

The Rte_CData API provides access to per-instance and shared calibration parameters declared within a software component. All instances of a component type access the same data for shared parameters whereas each accesses a unique copy for per-instance parameters.

6.5.2 Return Values

The return value is typed according to the type of the configuration data. For primitive data types the value is returned however for complex types a pointer to the calibration data is returned.

6.5.3 Notes

A Rte_CData API is created for each per-instance or shared calibration element.

RTA-RTE does not create the calibration data – see Section 10.5. The actual calibration data must be declared by the user in an external module.

6.5.4 Example

```
FUNC(void, RTE_APPL_CODE)
tt1(Rte_Instance self)
{
  UInt16 abc = Rte_CData_abc(self);
}
```

6.6 Rte_Enter

```
Std_ReturnType
Rte_Enter_<area>([IN Rte_Instance <instance>])
```

Where <area> is the exclusive area name.

6.6.1 Description

The Rte_Enter API call is invoked by an application software component to define the start of an exclusive area. The name of the exclusive area is included as part of the call name.

All instances of a component are independent and therefore the scope of a exclusive area extends only to the runnable entities within each instance of a component. When a runnable entity has ‘entered’ an exclusive area no other runnable entity in the component instance can enter the area until the first runnable entity has invoked Rte_Exit.

6.6.2 Return Values

RTE_E_OK – The exclusive area was entered successfully.The underlying implementation of explicit exclusive areas uses OS resources. Within the AUTOSAR OS it is not possible for a resource lock to fail and therefore neither can the Rte_Enter API.

6.6.3 Notes

The Rte_Enter API call is created by RTA-RTE when a runnable entity declares that it “can enter” an exclusive area.

The RTE generator will ensure that the API expands to a null macro when possible – for example when all accessing runnable entities are mapped to the same task or mapped to tasks that cannot preempt.

The implementation of the exclusive area will depend the implementation strategy specified in the ECU description file. If no strategy is specified the default is "Os resource".

It is not valid when using the AUTOSAR OS to wait on an OS event when one or more resources are locked. Application software components should therefore avoid invoking RTE APIs that may block, including synchronous client-server calls, when an exclusive area is locked.

*It is not valid when using the AUTOSAR OS to invoke **any** OS API when interrupts are locked. RTA-RTE uses AUTOSAR OS APIs within RTE API calls and hence application software components must exercise caution before invoking any RTE API call when an exclusive area that uses "interrupt blocking" is locked.*

Invocations of `Rte_Enter` (and `Rte_Exit`) may be nested as long as areas are exited in the reverse order they were entered. When "interrupt blocking" is used as an implementation strategy API calls should be avoided when running inside the exclusive area to avoid problems with missed task activations.

RTA-RTE supports selection of an existing OS resource to be used to implement an exclusive area using the vendor-specific `ExclusiveAreaOsResourceRef` parameter (see 4.19.2)

6.6.4 Example

Consider an exclusive area "A" to which a runnable declares "can enter" access. A component can enter the area as follows:

```
Rte_Enter_A(self);
```

6.7 Rte_Exit

```
Std_ReturnType
Rte_Exit_<area>([IN Rte_Instance <instance>])
```

Where <area> is the exclusive area name.

6.7.1 Description

The `Rte_Exit` API call is invoked by an application software component to define the end of an exclusive area. The name of the exclusive area is part of the API call name.

All instances of a component are independent and therefore the scope of a exclusive area extends only to the runnable entities within each instance of a component. When a runnable entity has 'entered' an exclusive area no other runnable entity in the com-

ponent instance can enter the area until the first runnable entity has invoked Rte_Exit.

6.7.2 Return Values

RTE_E_OK – The exclusive area was exited successfully.

6.7.3 Notes

The Rte_Exit API call is created by RTA-RTE when a runnable entity declares that it "can enter" an exclusive area.

The RTE generator will ensure that the API expands to a null macro when possible – for example when all accessing runnable entities are mapped to the same task or mapped to tasks that cannot preempt.

Invocations of Rte_Enter (and Rte_Exit) may be nested as long as areas are exited in the reverse order they were entered.

6.7.4 Example

Consider an exclusive area "A" to which a runnable has "can enter" access. Having entered the area using Rte_Enter, a component can exit the area as follows:

```
Rte_Exit_A(self);
```

6.8 Rte_IFeedback

```
Std_ReturnType
Rte_IFeedback_<re>_<port>_<item>([IN Rte_Instance <inst>])
```

Where <re> is the accessing runnable entity name, <port> is the port name and <item> the data element within the sender-receiver interface categorizing the port.

6.8.1 Description

The Rte_IFeedback API call provides access to transmission acknowledgement notifications for a sender-receiver communication. Unlike the Rte_Feedback API the API is always non-blocking.

The Rte_IFeedback API takes no parameters other than the instance handle since the return value is used to indicate either the success or failure of the API call and the feedback status.

6.8.2 Return Values

RTE_E_NO_DATA – No data (feedback information) was available and no other error occurred when the feedback read was attempted.

RTE_E_UNCONNECTED – Unconnected provider.

RTE_E_TRANSMIT_ACK – A "transmission acknowledgement" has been received. This return value indicates no error was detected during execution of the API call.

6.8.3 Notes

Transmission acknowledgement is enabled for a provided variableDataPrototype by the presence of an AcknowledgementRequest with type transmission.

A non-blocking API is created for a provided variableDataPrototype if acknowledgement is enabled and a DataWriteAccess references the variableDataPrototype or a DataWriteCompletedEvent references the runnable and the variableDataPrototype.

6.8.4 Example

Consider a provided port, pa, containing a data element val of type SInt16 that is specified using the SUCCESS element accessed from runnable re1. The following calls can then be made:

```
Rte_Write_pa_val(self, 23);
if (Rte_IFeedback_re1_pa_val(self) == RTE_E_TRANSMIT_ACK)
{
  /* Transmit okay */
}
```

6.9 Rte_Feedback / Rte_SwitchAck

```
Std_ReturnType
Rte_Feedback_<port>_<item>([IN Rte_Instance <inst>])
```

Or

```
Std_ReturnType
Rte_SwitchAck_<port>_<mode>([IN Rte_Instance <inst>])
```

Where <port> is the port name, <item> the data element within the sender-receiver interface categorizing the port and <mode> the mode declaration group prototype within the sender-receiver interface categorizing the port.

6.9.1 Description

The Rte_Feedback API call provides access to transmission acknowledgement notifications for a sender-receiver communication and mode switch acknowledgements for mode managers. The API can be either non-blocking or blocking depending on software-component configuration. Alternatively a runnable entity can be activated by RTA-RTE when the transmission status or mode switch acknowledgement is available in which case no API is created.

The Rte_Feedback API takes no parameters other than the instance handle since the return value is used to indicate either the success or failure of the API call and the feedback status.

6.9.2 Return Values

RTE_E_NO_DATA (non-blocking API only) – No data (feedback information) was available and no other error occurred when the feedback read was attempted.

RTE_E_TIMEOUT (blocking API only) – No data (feedback information) was available within the configured timeout and no other error occurred when the feedback read was attempted.

RTE_E_TRANSMIT_ACK – A “transmission acknowledgement” or “mode switch acknowledgement” has been received. This return value indicates no error was detected during execution of the API call.

6.9.3 Notes

Transmission acknowledgement is enabled for a provided DataElementPrototype by the presence of an AcknowledgementRequest with type transmission.

Mode switch acknowledgement is enabled for a provided ModeDeclarationGroup-Prototype by the presence of an ModeSwitchAck with a specified timeout.

A blocking API is created for a provided DataElementPrototype (ModeDeclarationGroup-Prototype) if acknowledgement is enabled and a WaitPoint references a DataSend-CompletedEvent (ModeSwitchedAckEvent) that in turn references the DataElement-Prototype (ModeSwitchPoint).

A non-blocking API is created for a provided DataElementPrototype (ModeDeclaration-GroupPrototype) if acknowledgement is enabled and a DataSendPoint (ModeSwitch-Point) references the DataElementPrototype but no DataSendCompletedEvent (Mode-SwitchedAckEvent) references the DataElementPrototype (ModeSwitchPoint).

When a DataSendCompletedEvent (ModeSwitchedAckEvent) that references a Data-ElementPrototype (ModeSwitchPoint) and a runnable entity then RTA-RTE will trigger the specified runnable when the acknowledgment is processed and will provide a non-blocking API call to read the acknowledgement state. It is not valid to combine activating a runnable entity with a blocking API call.

6.9.4 Example

Consider a provided port, pa, containing a data element `val` of type `SInt16` that is specified using the SUCCESS element with a feedback method of wake_up_of_wait_point. The following calls can then be made:

```
Rte_Send_pa_val(self, 23);
if (Rte_Feedback_pa_val(self) == RTE_E_TRANSMIT_ACK)
{
  /* Transmit okay */
}
```

6.10 Rte_IInvalidate

```
void
Rte_IInvalidate_<re>_<p>_<d>([IN Rte_Instance <inst>))
```

Where <re> is the runnable entity name, <p> is the port name and <d> the data element within the sender-receiver interface categorizing the port.

6.10.1 Description

Mark the data as "invalid". The transmission of the invalid value will occur after the runnable entity <re> has terminated.

6.10.2 Return Values

None.

6.10.3 Notes

An `Rte_IInvalidate` API is created for a provided DataElementPrototype for which a runnable entity has DataWriteAccess where the referenced DataElementPrototype is marked as "can invalidate".

6.10.4 Example

Consider a provide port, pa, containing a data element `val` of type `SInt16` which is non-queued, accessed using implicit communication by runnable `re1` and marked as "can invalidate". The following calls can then be made from within `re1`:

```
Rte_IInvalidate_re1_pa_val(self);
```

6.11 Rte_Invalidate

```
Std_ReturnType
Rte_Invalidate_<p>_<d>([IN Rte_Instance <inst>))
```

Where p is the port name and d the data element within the sender-receiver interface that categorizes the port.

6.11.1 Description

Marks the data as "invalid" and then transmits as normal (depending on configuration the transmission of the invalid data marker may occur immediately or may be deferred).

6.11.2 Return Values

RTE_E_OK – The data has been invalidated successfully.

RTE_E_COM_STOPPED – A communications error occurred – the data has not been successfully passed to the communication service.

6.11.3 Notes

An Rte_Invalidate API is created for a DataSendPoint that references a non-queued DataElementPrototype that is marked as “can invalidate”.

AUTOSAR restricts invalidation to integer types.

6.11.4 Example

Consider a provide port, pa, containing a data element val of type SInt16 which is non-queued and marked as “can invalidate”. The following call can then be made:

```
Rte_Invalidate_pa_val(self);
```

6.12 Rte_IRead

```
<data type>
Rte_IRead_<re>_<p>_<d>([IN Rte_Instance <inst>))
```

Where <re> is the runnable entity name, <p> is the port name and <d> the data element within the sender-receiver interface categorizing the port.

6.12.1 Description

Perform an implicit read on a sender-receiver communication using non-queued semantics. The API is always implemented as a macro rather than a generated function.

The data item accessed by the Rte_IRead API is guaranteed not to change during execution of runnable <re>.

The Rte_IRead API returns the read data using the API return type.

6.12.2 Return Values

Dependent on data value.

6.12.3 Notes

An Rte_IRead API is created for a required DataElementPrototype if the runnable entity has DataReadAccess that refers to the DataElementPrototype.

Implicit communication supports primitive data types and complex types including records.

6.12.4 Example

Consider a required port, ra, containing a data element val of type SInt16 accessed from runnable entity re1. The following call can then be made:

```
SInt16 a;
a = Rte_IRead_re1_ra_val(self);
```

6.13 Rte_IWrite

```
void
Rte_IWrite_<re>_<p>_<d>([IN Rte_Instance <inst>]
                        IN <type> <data>)
```

Where re is the runnable entity name, p is the port name and d the data element within the sender-receiver interface categorizing the port.

6.13.1 Description

Perform an implicit write on a sender-receiver communication using “data” semantics. The API is always implemented as a macro rather than a generated function.

The data item written by the `Rte_IWrite` API is only made visible to other components after the runnable `re` has terminated.

6.13.2 Return Values

None.

6.13.3 Notes

An `Rte_IWrite` API is created for a required DataElementPrototype if the runnable entity has DataWriteAccess that refers to the DataElementPrototype.

Implicit communication supports primitive data types and records. When a complex data type is used the second parameter is a pointer to the data.

6.13.4 Example

Consider a provided port, pa, containing a data element `val` of type `SInt16` accessed from runnable entity `re1`. The following call can then be made:

```
Rte_IWrite_re1_pa_val(self, 23);
```

6.14 Rte_IWriteRef

```
<type>*
Rte_IWriteRef_<re>_<p>_<d>([IN Rte_Instance])
```

Where re is the runnable entity name, p is the port name and d the data element within the sender-receiver interface categorizing the port.

6.14.1 Description

Access a reference to the RTE manager buffer used for implicit writes on a sender-receiver communication using “data” semantics. The API is always implemented as a macro rather than a generated function. The reference can be used to directly update members/elements of complex types.

The data item written by the `Rte_IWriteRef` API is only made visible to other components after the runnable re has terminated.

6.14.2 Return Values

`Rte_IWriteRef` returns a reference to the corresponding data element. Therefore the return type of `Rte_IWriteRef` is dependent on the data element type.

6.14.3 Notes

An `Rte_IWriteRef` API shall be created for a provided DataElementPrototype if the RunnableEntity has DataWriteAccess that refers to the DataElementPrototype. Thus the API will be generated for both primitive and complex data types though it is most useful with complex types to update single fields/elements of the type.

The `Rte_IWriteRef` API can be used as an l-value within an assignment.

6.14.4 Example

Consider a required port, ra, containing a data element `val` of structure type RecType accessed from runnable entity `re1`. The following calls can then be made:

```
RecType* r = Rte_IWriteRef_re1_ra_val(self);
r->field = 23;
```

6.15 Rte_IrvIRead

```
<data type>
Rte_IrvIRead_<re>_<name>([IN Rte_Instance <inst>])
```

Where <re> is the runnable entity name and <name> the inter-runnable variable name.

6.15.1 Description

Provide read access to an inter-runnable variable with an IMPLICIT communication approach. Where data consistency is required, access to the inter-runnable variable by a runnable entity is redirected to a copy created by RTA-RTE.

The `Rte_IrvIRead` API returns the value of the inter-runnable variable using the API return type.

6.15.2 Return Values

Dependent on data value. Note that inter-runnable variables do not support complex types or strings.

6.15.3 Notes

The `Rte_IrvIRead` API mapping is implemented as a macro written to the component's application header file by the RTE generator. If support for multiple instances is disabled within the software-component's internal behavior the macro is implemented as a direct access of the component data structure.

An Rte_IrvIRead API is created for each "ReadVariable" with an IMPLICIT communication approach that is referenced by a runnable entity. The generated RTE includes appropriate data copies to ensure that the access to the inter-runnable variable is atomic.

The concurrency control applied to inter-runnable variable access ensures that the read is atomic – it does not, however, ensure that a read-modify-write cycle is protected. If such protection is required then a per-instance memory section and an exclusive area should be used instead.

Inter-runnable variables support primitive data types (excluding strings).

6.15.4 Example

Consider an inter-runnable variable irv1 (with type SInt16 and IMPLICIT access) referenced by runnable entity re1. The following API call can then be made:

```
SInt16 a;
a = Rte_IrvIRead_re1_irvl(self);
```

6.16 Rte_IrvIWrite

```
void
Rte_IrvIWrite_<re>_<name>([IN Rte_Instance <inst>],
                          IN <data>)
```

Where <re> is the runnable entity name and <name> the inter-runnable variable name.

6.16.1 Description

Provide write access to an inter-runnable variable. Where data consistency is required access to the inter-runnable variable by a runnable entity is redirected to a copy created by RTA-RTE

6.16.2 Return Values

The return value of Rte_IrvIWrite API is always void – the call is implemented as a macro and cannot fail.

6.16.3 Notes

The Rte_IrvIWrite API mapping is implemented as a macro written to the component's application header file by the RTE generator. If support for multiple instances is disabled within the software-component's internal behavior the macro is implemented as a direct access of the component data structure.

An Rte_IrvIWrite API is created for each "WrittenVariable" with an IMPLICIT communication approach that is referenced by a runnable entity. The generated RTE includes appropriate concurrency control to ensure that the write of the inter-runnable variable is atomic.

The concurrency control applied to inter-runnable variable access ensures that the

write is atomic – it does not, however, ensure that a read-modify-write cycle is protected. If such protection is required then a per-instance memory section and an exclusive area should be used instead.

Inter-runnable variables support primitive data types (excluding strings).

6.16.4 Example

Consider an inter-runnable variable `irv1` (with type `SInt16` and IMPLICIT access) referenced using 'written' semantics by runnable entity `re1`. The following API call can then be made:

```
SInt16 a;
Rte_IrvIWrite_re1_irvl(self, a);
```

6.17 Rte_IrvRead

```
<data type>
Rte_IrvRead_<re>_<name>([IN Rte_Instance <inst>])
```

Where <re> is the runnable entity name and <name> the inter-runnable variable name.

6.17.1 Description

Provide read access to an inter-runnable variable with an `EXPLICIT` communication approach. The access to the inter-runnable variable is guaranteed to be atomic.

The `Rte_IrvRead` API returns the value of the inter-runnable variable using the API return type.

6.17.2 Return Values

Dependent on data value. Note that inter-runnable variables do not support complex types or strings.

6.17.3 Notes

The `Rte_IrvRead` API mapping is implemented as a macro written to the component's application header file by the RTE generator. If support for multiple instances is disabled the macro is implemented as a direct invocation of the function generated within the RTE.

An `Rte_IrvRead` API is created for each "ReadVariable" with an EXPLICIT communication approach that is referenced by a runnable entity. The generated RTE includes appropriate concurrency control to ensure that the access to the inter-runnable variable is atomic.

The concurrency control applied to inter-runnable variable access ensures that the read is atomic – it does not, however, ensure that a read-modify-write cycle is protected. If such protection is required then a per-instance memory section and an exclusive area should be used instead.

Inter-runnable variables support primitive data types (excluding strings).

6.17.4 Example

Consider an inter-runnable variable `irv1` (with type `SInt16`) referenced by runnable entity `re1`. The following API calls can then be made:

```
SInt16 a;
a = Rte_IrvRead_re1_irvl(self);
```

6.18 Rte_IrvWrite

```
void
Rte_IrvWrite_<re>_<name>([IN Rte_Instance <inst>],
                         IN <data>)
```

Where <re> is the runnable entity name and <name> the inter-runnable variable name.

6.18.1 Description

Provide write access to an inter-runnable variable with an EXPLICIT communication approach. The access to the inter-runnable variable is guaranteed to be atomic.

6.18.2 Return Values

The return value of `Rte_IrvWrite` API is always void – the call is implemented as a macro and therefore cannot fail.

6.18.3 Notes

The `Rte_IrvWrite` API mapping is implemented as a macro written to the component's application header file by the RTE generator. If support for multiple instances is disabled the macro is implemented as a direct invocation of the function generated within the RTE.

An `Rte_IrvWrite` API is created for each "WrittenVariable" with an EXPLICIT communication approach that is referenced by a runnable entity. The generated RTE includes appropriate concurrency control to ensure that the write of the inter-runnable variable is atomic.

The concurrency control applied to inter-runnable variable access ensures that the write is atomic – it does not, however, ensure that a read-modify-write cycle is protected. If such protection is required then a per-instance memory section and an exclusive area should be used instead.

Inter-runnable variables support primitive data types (excluding strings).

6.18.4 Example

Consider an inter-runnable variable `irv1` (with type `SInt16`) referenced using 'written' semantics by runnable entity `re1`. The following API calls can then be made:

```
SInt16 a;
Rte_IrvWrite_re1_irvl(self, a);
```

6.19 Rte_IStatus

```
Std_ReturnType
Rte_IStatus_<re>_<p>_<d>([IN Rte_Instance <inst>])
```

Where <re> is the runnable entity name, <p> is the port name and <d> the data element within the sender-receiver interface categorizing the port.

6.19.1 Description

Provide read access to the error state of an implicitly accessed datum.

6.19.2 Return Values

RTE_E_OK – No error detected.

RTE_E_COM_STOPPED – A communications error occurred – the data has not been successfully passed to the communication service.

6.19.3 Notes

An `Rte_IStatus` API is created for each datum with implicit read access and a runnable entity triggered by a DataReceiveErrorEvent.

6.19.4 Example

Consider a SW-C with port prototype pa categorized by an interface containing datum value (with type `SInt16`) where the datum is accessed using ‘read’ access by `re1` and a DataReceiveErrorEvent is configured to activate `re1`. The following API calls can then be made:

```
Std_ReturnType r;
r = Rte_IStatus_re1_pa_value(self);
```

6.20 Rte_IsUpdated

```
Std_ReturnType
Rte_IsUpdated_<p>_<vdp>([IN Rte_Instance <inst>])
```

Where <p> is the port name and <vdp> the VariableDataPrototype within the sender-receiver interface categorizing the port.

6.20.1 Description

Indicate whether the VariableDataPrototype has been updated or not.

6.20.2 Return Values

TRUE – DataElement updated since last read.

FALSE – DataElement not updated since last read.

6.21 Rte_MainFunction

```
void
Rte_MainFunction(void)
```

6.21.1 Description

Rte_MainFunction is a vendor-specific API used by RTA-RTE to implement Minimum Start Intervals (Section 4.10.10).

If Minimum Start Intervals are used, then Rte_MainFunction must be called periodically to notify RTA-RTE of the passing of time. By default, RTA-RTE expects the function to be called every 10 milliseconds, but you can choose your own period by using the --period command-line option. Note however that all minimum start intervals in the system must be an integer multiple of this period.

6.21.2 Return Values

None.

6.21.3 Notes

The Rte_MainFunction API is declared in Rte_Main.h.

The Rte_MainFunction API is RTA-RTE specific.

Rte_MainFunction can always be called, but when there are no Minumum Start Intervals present, it has no effect. You can use the preprocessor symbol RTE_MAINFUNCTION_REQUIRED to check whether it is necessary to call Rte_MainFunction.

The configured period is made available with the pre-processor symbol RTE_MAINFUNCTION_PERIOD_NS that expands to the period in nanoseconds. For convenience the symbols RTE_MAINFUNCTION_PERIOD_US and RTE_MAINFUNCTION_PERIOD_MS are provided at microsecond and millisecond precision respectively. These expand to integer values when the period is an integer multiple of one microsecond or one millisecond respectively, otherwise they expand to floating point values.

6.21.4 Example

Assume ISR isr1 is triggered every millisecond. Rte_MainFunction can be invoked at the correct rate using the following code:

```
ISR(isr1)
```

```
{
#ifdef RTE_MAINFUNCTION_REQUIRED
# if ( ( RTE_MAINFUNCTION_PERIOD_NS % 1000000UL ) != 0 )
#  error "Configured Rte_MainFunction invocation period is not a
    multiple of 1ms"
# endif /* ( ( RTE_MAINFUNCTION_PERIOD_NS % 1000000UL ) != 0 ) */

  static uint16 count = RTE_MAINFUNCTION_PERIOD_MS;
#endif /* RTE_MAINFUNCTION_REQUIRED */

#ifdef RTE_MAINFUNCTION_REQUIRED
  if ( 0 == --count )
  {
    count = RTE_MAINFUNCTION_PERIOD_MS;
    Rte_MainFunction();
  }
#endif /* RTE_MAINFUNCTION_REQUIRED */

  ...
}
```

6.22 Rte_Mode

```
Rte_ModeType_<m>
Rte_Mode_<port>_<item>([IN Rte_Instance <inst>])
```

Where <port> is the port name, <item> the mode declaration group prototype name within the sender-receiver interface categorizing the port and <m> the name of the referenced mode declaration group.

6.22.1 Description

Read the current mode for a “mode graph”.

6.22.2 Return Values

The return value from the Rte_Mode API indicates the current mode. The name of the returned data type is derived from the name of the mode declaration group.

6.22.3 Notes

The Rte_Mode API is created when a ModeDeclarationGroupPrototype is declared within a provided or required interface.

The Rte_Mode API is always a non-blocking API.

The Rte_Mode API generated for an unconnected port always returns the mode declaration group’s initial mode. No mode switch event other than ENTRY to the initial mode (i.e. within Rte_Start) will ever be triggered.

6.22.4 Example

Consider a required port, ra, containing a mode declaration group prototype mode that references mode declaration group mg1. The following API call can then be made:

```
Rte_ModeType_mg1 m = Rte_Mode_ra_mode(self);
```

6.23 Rte_Ports

```
Rte_PortHandle_<i>_<P/R>
Rte_Ports_<i>_<P/R>([IN Rte_Instance <inst>])
```

Where <i> is the interface name and <P/R> dependent on whether the ports ‘provide’ or ‘require’ the interface.

6.23.1 Description

The `Rte_Ports` API provides access to the base of an array created for each interface (and ‘provide’ or ‘require’ usage) of a software-component. The `Rte_Ports` API supports iteration through similarly typed ports when combined with the `Rte_NPorts` API.

6.23.2 Return Values

The `Rte_Ports` API returns a port handle corresponding to the appropriate interface and ‘provide’ or ‘require’ usage. Port handles are declared in the application header file.

6.23.3 Notes

One `Rte_Ports` API is created for each interface (inc. ‘provide’ or ‘require’ usage) used within a software-component. Thus a software-component includes port specifications that both require and provide the same interface will result in two `Rte_Ports` APIs being generated.

RTA-RTE includes an explicit array within the component data structure to store individual port data structures. This is safe with regards port iteration and consistent with the AUTOSAR SWS requirements but is inconsistent with the example given in the AUTOSAR SWS.

6.23.4 Example

Consider a software component with two required ports, ra and rb, both typed by interface i2. An `Rte_Ports` API consistent with the following signature would then be created:

```
Rte_PortHandle_i2_R Rte_Ports_i2_R(Rte_Instance);
```

6.24 Rte_NPorts

```
uint8
Rte_NPorts_<i>_<P/R>([IN Rte_Instance <inst>])
```

Where xmltagi is the interface name and <P/R> dependent on whether the ports 'provide' or 'require' the interface.

6.24.1 Description

The `Rte_NPorts` API provides access to the size of the array created for each interface (and 'provide' or 'require' usage) of a software-component. The `Rte_NPorts` API supports iteration through similarly typed ports when combined with the `Rte_Ports` API.

6.24.2 Return Values

The `Rte_NPorts` API returns a `uint8` corresponding to the number of ports that use the same interface and 'provide' or 'require' usage.

6.24.3 Notes

One `Rte_NPorts` API is created for each interface (inc. 'provide' or 'require' usage) used within a software-component for which the indirect API is created. Thus a software-component includes port specifications that both require and provide the same interface will result in two `Rte_NPorts` APIs being generated.

6.24.4 Example

Consider a software component with two required ports, `ra` and `rb`, both typed by interface `i2`. An `Rte_Ports` API consistent with the following signature would then be created:

```
uint8 Rte_NPorts_i2_R(Rte_Instance);
```

The `Rte_NPorts_i2_R` API would return two since there are two require ports typed by interface `i2`.

The `Rte_NPorts` API can be combined with the `Rte_Ports` API to support iteration over similarly typed ports, for example:

```
uint8 count = Rte_NPorts_i2_R(self);
Rte_PortHandle_i2_R base = Rte_Ports_i2_R(self);
for ( p = 0; p < count; p++ )
{
  base[p]->Send_a(49);
}
```

6.25 Rte_Port

```
Rte_PortHandle_<i>_<P/R>
Rte_Port_<port>([IN Rte_Instance <inst>])
```

Where <port> is the port name, <i> the interface name and <P/R> dependent on whether the ports 'provide' or 'require' the interface.

6.25.1 Description

The Rte_Port API provides access to port handle for a specified port of a software-component. The API allows a software-component to extract a sub-group of ports typed by the same interface in order to iterate over this sub-group.

6.25.2 Return Values

The Rte_Ports API returns a port handle corresponding to the appropriate interface and 'provide' or 'require' usage. Port handles are declared in the application header file.

6.25.3 Notes

One Rte_Port API is created for each port declared within a software-component.

6.25.4 Example

Consider a software component with two required ports, ra and rb, both typed by interface i2. Two Rte_Port APIs would be created consistent with the following signature:

```
Rte_PortHandle_i2_R Rte_Ports_ra(Rte_Instance);
Rte_PortHandle_i2_R Rte_Ports_rb(Rte_Instance);
```

6.26 Rte_Pim

```
<type>*
Rte_Pim_<name>([IN Rte_Instance <inst>])
```

Where <name> is the short-name name of the per-instance memory section to access.

6.26.1 Description

The Rte_Pim API provides access to a named per-instance memory (PIM) section defined in the software component type. The function returns a pointer to a data memory section that can be used directly, without any need for type casting, by the software-component.

6.26.2 Return Values

A pointer to an instance of the type defined for the per-instance memory section.

The data type of the per-instance memory section is defined in the software component's application header file and therefore the actual type for each generated Rte_Pim call will vary.

6.26.3 Notes

An Rte_Pim API is created for each defined PerInstanceMemorySection within a software-component.

RTA-RTE instantiates the per-instance memory section once for each component instance. When multiple instances of a component are possible the macro is mapped to access the state within the specified instance. When only a single instance is known to exist, the indirection through the instance handle may be elided and thus impose no run-time penalty on access to static memory sections.

6.26.4 Example

The following example illustrates the use of Rte_Pim. The following declaration of a per-instance memory section:

```
<PER-INSTANCE-MEMORY>
<SHORT-NAME>val</SHORT-NAME>
<TYPE>dataType</TYPE>
<TYPE-DEFINITION>
  struct { uint8 var1; ... }
</TYPE-DEFINITION>
</PER-INSTANCE-MEMORY>
```

Results in the creation by RTA-RTE of an instance of dataType that can be accessed through the Rte_Pim_val API call:

```
FUNC(void, RTE_APPL_CODE)
tt1(Rte_Instance self)
{
  dataType* ds = Rte_Pim_val(self);
  ds->var1 = 23;
  // ...
}
```

6.27 Rte_Read

```
Std_ReturnType
Rte_Read_<port>_<item>([IN Rte_Instance <inst>,]
                       OUT <data>)
```

Where <port> is the port name and <item> the data element within the sender-receiver interface that categorizes the port.

6.27.1 Description

Perform an explicit read on a sender-receiver communication using “data” semantics.

The Rte_Read API includes exactly one OUT parameter to pass back the received data – the data will be passed by reference for all data types.

6.27.2 Return Values

RTE_E_OK – The data has been read successfully.

RTE_E_TIMEOUT (blocking API only) – No data was received within the configured timeout.

RTE_E_NO_DATA (non-blocking API only) – No data was available for reading but no other error occurred when the read was attempted. This return value is not considered an error.

6.27.3 Notes

The Rte_Read call is created when a DataElementPrototype has an isQueued declaration of “false”.

A non-blocking API is created if a DataReceivePoint references a required DataElementPrototype with ‘data’ semantics

Blocking Rte_Read API calls are not permitted in the AUTOSAR RTE specification.

When a DataReceiveEvent that references a DataElementPrototype and a runnable entity then RTA-RTE will trigger the specified runnable when a datum is received and will provide a non-blocking API call to read the received datum. It is not valid to combine activating a runnable entity with a blocking API call.

6.27.4 Example

Consider a required port, ra, containing a data element `val` of type `SInt16` with an `isQueued` declaration of “false”. The following calls can then be made:

```
SInt16 a;
Std_ReturnType e = Rte_Read_ra_val(self, &a);
if (e == RTE_E_OK)
{
  /* Read okay */
}
```

6.27.5 Related APIs

See also `Rte_DRead`, `Rte_Receive`.

6.28 Rte_DRead

```
<type>
Rte_DRead_<port>_<item>([IN Rte_Instance <inst>])
```

Where <port> is the port name and <item> the data element within the sender-receiver interface that categorizes the port.

6.28.1 Description

Perform an explicit read on a sender-receiver communication using “data” semantics.

The Rte_Read API includes exactly one OUT parameter to pass back the received data – the data will be passed by reference for all data types.

6.28.2 Return Values

RTE_E_OK – The data has been read successfully.

RTE_E_TIMEOUT (blocking API only) – No data was received within the configured timeout.

RTE_E_NO_DATA (non-blocking API only) – No data was available for reading but no other error occurred when the read was attempted. This return value is not considered an error.

6.28.3 Notes

The Rte_Read call is created when a DataElementPrototype has an isQueued declaration of “false”.

A non-blocking API is created if a DataReceivePoint references a required DataElement-Prototype with ‘data’ semantics

Blocking Rte_Read API calls are not permitted in the AUTOSAR RTE specification.

When a DataReceiveEvent that references a DataElementPrototype and a runnable entity then RTA-RTE will trigger the specified runnable when a datum is received and will provide a non-blocking API call to read the received datum. It is not valid to combine activating a runnable entity with a blocking API call.

6.28.4 Example

Consider a required port, ra, containing a data element val of type SInt16 with an isQueued declaration of “false”. The following calls can then be made:

```
SInt16 a;
Std_ReturnType e = Rte_Read_ra_val(self, &a);
if (e == RTE_E_OK)
{
  /* Read okay */
}
```

6.29 Rte_Receive

```
Std_ReturnType
Rte_Receive_<port>_<item>([IN Rte_Instance <inst>,]
                          OUT <data>)
```

Where `<port>` is the port name and `<item>` the data element within the sender-receiver interface that categorizes the port.

6.29.1 Description

Perform an explicit read on a sender-receiver communication using “event” semantics.

The `Rte_Receive` API includes exactly one OUT parameter to pass back the received data – the data will be passed by reference for all data types.

6.29.2 Return Values

RTE_E_OK – The data has been read successfully.

RTE_E_TIMEOUT (blocking API only) – No data was received within the configured timeout.

RTE_E_NO_DATA (non-blocking API only) – No data was available for reading but no other error occurred when the read was attempted. This return value is not considered an error.

6.29.3 Notes

The `Rte_Receive` call is created when a DataElementPrototype is “QUEUED”.

A non-blocking API is created if a DataReceivePoint references a required DataElement-Prototype with ‘QUEUED’ semantics

A blocking API is created if a DataReceivePoint references a required DataElement-Prototype with ‘QUEUED’ semantics and a WaitPoint references a DataReceiveEvent that references the same DataElementPrototype.

When a DataReceiveEvent that references a DataElementPrototype and a runnable entity then RTA-RTE will trigger the specified runnable when an event is received and will provide a non-blocking API call to read the received event. It is not valid to combine activating a runnable entity with a blocking API call.

6.29.4 Example

Consider a required port, `ra`, containing a data element `val` of type `SInt16` with the isQueued attribute set to “true”. The following calls can then be made:

```
SInt16 a;
Std_ReturnType e = Rte_Receive_ra_val(self, &a);
if (e == RTE_E_OK)
{
  /* Receive okay */
```

```
}
```

6.30 Rte_Result

```
Std_ReturnType
Rte_Result_<port>_<op>([IN Rte_Instance <inst>,]
                       [OUT <out_param_1>], ...,
                       [OUT <out_param_n>])
```

Where `<port>` is the port name and `<op>` the operation within the client-server interface that categorizes the port.

6.30.1 Description

The `Rte_Result` API is used to poll for the result and the error state of an asynchronous client-server communication. The call can be either blocking or non-blocking.

The `Rte_Result` API includes zero or more OUT parameters to pass back results from the server. All OUT parameters are passed by reference for all data types

6.30.2 Return Values

The return value is used to indicate errors from either the `Rte_Result` call itself or communication errors detected before the API call was made.

RTE_E_OK – The call completed successfully.

RTE_E_TIMEOUT (blocking call only) – No reply was received within the configured timeout.

RTE_E_NO_DATA (non-blocking call only) – No data was available for reading but no other error occurred when the read was attempted. This return value is not considered an error.

6.30.3 Notes

A non-blocking API is created for an AsynchronousServerCallReturnsEvent that references a required OperationPrototype and no WaitPoint references the AsynchronousServerCallReturnsEvent.

A blocking API is created for an AsynchronousServerCallReturnsEvent that references a required OperationPrototype and a WaitPoint references the AsynchronousServerCallReturnsEvent.

When an AsynchronousServerCallReturnsEvent that references an OperationPrototype and a runnable entity then RTA-RTE will trigger the specified runnable when the server result is received and will provide a non-blocking API call to read the received result. It is not valid to combine activating a runnable entity with a blocking API call.

6.30.4 Example

Consider a required port, pa, containing an operation op1 which is invoked asynchronously with the result being collected using a blocking API call. The following calls can then be made:

```
SInt16 a;
Std_ReturnType e;

Rte_Call_pa_op1(self, 1);
e = Rte_Result_pa_op1(self, &a)
if (e == RTE_E_OK)
{
  /* Result received okay */
}
```

6.31 Rte_Send

```
Std_ReturnType
Rte_Send_<port>_<item>([IN Rte_Instance <inst>,]
                       IN <data>)
```

Where <port> is the port name and <item> the data element within the sender-receiver interface that categorizes the port.

6.31.1 Description

Initiates a sender-receiver communication using queued semantics.

The Rte_Send API includes exactly one IN parameter for data – this will be passed by value for primitive data types and by reference for all other types.

6.31.2 Return Values

RTE_E_OK – The data has been passed to communication service successfully. Depending on bus load the data may or may not have been transmitted.

RTE_E_COM_STOPPED – A communications error occurred – the data has not been successfully passed to the communication service.

6.31.3 Notes

An Rte_Send API call is created when a DataSendPoint is specified for a provided DataElementPrototype for which the isQueued attribute is set to “true”.

The API returns when the signal has been passed to the communication service for transmission. Depending on the communication server the transmission may or may not have been acknowledged by the receiver.

If a sender port has multiple receivers connected, the generated Rte_Send API will try to write to all receivers independently. This ensures that. for example, the overflow in one component's queue does not prevent the transmission of this message to other components.

6.31.4 Example

Consider a provided port, ra, containing a data element val of type `SInt16` with the `isQueued` attribute set to "true". The following call can then be made:

```
Std_ReturnType e = Rte_Send_ra_val(self, 23);
if (e == RTE_E_OK)
{
  /* Transmission okay */
}
```

6.32 Rte_Start

```
Std_ReturnType
Rte_Start( void )
```

6.32.1 Description

This function is invoked (typically by the ECU state manager or possibly by user code) to start the RTE.

The `Rte_Start` API is responsible for starting the execution of periodic runnables and for initiating execution of runnables trigger by ModeSwitchEvents for ENTRY to the initial mode.

When the `Rte_Start` API completes the RTE is initialized and the RTE's response to inter-ECU communication is enabled.

6.32.2 Return Values

RTE_E_OK – No error detected.

RTE_E_LIMIT – Unable to start periodic runnable entities.

6.32.3 Notes

The `Rte_Start` invokes OS APIs to start the execution of timing triggered runnables.

6.33 Rte_Stop

```
Std_ReturnType
Rte_Stop( void )
```

6.33.1 Description

This function is invoked (typically by the ECU state manager or possibly by user code) to stop the RTE.

The `Rte_Stop` API is responsible for stopping further execution of periodic runnables. Note that runnables already running will not be terminated.

When the `Rte_Stop` API completes no further RTE initiated activity will take place and the RTE's response to inter-ECU communication is disabled.

6.33.2 Return Values

RTE_E_OK – No error detected.

RTE_E_LIMIT – Error when trying to stop periodic runnable entities.

6.33.3 Notes

The `Rte_Stop` API invokes OS APIs to halt execution of timing triggered runnables. The selected API depends on the chosen OS plug-in.

6.34 Rte_Switch

```
Std_ReturnType
Rte_Switch_<port>_<item>([IN Rte_Instance <inst>,]
                         IN <data>)
```

Where `<port>` is the port name and `<item>` the mode declaration group prototype name within the sender-receiver interface categorizing the port.

6.34.1 Description

Initiates a mode switch transition within a "mode graph".

The `Rte_Switch` API includes exactly one IN parameter for data passed by value. The IN parameter indicates the required 'next' mode.

The `Rte_Switch` API initiates the mode switch if the mode graph is not currently involved in a transition. Otherwise the request is queued.

The `Rte_Switch` API supports *synchronous* and *asynchronous* mode switches. The latter method is selected when explicitly enabled by all mode users.

6.34.2 Return Values

RTE_E_OK – The mode switch request has been queued successfully.

RTE_E_LIMIT – The mode switch request has been discarded due to a full request queue.

6.34.3 Notes

The Rte_Switch call is generated for each mode declaration group prototype within the categorizing interface for a provided port where at least one runnable entity has a ModeSwitchPoint declared for the mode declaration group prototype.

If a provided port has multiple receivers connected, the mode switch will apply to all receivers.

The Rte_Switch API generated for an unconnected port discards the mode switch request and returns RTE_E_OK. No mode switch will occur as a result of the API's invocation.

6.34.4 Example

Consider a provided port, ra, containing a mode declaration group prototype mode. Furthermore, assume that the referenced mode declaration group defines the mode RUN. The following Rte_Switch call can then be made:

```
Std_ReturnType e;
e = Rte_Switch_ra_mode(self, RTE_MODE_RUN);
if (e != RTE_E_OK)
{
  /* Mode switch request failed */
}
```

6.35 Rte_Tick_Timeouts

```
void
Rte_Tick_Timeouts(void)
```

6.35.1 Description

Rte_Tick_Timeouts ticks the Rte_Tout_Counter and sets any necessary OS events when alarms expire.

To prevent missed runnable activations and event settings it is important not to tick Rte_Tout_Counter directly but to call Rte_Tick_Timeouts instead.

6.35.2 Return Values

None.

6.35.3 Notes

Rte_Tick_Timeouts is needed in when generated RTE has timeouts present in the system and is generated for AUTOSAR OS or OSEK OS.

ETAS *The Rte_Tick_Timeouts API is a vendor-specific mechanism created in order to ensure compatibility with the restrictions placed by AUTOSAR OS and OSEK OS on the use of OS API calls in Alarm Callbacks.*

Rte_Tick_Timeouts should be called at the interval (in microseconds) defined by RTA-RTE in RTE_ALARM_COUNTER_TICK_INTERVAL_US.

Although the prototype is always present in Rte.h, the function itself is only generated if timeouts are present in the system. If no timeouts are present the RTE_ALARM_COUNTER_TICK_INTERVAL_US preprocessor symbol is undefined.

It is recommended to use conditional compilation (see the example below) to cause the call to Rte_Tick_Timeouts call only to be included when RTE_ALARM_COUNTER_TICK_INTERVAL_US is defined

6.35.4 Example

This example demonstrates the recommended way of calling Rte_Tick_Timeouts from within an ISR body.

```
uint8 count = USECONDS_TO_TICKS ( RTE_ALARM_COUNTER_TICK_INTERVAL_US
    );

...

#ifdef RTE_ALARM_COUNTER_TICK_INTERVAL_US
if ( 0 == --count )
{
  count = USECONDS_TO_TICKS ( RTE_ALARM_COUNTER_TICK_INTERVAL_US );
  Rte_Tick_Timeouts();
}
#endif /* RTE_ALARM_COUNTER_TICK_INTERVAL_US */
```

6.36 Rte_Trigger

```
void
Rte_Trigger_<p>_<itp>(void)
```

Where <p> is the port name and <itp> is the name of the InternalTriggeringPoint.

6.36.1 Description

Trigger the execution for all runnables whose ExternalTriggerOccurredEvent is associated with the trigger.

6.37 Rte_IrTrigger

```
void
Rte_IrTrigger_<re>_<itp>(void)
```

Where `<re>` is the runnable entity name and `<itp>` is the name of the InternalTriggeringPoint.

6.37.1 Description

Trigger the execution for all runnables whose InternalTriggerOccurredEvent is associated with the trigger.

6.38 Rte_Write

```
Std_ReturnType
Rte_Write_<port>_<data>([IN Rte_Instance <inst>,]
                        IN <data>)
```

Where `<port>` is the port name and `<item>` the data element within the sender-receiver interface that categorizes the port.

6.38.1 Description

`Rte_Write` initiates sender-receiver communication using “UNQUEUED” semantics.

The call includes exactly one IN parameter for data – this will be passed by value for primitive data types and by reference for all other types.

6.38.2 Return Values

RTE_E_OK – The data has been passed to communication service successfully. Depending on bus load the data may or may not have been transmitted.

RTE_E_COM_STOPPED – A communications error occurred – the data has not been successfully passed to the communication service.

RTE_E_SIGNAL_DISCARDED – The signal was discarded (failed to pass an outgoing filter) without being transmitted. This return is not considered to be an error.

Value-based filters are not implemented in RTA-RTE

RTE_E_LIMIT – An internal limit (e.g. queue length) has been exceeded.

6.38.3 Notes

An `Rte_Write` API call is created when a DataSendPoint is specified for a provided DataElementPrototype with ‘data’ semantics. The `Rte_Write` API call is generated when the isQueued attribute for the data element prototype is set to “false”.

The API returns when the signal has been passed to the communication service for transmission. Depending on the communication server the transmission may or may not have been acknowledged by the receiver.

If a sender port has multiple receivers connected, the Rte_Write API must try to write to all receivers independently. This ensures that overflow in one component's queue does not prevent the transmission of this message to other components.

6.38.4 Example

Consider a provided port, ra, containing a data element val of type SInt16 with the isQueued attribute set to “false”. The following calls can then be made:

```
Std_ReturnType e = Rte_Write_ra_val(self, 23);
if (e == RTE_E_OK)
{
  /* Transmission okay */
}
```

7 RTE Runnable API Reference

A software-component implementation defines functions that provide the entry points for runnable entities. The call name and parameters of the entry point function vary depending on the event that triggers the runnable entity.

7.1 Supported RTE Events

The following classes of RTE event are supported by RTA-RTE;

Asynchronous Server Call Returns Event – a runnable entity activated when the result of an asynchronous invocation of a client-server operation is available. The function providing the runnable entity's entry point has `void` return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

Data Receive Error Event – a runnable entity invoked by RTA-RTE when a data receive error occurs. The function providing the runnable entity's entry point has `void` return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

Data Received Event – a runnable entity invoked by RTA-RTE when a signal is received. The function providing the runnable entity's entry point has `void` return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

Data Send Completed Event – a runnable entity invoked by RTA-RTE when a transmission completes. The function providing the runnable entity's entry point has `void` return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

Mode Switch Event – a runnable entity invoked by RTA-RTE either on entry to, or on exit from, a mode. The function providing the runnable entity's entry point has `void` return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

Uniquely, RTA-RTE supports a "fast init" activation method for Mode Switch Events that bypasses the normal AUTOSAR activation method for events that are triggered only once at system startup. See Section 7.3 below for further details.

Mode Switched Ack Event – a runnable entity invoked by RTA-RTE when a mode transition is complete. The function providing the runnable entity's entry point has `void` return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

Operation Invoked Event – a runnable entity that implements an operation in a client-server interface. The runnable entity's entry function can take zero or more parameters in addition to the instance handle depending on the input configuration and has a return type of `void` or `Std_ReturnType` depending on whether or not the server returns an application error..

Timing Event – Runnable entity triggered by time. The function providing the runnable entity's entry point has void return type and takes no parameters (other than an instance handle if the software-component supports multiple instantiation).

The name for an runnable entity's entry point function contained within the <SYMBOL> element in the software-component description. The application header file includes a prototype for the required entry point function.

All runnable entities, whatever the event that triggers the runnable, must be declared in the input XML.

7.2 Signature

Apart from runnables triggered by an OperationInvoked event all functions implementing a runnable entity have the same signature:

```
FUNC(void, RTE_APPL_CODE)
<name>([IN Rte_Instance <instance>])
```

Where <name> is the runnable entity's <SYMBOL> and <instance> is a user-defined name for the instance handle. The instance handle is omitted if the software-component is declared as not supporting multiple instantiation.

The function implementing the runnable should be declared as RTE_APPL_CODE.

The function implementing an OperationInvoked event must declare a formal parameter list matching the operation arguments. The formal parameter list can include port-defined arguments or operation arguments:

```
( FUNC(void, RTE_APPL_CODE)
| FUNC(Std_ReturnType, RTE_APPL_CODE )
<name>([IN Rte_Instance <instance>],
       [IN <portDefArg_1>, ...
        IN <portDefArg_n>]
       [IN|INOUT|OUT <param_1>, ...
        IN|INOUT|OUT <param_n>] )
```

Where <name> is the runnable entity's <SYMBOL> and <instance> is a user-defined name for the instance handle. The instance handle is omitted if the software-component is declared as not supporting multiple instantiation.

For an OperationInvoked event, the return type must be FUNC(void, RTE_APPL_CODE) except where the interface declares one or more application errors in which case the type must be FUNC(Std_ReturnType, RTE_APPL_CODE).

All instances of a component are independent however the runnable entities of each instance share the same code. As a consequence multiple threads of control can be

concurrently executing the same runnable entity and therefore either the runnable entity must be reentrant or multiple instances must be explicitly forbidden.

7.3 SWC Initialization

AUTOSAR modes can be used to execute code when the RTE is started, e.g. to initialise internal data structures etc. Each mode declaration group describes an initial mode – to activate a runnable when the system is started created by a `<MODE-SWITCH-EVENT>` for entry to the initial mode.

A runnable entity within a software-component can be started when the RTE is started by declaring a `<MODE-SWITCH-EVENT>` for entry to an initial mode.

However for a runnable that only runs once at system start the infrastructure code created by RTA-RTE to support the AUTOSAR mode mechanism is superfluous. Therefore RTA-RTE provides the *FastInit* mechanism to optimize activation for specified ModeSwitchEvents by mapping them to specified *FastInit* tasks as simple function calls. The use of a *FastInit* task both simplifies the activation code necessary for the runnable associated with the event as well as removing the normal infrastructure code created by RTA-RTE to support the AUTOSAR mode mechanism.

The *FastInit* mechanism must be enabled for each applicable ModeSwitchEvent using the `--fast-init` command-line option.

An event that is specified as FastInit will not be activated by RTA-RTE during a user-triggered mode switch. Instead the FastInit task must be activated by user code at the appriopriate time.

8 VFB Tracing

When VFB tracing is enabled within the RTE module configuration (see Section 4.19.2) RTA-RTE inserts VFB Trace event hook calls into generated code. However VFB tracing must also be globally enabled within the user-supplied configuration file and the specific required hooks enabled for tracing to have an effect.

8.1 Enabling VFB Tracing

During RTE generation phase, the RTE generator creates a header file, `Rte_Hook.h`, that defines the signature and usage of VFB Trace event hook calls. The generated `Rte_Hook.h` file `#includes` the user-supplied configuration file `Rte_Cfg.h` that:

- Globally enables or disables all VFB Trace events.
- Enables individual VFB Trace events.

8.1.1 Global Enable

The following definition, within `Rte_Cfg.h`, globally disables all VFB Trace events:

```
#define RTE_VFB_TRACE (0)
```

And the following definition globally enables VFB Trace events:

```
#define RTE_VFB_TRACE (1)
```

When VFB Tracing is globally disabled no VFB Trace hooks will be made (irrespective of the configuration of individual trace events) and the VFB Trace hooks embedded within the generated RTE by RTA-RTE will have zero run-time impact.

When VFB Tracing is globally enabled individual trace events are enabled within `Rte_Cfg.h` by defining the name of the trace event hook function. For example, the “OS Task Dispatch” trace event is enabled by the following definition in `Rte_Cfg.h`:

```
#define Rte_Task_Dispatch
```

Whereas the “API Start” trace event for software-component type swc, port pa, data item d2 and API call `Rte_Write` would be enabled by the following definition in `Rte_Cfg.h`:

```
#define Rte_WriteHook_swc_pa_d2_Start
```

8.2 Trace Events

8.2.1 RTE API Start

RTE API Start is invoked by the RTE when an API call is made by a software component.

```
void
Rte_<api>Hook_<c>_<ap>_Start([IN Rte_Instance <inst>,]
                                <params>)
```

Where <api> is the API root name (`Write`, `Call`, `Feedback`, etc.), <c> the software-component type name and <ap> the access point (combination of port name and data item name/operation name separated by an underscore).

The parameters of the RTE API Start trace event hook are the same as the corresponding RTE API. The Instance handle is elided when a software-component is singly instantiated.

8.2.2 RTE API Return

RTE API Return is invoked by the RTE just before an API call returns control to a component.

```
void
Rte_<api>Hook_<c>_<ap>_Return([IN Rte_Instance <inst>,] <params>)
```

Where <api> is the API root name (`Write`, `Call`, `Feedback`, etc.), <swc> the software-component type name and <ap> the access point (combination of port name and data item name/operation name separated by an underscore).

The parameters of the RTE API Return trace event hook are the same as the corresponding RTE API. The Instance handle is elided when a software-component is singly instantiated.

8.2.3 Signal Transmission

A trace event indicating a transmission request of an Inter-ECU signal or signal group by the RTE. Invoked by the RTE just before `Com_SendSignal` or `Com_SendSignalGroup` is invoked.

```
void
Rte_ComHook_<signal>_SigTx(<data>)
```

Where `<signal>` is the system signal name and `<data>` is a pointer to the signal data to be transmitted.

For a system signal, the parameter of the signal transmission trace event hook is the data to be transmitted.

8.2.4 Signal Reception

A trace event indicating an attempt to read Inter-ECU signal by the RTE. Invoked by the RTE after successful return from Com_ReceiveSignal.

```
void
Rte_ComHook_<signal>_SigRx(<data>)
```

Where `<signal>` is the system signal name and `<data>` is a pointer to the signal data received.

The parameter of the signal reception trace event hook is the data received.

8.2.5 COM Notification

A trace event indicating the start of a COM notification. Invoked by the generated RTE code on entry to the COM call-back.

```
void
Rte_ComHook_<signal>(void)
```

Where <signal> is the system signal name.

8.2.6 OS Task Activation

A trace event invoked by the RTE immediately before activating the specified task.

```
void
Rte_Task_Activate(TaskType t)
```

Where <t> is the handle of the task to be activated.

8.2.7 OS Task Dispatch

A trace event invoked by the RTE immediately on dispatch of the specified generated task (provided it contains runnable entities).

```
void
Rte_Task_Dispatch(TaskType t)
```

Where <t> is tha handle of the task to be activated.

8.2.8 OS Task Set Event

A trace event invoked immediately before generated RTE code attempts to set an OS Event.

```
void
Rte_Task_SetEvent(TaskType task, EventType ev)
```

Where <t> is tha handle of the task to be activated and <ev> the Os event to be set.

8.2.9 OS Task Wait Event

A trace event invoked immediately before generated RTE code attempts to wait for an OS Event.

```
void
Rte_Task_WaitEvent(TaskType task, EventType ev)
```

Where <t> is tha handle of the task to be activated and <ev> the Os event(s).

8.2.10 OS Task Wait Event Return

A trace event invoked immediately before generated RTE code returns from waiting for an OS Event.

```
void
Rte_Task_WaitEventRet(TaskType task, EventType ev)
```

Where <t> is tha handle of the task to be activated and <ev> the set Os event(s).

8.2.11 Runnable Entity Invocation

Event invoked by the RTE just before execution of runnable entry starts via its entry point. This trace event occurs after any copies of data elements are made to support the Rte_IRead API Call.

```
void
Rte_Runnable_<swc>_<reName>_Start(IN Rte_Instance <inst>)
```

Where <swc> is the software-component type name and <reName> the runnable entity name.

8.2.12 Runnable Entity Termination

Event invoked by the RTE immediately execution returns to RTE code from a runnable entity. This trace event occurs before write-back of data elements are made to support the Rte_IWrite API Call.

```
void
Rte_Runnable_<c>_<reName>_Return([IN Rte_Instance <inst>)]
```

Where <swc> is the software-component type name and <reName> the runnable entity name.

8.3 Trace Event Implementation

The implementation of a trace event hook function is entirely within the user domain – all that is necessary to supply when linking is a function implementation that conforms to one of the trace event signatures defined above.

For example, the task dispatch event might be implemented as follows:

```
#include <Os.h>
#include Rte_Hook.h''

void Rte_Task_Dispatch(TaskType t)
{
   if ( t == taskA )
   {
      /* Log task A dispatch */
   }
}
```

8.4 Optimization

Enabling VFB tracing, either via the ECU configuration description or the command-line option, disables certain optimizations within the generated RTE:

- Intra-task (and inter-task to a ‘pure’ server) client-server communication is not optimized to a direct function call.

9 Memory Mapping and Compiler Abstraction

9.1 Memory mapping principles

RTA-RTE-generated code and data allocations are placed into memory sections using the AUTOSAR memory mapping scheme. Declarations and definitions of code and data objects are wrapped with *Memory Allocation Keyword*s (*MAKW*s), allowing RTE code to remain target-independent. You must supply or generate the `Rte_MemMap.h` file that provides the compiler-specific memory mapping implementation (e.g. emitting `#pragma`s, redefining memory section macros).

9.1.1 Format of Memory Allocation Keywords

Regardless of the type of object, all MAKWs generated by RTA-RTE conform to a standard format:

```
<PREFIX>_[<INFIX>_]<ACTION>_<SECNAME>
```

A MAKW's `<ACTION>` is either `START_SEC` for MAKWs that denote the start of a memory section, or `STOP_SEC` for MAKWs that denote the end of a section. Within the generated code, each `START_SEC` MAKW will be followed by a corresponding `STOP_SEC` MAKW. The MAKW system does not support nesting, so a section must be stopped before a new one can be started.

9.2 Memory mapping for code objects

All code objects generated by RTA-RTE are wrapped in MAKWs where the `<PREFIX>` is `RTE`. With the exception of task bodies that have a specific `SwAddrMethod` assigned, code objects use the `SwAddrMethod` `CODE`, and thefore the MAKW `<SECNAME>` is also `CODE`. All MAKWs associated with a code object also contain an infix that is used to identify the context in which the generated code will be executed.

The MAKW infix and the associated memory section information in the `Rte_BSWMD.arxml` report can help with the configuration of a memory protection scheme to ensure freedom from interference requirements are satisfied. For code objects, this information is always generated. If isolation of code objects is not required in your system, you can configure your memory mapping to allocate multiple AUTOSAR `MemorySections` to one physical memory section.

When configuring a memory protection scheme, you must ensure that all code objects are accessible from all contexts in which they will be executed, including trusted contexts if applicable.

The infixes used for the various categories of code objects are as follows:

9.2.1 SWCT-related code

For APIs and other generated functions associated with a specific SWCT, the section identification infix is the `ShortName` of that SWCT. Each generated API is wrapped with `RTE_<SWCT>_START_SEC_CODE` and `RTE_<SWCT>_STOP_SEC_CODE`.

```
/* RTE API FUNCTION PROTOTYPES */
```

```
#define RTE_SWCA_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(Std_ReturnType, RTE_CODE) Rte_<API>(...);
#define RTE_SWCA_STOP_SEC_CODE
#include "Rte_MemMap.h"

/* RTE API FUNCTION BODIES */
#define RTE_SWCA_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(Std_ReturnType, RTE_CODE) Rte_<API>(...)
{
   ...
}
#define RTE_SWCA_STOP_SEC_CODE
#include "Rte_MemMap.h"
```

9.2.2 BSWMD-related code

For Basic Software Scheduler (SchM) APIs associated with a specific BSWMD, the section identification infix is the ShortName of that BSWMD. Each generated API is wrapped with RTE_<BSWMD>_START_SEC_CODE and RTE_<BSWMD>_STOP_SEC_CODE.

```
/* RTE API FUNCTION PROTOTYPES */
#define RTE_BSWA_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(Std_ReturnType, RTE_CODE) SchM_<API>(...);
#define RTE_BSWA_STOP_SEC_CODE
#include "Rte_MemMap.h"

/* RTE API FUNCTION BODIES */
#define RTE_BSWA_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(Std_ReturnType, RTE_CODE) SchM_<API>(...)
{
   ...
}
#define RTE_BSWA_STOP_SEC_CODE
#include "Rte_MemMap.h"
```

9.2.3 Task bodies

Each task body is wrapped with RTE_<TASK>_START_SEC_<NAME> and RTE_<TASK>_STOP_SEC_<NAME>.

For task bodies, the task name is used as the section identification infix. By default, task bodies use the default SwAddrMethod CODE, so the <NAME> portion of the MAKW is also CODE.

A user-specified SwAddrMethod applies to a task body when its corresponding OsTask contains an OsMemoryMappingCodeLocationRef. When a user-specified SwAddrMethod applies, the <NAME> portion of the MAKW is the shortName of the referenced SwAddrMethod.

```
/* RTE_TASKBODIES_START */

/* The OsTask for taskTx does not reference   /
/  a swAddrMethod (default "CODE" assumed)   */

#define RTE_TASKTX_START_SEC_CODE
#include "Rte_MemMap.h"
TASK(taskTx)
{
   ...
}
#define RTE_TASKTX_STOP_SEC_CODE
#include "Rte_MemMap.h"

/* The OsTask object for taskRx references a  /
/  swAddrMethod with shortName "TaskRxSADM"  */

#define RTE_TASKRX_START_SEC_TaskRxSADM
#include "Rte_MemMap.h"
TASK(taskRx)
{
   ...
}
#define RTE_TASKRX_STOP_SEC_TaskRxSADM
#include "Rte_MemMap.h"
```

9.2.4 Communication callbacks

COM and LdCom callback functions documented in sections 5.9.1 and 5.9.2 of [SWS_Rte] (e.g. Rte_COMCbk_<sn>) use the section identification infix SIG_<sn>, where <sn> is the name of the COM signal, COM signal group or LdComIPdu associated with the callback function.

```
/* RTE CALLBACKS */
#define RTE_SIG_MYSIGNAL_UINT32_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(void, RTE_CODE) Rte_COMCbk_MySignal_UInt32(void)
{
   ...
}
#define RTE_SIG_MYSIGNAL_UINT32_STOP_SEC_CODE
#include "Rte_MemMap.h"
```

9.2.5 OS callbacks

OS callback functions (including alarm callbacks and IOC pull receiver callbacks) use the section identification infix OSCBK_<cbkname>, where <cbkname> is the name of the function generated by RTA-RTE to implement the callback.

```
#define RTE_OSCBK_RTE_TIMEOUTALARM2_CBK_START_SEC_CODE
#include "Rte_MemMap.h"
ALARMCALLBACK( Rte_TimeoutAlarm2_Cbk )
{
   ...
}
#define RTE_OSCBK_RTE_TIMEOUTALARM2_CBK_STOP_SEC_CODE
#include "Rte_MemMap.h"
```

9.2.6 NVM access code

Code that is used to access an NvBlockDescriptor in an NvBlockSwComponentType uses the section identification infix <SWCT>_<NvBlockDescriptorName>.

NVM access code includes the NVM service callbacks documented in section 5.9.3 of [SWS_Rte]. As Rte_GetMirror is expected to be executed in a different context from the other NVM callbacks, it is further differentiated by an additional NVM_ in its section identification infix, meaning that it uses the infix NVM_<SWCT>_<NvBlockDescriptorName>.

In both cases, <SWCT> is the name of the NvBlockSwComponentType and <NvBlockDescriptorName> is the name of the NvBlockDescriptor for which the code is generated.

9.2.7 Lifecycle APIs

The RTE lifecycle APIs (enumerated in section 5.7 of [SWS_Rte]), Basic Software Scheduler (SchM) lifecycle APIs (section 6.7 of [SWS_Rte]), and the function Rte_MainFunction use the section identification infix MAIN, resulting in them being wrapped with RTE_MAIN_START_SEC_CODE and RTE_MAIN_STOP_SEC_CODE

```
/* RTE LIFECYCLE API FUNCTION PROTOTYPES */
#define RTE_MAIN_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(Std_ReturnType, RTE_CODE) Rte_Start(void);
#define RTE_MAIN_STOP_SEC_CODE
#include "Rte_MemMap.h"

/* RTE LIFECYCLE API FUNCTION BODIES */
#define RTE_MAIN_START_SEC_CODE
#include "Rte_MemMap.h"
FUNC(Std_ReturnType, RTE_CODE) Rte_Start(void)
{
```

```
    ...
}
#define RTE_MAIN_STOP_SEC_CODE
#include "Rte_MemMap.h"
```

9.2.8 RTE library code

RTE code not related to any SWC, task or callback is assigned one of three possible section identification infixes depending on the behaviour of the code:

- If the function modifies global state, the infix used is SYS, giving MAKWs RTE_SYS_START_SEC_CODE and RTE_SYS_STOP_SEC_CODE.
- If the function calls OS or BSW functions, the infix is EXT, giving MAKWs RTE_EXT_START_SEC_CODE and RTE_EXT_STOP_SEC_CODE.
- If the function neither modifies global state nor calls OS/BSW functions, the infix is LIB, giving MAKWs RTE_LIB_START_SEC_CODE and RTE_LIB_STOP_SEC_CODE.

The SYS category is the highest priority, so if a function both modifies global state directly and calls OS or BSW functions, it is considered to belong to the SYS group.

9.3 Memory mapping for data objects

Data objects generated by RTA-RTE are wrapped in MAKWs where the <PREFIX> is RTE and the <SECNAME> is derived from the SwAddrMethod referenced by the AutosarDataPrototype.

If the SwAddrMethod has a memoryAllocationKeywordPolicy of addrMethodShortName, the resulting <SECNAME> is the SwAddrMethod's shortName.

If the SwAddrMethod has a memoryAllocationKeywordPolicy of addrMethodShortNameAndAlignment, the resulting <SECNAME> is the SwAddrMethod's shortName plus one of the following alignment suffixes:

- _BOOLEAN
- _8
- _16
- _32
- _64
- _UNSPECIFIED

Consequently, if a configuration results in a global buffer for a VariableDataProtype associated with a SwAddrMethod named group1, and that SwAddrMethod has a memoryAllocationKeywordPolicy of swAddrMethodShortName, RTA-RTE generates the following MAKWs around the declaration or definition of the global buffer:

```
#define RTE_START_SEC_group1
#include "Rte_MemMap.h"
...
#define RTE_STOP_SEC_group1
#include "Rte_MemMap.h"
```

9.3.1 Default SwAddrMethods

When data implied by the input model has no associated SwAddrMethod, RTA-RTE uses a default SwAddrMethod according to the AUTOSAR Memory Mapping specification:

- VAR_CLEARED for variables that are uninitialized, or initialized to zero.
- VAR_INIT for variables with initial values.
- CONST for constants.
- CALIB for calibration parameters.
- CODE for code.

These SwAddrMethods will always exist in any RTE implementation generated by RTA-RTE. If the input configuration provides a SwAddrMethod with the same shortName as any of these defaults, the provided and internal definitions must be compatible. RTA-RTE will raise a warning if this is not the case.

9.3.2 SwAddrMethods for Mode Machine Instance variables

The SwAddrMethod for Mode Machine Instance variables (ModeCurrent, ModeNext, ModeQueue, ModeQindex, ModeOld and ModeStatus) is the first-encountered SwAddrMethod applied to the ImplementationDataType that is mapped to the associated ModeDeclarationGroup.

Where an ImplementationDataType is typed by referencing another ImplementationDataType, RTA-RTE will follow these references until a SwAddrMethod is found. If no user defined SwAddrMethod is found, RTA-RTE will apply a default SwAddrMethod.

To allow specific Mode Machine Instances to be allocated to identifiable memory sections, a SwAddrMethod can be assigned directly using a FlatInstanceDescriptor.

This FlatInstanceDescriptor references the ModeDeclarationGroupPrototype in the PPortPrototype of the ModeManager, and has the role MMI_SWADDRMETHOD. Note that this role is specific to RTA-RTE and is not specified by AUTOSAR.

The SwAddrMethod referenced by this FlatInstanceDescriptor will be used in preference to any SwAddrMethod that is associated with the MMI's mapped ImplementationDataType.

9.3.3 Context-specific MAKWs for data objects

To provide support for freedom from interference (FFI) in a safety-relevant system, RTA-RTE allows the generation of data objects into context-specific memory sections. Memory protection can then be configured to ensure that write access to those data objects is not permitted from outside the relevant execution context.

Objects that are always allocated to context-specific sections

Some types of data object are always allocated to context-specific sections in the generated RTE when the input configuration contains `EcucPartitions`:

For the following objects, the MAKW <INFIX> is set to the `shortName` of the `EcucPartition` from which the objects are accessed:

- Queue variables (queue buffer and dynamic structure) used for queued communication within a single partition.
- Mode machine instance variables used for mode switching
- Runnable activation flags

The MAKW <SECNAME> for these objects is set based on the applicable `SwAddrMethod`, therefore the generated MAKW will have the form:

```
#define RTE_<EcucPartition>_START_SEC_<SwAddrMethod>[_<Alignment>]
```

The following objects use context-specific memory sections where the MAKW <INFIX> is not based on the name of the `EcucPartition`:

- Resource lock counter variables - the <INFIX> is `shortName` of the `OsApplication` associated with the resource
- Resource handle arrays and resource lock counter arrays - these use the <INFIX> SYS
- Prescalar counter variables - the <INFIX> is the `shortName` of the `OsTask` in which the prescalar variables are used.
- Variables that store `ArTypedPerInstanceMemorys` when they are aggregated by a `BswInternalBehavior` - the <INFIX> here is the `shortName` of the parent `BswModuleDescription`

The `--use-partition-sections` option has no effect on data objects that are always allocated to context-specific memory sections. For these objects the context-specific MAKW infix is always generated. If isolation of these objects is not required in your system, you can configure your memory mapping to allocate multiple AUTOSAR `MemorySections` to one physical memory section.

Objects that are optionally allocated to context-specific sections

The remaining types of variable data object can optionally be allocated to partition-specific memory sections by enabling the `--use-partition-sections` option.

When `--use-partition-sections` is enabled, the MAKW <SECNAME> for these objects is prepended with either the `shortName` of the `EcucPartition` associated with those objects, or the `shortName` of the associated `OsApplication` where it is not mapped to an `EcucPartition`:

```
#define RTE_START_SEC_<EcucPartition>_<SwAddrMethod>[_<Alignment>]
```

or

```
#define RTE_START_SEC_<OsApp>_<SwAddrMethod>[_<Alignment>]
```

9.4 Reporting RTE objects to other AUTOSAR tooling

Code and data objects created by RTA-RTE are described in the `Rte_BSWMD.arxml` report which is produced along with the generated RTE implementation.

Each `SwAddrMethod` used in the generated RTE will be described in the `Rte_BSWMD.arxml` report, inside an `ARPackage` named `SwAddrMethods`.

Within the `Rte_BSWMD.arxml`, information about the memory sections used by the generated RTE implementation can be found in a `ResourceConsumption` with `shortName` `ResConsumption`

This information can be used in the process of generating the `Rte_MemMap.h` memory mapping file.

9.4.1 Reporting of SectionNamePrefixes

For each MAKW containing an <INFIX> that is used in the generated RTE, a `SectionNamePrefix` will be emitted inside the `ResourceConsumption` named `ResConsumption` in the generated `Rte_BSWMD.arxml` file, with `symbol` and `shortName` both equal to RTE_<INFIX>:

```
<RESOURCE-CONSUMPTION>
  <SHORT-NAME>ResConsumption</SHORT-NAME>

  <MEMORY-SECTIONS>
  ...
  </MEMORY-SECTIONS>

  <SECTION-NAME-PREFIXS>
    <SECTION-NAME-PREFIX>
      <SHORT-NAME>RTE_MAIN</SHORT-NAME>
      <SYMBOL>RTE_MAIN</SYMBOL>
    </SECTION-NAME-PREFIX>
  </SECTION-NAME-PREFIXS>

</RESOURCE-CONSUMPTION>
```

No SectionNamePrefix is emitted for MAKWs that do not contain an <INFIX>, and the corresponding MemorySections do not reference a prefix. The default prefix of RTE will apply to RTE-related objects that do not reference any other prefix.

9.4.2 Reporting of MemorySections

For each MAKW used in the generated RTE, a corresponding MemorySection will be emitted inside the ResourceConsumption in the generated Rte_BSWMD.arxml file.

If the MAKW contains an <INFIX>, the MemorySection will:

- Have shortName RTE[_<INFIX>]_<SWADDRMETHOD>[_<ALIGNMENT>]
- Have alignment <ALIGNMENT>
- Reference the RTE_<INFIX> SectionNamePrefix
- Reference the <SWADDRMETHOD> SwAddrMethod
- Have symbol <SWADDRMETHOD>[_<ALIGNMENT>]

If the MAKW does not contain an <INFIX>, the MemorySection will:

- Have shortName <SWADDRMETHOD>[_<ALIGNMENT>]
- Have alignment <ALIGNMENT>
- Reference the <SWADDRMETHOD> SwAddrMethod
- Have symbol <SWADDRMETHOD>[_<ALIGNMENT>]

The optional [_<ALIGNMENT>] in the MemorySection symbol and shortName exists if an alignment applies to the objects contained in that section. Even where an alignment applies, it will not affect the generated MemorySection if the referenced SwAddrMethod has a memoryAllocationKeywordPolicy of addrMethodShortName.

If the MemorySection has code objects allocated to it, it will also reference an ExecutableEntity for each C symbol contained in that memory section:

```
<RESOURCE-CONSUMPTION>
  <SHORT-NAME>ResConsumption</SHORT-NAME>

  <MEMORY-SECTIONS>
    <MEMORY-SECTION>
      <SHORT-NAME>RTE_MAIN_CODE</SHORT-NAME>
      <EXECUTABLE-ENTITY-REFS>
        <EXECUTABLE-ENTITY-REF BASE="Rte_BSWMD_BswModuleDescriptions"
                               DEST="BSW-CALLED-ENTITY">
          Rte/RteInternalBehavior/Rte_Start
```

```
        </EXECUTABLE-ENTITY-REF>
        <EXECUTABLE-ENTITY-REF BASE="Rte_BSWMD_BswModuleDescriptions"
                               DEST="BSW-CALLED-ENTITY">
          Rte/RteInternalBehavior/Rte_Stop
        </EXECUTABLE-ENTITY-REF>
      </EXECUTABLE-ENTITY-REFS>
      <PREFIX-REF BASE="Rte_BSWMD_BswImplementations"
                  DEST="SECTION-NAME-PREFIX">
        Rte/ResConsumption/RTE_MAIN
      </PREFIX-REF>
      <SW-ADDRMETHOD-REF BASE="Rte_BSWMD_SwAddrMethods"
                         DEST="SW-ADDR-METHOD">
        CODE
      </SW-ADDRMETHOD-REF>
      <SYMBOL>CODE</SYMBOL>
    </MEMORY-SECTION>
  </MEMORY-SECTIONS>

  <SECTION-NAME-PREFIXS>
  ...
  </SECTION-NAME-PREFIXS>

</RESOURCE-CONSUMPTION>
```

9.4.3 Generation of Rte_MemMap.h

In a typical system, the Rte_MemMap.h will be generated automatically, using the MemorySection information contained in Rte_BSWMD.arxml. Each MemorySection is associated with one START_SEC and one STOP_SEC MAKW (each of which may be used multiple times in the generated RTE code).

The MAKWs can be constructed from the MemorySection data following the rules set out by AUTOSAR in [SWS_MemMap_00022].

By default RTA-RTE uses the Memory Allocation Sequence both for data allocations and for external declarations. Some compilers issue warnings for this; others issue warnings if the declarations differ from the allocations. The --deviate-memmap-decls=0 option can be used to disable the generation of MAKWs around declarations.

9.4.4 ApplicationSwComponentType-specific MemMap.h

RTA-RTE expects a <shortName>_MemMap.h for each ApplicationSwComponentType used in the ECU configuration. This file is for the entry points of the RunnableEntitys in the ASW Component. This feature does not apply to entry points in BSW compoments.

RTA-RTE can generate skeleton ‹*shortName*›_MemMap.h files for associated SWCs. Use the --samples option to do this.

The sample memory map files generated by RTA-RTE are intended as examples only and should be verified as suitable for the chosen target development hardware configuration before use.

9.4.5 Variation: Implicit Communication

When option `--implicit-allocation-method` is enabled, the task-specific implicit buffer structures are placed according to `SwAddrMethods` named `VAR_IMPLICITSR_`‹*TASKNAME*› where ‹*TASKNAME*› is the `OsTask.shortName` converted to upper case.

9.5 Compiler Abstraction

The AUTOSAR compiler abstraction enables objects located with the memory mapping to be referenced. This is achieved through the definition of object classes within the file `Compiler_Cfg.h`.

9.5.1 Defined Classes

RTA-RTE defines the following object classes:

`RTE_CONST` – Memory class used to access objects declared within the generated RTE using RTE_*_SEC_CONST_*.

`RTE_DATA` – Memory class used to access objects declared within the generated RTE using RTE_*_SEC_DATA_*.

`RTE_CODE` – Memory class used to access (i.e. create function pointers) to objects declared within the generated RTE using RTE_*_SEC_CODE.

`RTE_APPL_CONST` – Memory class uses to access constant data within an application, typically via a pointer passed into an RTE API.

`RTE_APPL_DATA` – Memory class uses to access data within an application, typically via a pointer passed into an RTE API.

`RTE_APPL_CODE` – Memory class used to access (i.e. create function pointers) to objects declared within the application code space.

`RTE_OS_CDATA` – Memory class used to access constant data within the operating system.

`RTE_LIBCODE` – Memory class used to declare and access RTE library functions.

No object class exists, or is required by the generated RTE, for access to objects created with the `TASKBODY` memory class.

9.5.2 Customization

RTA-RTE permits customization of the compiler abstraction through an XML *Memory Section Description File*. See *RTA-RTE Toolchain Integration Guide* for more details.

10 External Dependencies

RTA-RTE references symbols that are in the scope of external libraries and other AUTOSAR modules, in particular:

- C library (optional).
- OS Configuration.
- COM APIs (optional, only required when inter-ECU communication is present).
- OS APIs.
- Calibration parameters.

10.1 C Library

By default, RTA-RTE is independent of the C library and uses the RTE library define function `Rte_memcpy` to copy memory.

Alternatively, RTA-RTE can be configured to use the C library's `memcpy` function if the symbol `RTE_LIBC_MEMCPY` is defined when compiling both the RTE library and generated code.

The standard `memcpy` function from the C Library may be preferred for certain targets, for example, to reduce code size by sharing `memcpy` with other modules that already use the C Library or to improve the efficiency of compiled code by using a version of the `memcpy` function "built-in" to the compiler that produces optimal inline assembler.

10.2 OS Configuration

The RTE and OS for a particular ECU have a complex relationship in that the OS configuration partly influences the RTE and the RTE configuration partly influences the OS. At the minimum the names and priorities of the Os tasks that will execute the runnable entities and the Os tasks and Os ISRs that will execute the schedulable entities are necessary as an input to RTE generation. This information defines the concurrency model of the system and defines the set of tasks whose bodies are part of the generated RTE. The RTE will however require additional Os objects, for example to schedule those tasks, to provide for the execution of only the activated runnables during particular task executions and to serialize access to application / BSW module code and internal state, depending on the configuration of the involved software components and BSW modules. Furthermore the configuration may require particular settings for the Os tasks, for example an Os task that executes at least one runnable entity with a WaitPoint may need to be an extended task.

RTA-RTE optionally generates an OS configuration file (see Section 2.3) containing definitions for all OS objects used by the generated RTE and the necessary configuration for those Os tasks that have runnable entities or schedulable entities mapped to them. If this file is not used during OS configuration then the following OS objects will be required to be created:

10.2.1 Resources

`RTE_RESOURCE*` OsResources to support RTE APIs.

RTA-RTE uses an RTE-specific OsResource with an OsResourceProperty of `STANDARD` to serialize access to internal RTE state.

For systems that define only a single `OsApplication`, or do not define any `OsApplications` at all, RTA-RTE requires one resource named `RTE_RESOURCE`.

Where a system defines multiple `OsApplications`, RTA-RTE requires one resource for each `OsApplication`, named `RTE_RESOURCE_OS_APP_`*OsApp.shortName*.

The required resources must be included in the configuration passed to the Operating System. For single-`OsApplication` systems, RTA-RTE will include the configuration of `RTE_RESOURCE` in the OsNeeds report if it is not already present in the input configuration. For multi-`OsApplication` systems, the system integrator must configure the required resources manually.

Any task containing code that uses an RTE API needs to be declared as locking `RTE_RESOURCE`, or the `RTE_RESOURCE_OS_APP_`<NAME> for the `OsApplication` in which the task executes. Additionally, any task or ISR in whose context BSW modules may call back into RTE-generated code for notifications (e.g. for notification of COM reception) must also be declared as locking this resource. Failure to do so may result in runtime errors.

`Rte_EA_<i>_<n>/Rte_EI_<i>_<n>` An AUTOSAR standard / internal resource created to ensure serialized access to exclusive area <n> within runnable entities from SW-C instance <i>.

The name of the SW-C prototype, <i>, is an internal name created by the RTA-RTE RTE generator. The mapping between user visible component prototype name and internal name is included in `Rte_Const.h`.

RTA-RTE will not create a resource if an existing resource is specified within the ECUC file (see Section 4.19.2).

RTA-RTE will optimize away the use of an OS resource if either all accessing runnable entities are mapped to the same task or all accessing runnable entities are mapped to tasks with the same priority. If the use of the resource is optimized away by RTA-RTE then no OS object need be created.

10.2.2 Schedule Tables

`Rte_ScheduleTable` An AUTOSAR OS Schedule table used within the generated RTE for execution of tasks containing runnable entities triggered by Timing RTE events.

10.2.3 Counters

The RTA-RTE RTE generator uses counters for driving alarms required by generated code.

`Rte_Tick_Counter` An AUTOSAR OS Counter used by application code to control execution of `Rte_ScheduleTable` or generated periodic alarms.

The required counter tick rate depends on periods of Timing RTE events and is indicated by an RTA-RTE information message and by a definition within the generated file `Rte_Const.h`.

`Rte_TOut_Counter` An AUTOSAR OS Counter used by application code to control execution of sporadic alarms, e.g. for timeouts.

The required counter tick rate is fixed at 1ms and is defined within the generated file `Rte_Const.h`.

To prevent missed runnable activations and event settings it is important not to tick `Rte_Tout_Counter` directly but instead to call the generated RTE API `Rte_Tick_Timeouts` instead, see Section 6.35.

10.2.4 Alarms

The RTA-RTE RTE generator uses alarms for handling timeouts, for scheduling sporadic activities and, optionally, for scheduling periodic activities.

`Rte_<task>_timeoutAlarm` An AUTOSAR OS Alarm used within the generated RTE to indicate a timeout for a runnable mapped to OS task <task>.

`Rte_Alarm_<i>_<p>_<d>` Sporadic alarm created to indicate a timeout for SWC instance <i>, port <p> and datum/mode group <d>.

`rte_alAct_<p>_<o>` or `rte_alAct_<ev>` Periodic alarm defined for a TimingEvent.

If the ECUC associates OS event <ev> with the timing event instance has a defined OS event then the _<ev> form of the name is used. Otherwise the alarm name includes the timing event period <p> and offset <o>.

To prevemt overlong names, this release of RTA-RTE assigns internal names to alarms rather than constructing names according to the scheme above. Consult the generated OS configuration for details.

10.2.5 Events

The RTA-RTE RTE generator uses OS events both for internal task scheduling and for handling the wake-up-of-waitpoint receive mode.

`Rte_Activity` An AUTOSAR OS event used within the generated RTE for Wake-up-of-waitpoint handling to indicate that an RTE Event has occurred.

The OS event must be referenced by any task that contains runnable entities that can wait.

The event mask must be ’1’ and must be the same for all tasks using the event.

`Rte_Timeout` An AUTOSAR OS event used within the generated RTE for Wake-up-of-waitpoint handling to indicate a timeout has occurred while waiting for an RTE event.

The event must be referenced by any task that contains runnable entities that can wait and have a timeout specified.

The event mask must be '2' and must be the same for all tasks using the event.

When required, the RTA-RTE RTE generator uses OS events to perform internal task scheduling. If the OS event name is not specified in the ECU description, the RTE generator constructs OS event names as follows:

`Rte_Ev_<period>_<offset>` An AUTOSAR OS event created to indicate a Timing RTE event has occurred when multiple runnable entities are mapped to a task. The `<period>` and `<offset>` are the period and offset of the Timing RTE event. All runnable entities within a task with the same period share the same event.

This OS event is only created when no user-supplied OS event is referenced within the RTE Event's runnable entity map and runnable entities triggered by Timing RTE events and other RTE events are mapped to the same task.

`Rte_Ev_<i>_<p>_<d>` An AUTOSAR OS event created to trigger runnable started by an event associated with SW-C prototype <i>, port <p> and datum <d>.

The name of the SW-C prototype, <i>, is the internal name not the name declared within the composition since the latter is not necessarily unique.

This OS event is only created when no user-supplied OS event is referenced within the RTE Event's runnable entity map and runnable entities triggered by Timing RTE events and other RTE events are mapped to the same task.

The names of OS events used for internal task scheduling can also be specified in the ECU description. If specified the ECU description name will override the default name defined in the table above.

10.3 AUTOSAR COM

AUTOSAR COM module is a mandatory requirement of RTA-RTE when inter-ECU communication is used (RTA-RTE implements intra-ECU communication without using COM).

10.3.1 Initialization

COM must be started before the RTE. Therefore the RTE is not responsible for initializing COM and `Rte_Start` does not invoke the `Com_Init` API.

RTA-RTE uses periodic transmission when "cyclic" or "n-times" communication is used. Therefore the `Com_StartPeriodic` API should be used to initialize periodic activity within COM in addition to invoking `Com_Init`.

10.3.2 Data Types

RTA-RTE uses AUTOSAR data types when communicating with COM. The definitions of these data types are defined in `Rte_Type.h` and this file can be used to make the type definitions visible to a COM implementation.

10.3.3 Transmission and Invalidation

RTA-RTE uses signals or signal groups for transmission and therefore the following COM APIs are used:

```
Com_SendSignal( Com_MessageIdentifier,
                Com_ApplicationDataRef )
Com_UpdateShadowSignal( Com_MessageIdentifier,
                        Com_ApplicationDataRef )
Com_InvalidateSignal( Com_MessageIdentifier )
Com_InvalidateShadowSignal( Com_MessageIdentifier )
Com_SendSignalGroup( Com_SignalGroupIdType )
```

10.3.4 Reception

RTA-RTE uses signals or signal groups for reception and therefore the following COM APIs are used:

```
Com_ReceiveSignal( Com_MessageIdentifier,
                   Com_ApplicationDataRef )
Com_ReceiveShadowSignal( Com_MessageIdentifier,
                         Com_ApplicationDataRef )
Com_ReceiveSignalGroup( Com_SignalGroupIdType )
```

10.3.5 Call-backs

RTA-RTE uses COM call-backs to receive notification when communication related events have occurred; for example data reception. The call-backs are created in `Rte.c` but must be declared in the COM configuration attached to the appropriate signal.

Callbacks are used for the following events:

Rte_COMCbk_<signal> – Data reception.

Rte_COMCbkTOut_<signal> – Data reception timeout.

Rte_COMCbkInv_<signal> – Invalid data reception.

Rte_COMCbkTAck_<signal> – Acknowledgement of data transmission.

10.4 Operating System

An Operating System module is a mandatory requirement of RTA-RTE.

10.4.1 Concurrency Control

Internally RTA-RTE uses an AUTOSAR OS resource for concurrency control. In a multi-core system this become an OS resource for each AUTOSAR core, for core-local concurrency control. Generated code can either use an OS resource or interrupt blocking depending on the configuration. The following OS APIs are therefore used to control concurrency:

```
GetResource( ResourceType )
ReleaseResource( ResourceType )
SuspendOSInterrupts( void )
ResumeOSInterrupts( void )
```

All OS tasks or ISRs that invoke RTE API functions must be declared as locking the standard resource RTE_RESOURCE in a single-core system. This is done automatically in the generated OS configuration for all generated tasks.

For a multi-core system, tasks and ISRs on the master core (with core ID zero) must be declared as locking RTE_RESOURCE and tasks and ISRs on slave cores (core ID greater than zero) must be delcared as locking the standard resources RTE_RESOURCE_CORE<n> where <n> is the core ID of the slave core.

10.4.2 Alarms

RTA-RTE uses alarms for periodic and sporadic events. The following OS APIs are used to manipulate alarms:

```
SetRelAlarm( AlarmType, TickType, TickType )
CancelAlarm( AlarmType )
```

10.4.3 Events

RTA-RTE uses OS events to implement certain RTE events, for example, APIs requiring a WaitPoint and for RTA-RTE code within generated task bodies. The following APIs are used to manipulate OS events:

```
SetEvent( TaskType, EventType )
WaitEvent( EventType )
GetEvent( TaskType, EventType* )
ClearEvent( EventType )
```

10.4.4 Tasks

RTA-RTE uses tasks to execute runnable entities. The following OS APIs are used to manipulate tasks:

```
ActivateTask( TaskType )
ChainTask( TaskType )
TerminateTask( void )
GetTaskID( void )
```

10.4.5 Schedule Tables

RTA-RTE uses an AUTOSAR schedule table to execute tasks containing runnable entities triggered by Timing RTE events. The following OS APIs are used to manipulate the schedule table:

```
// OS R1.0:
StartScheduleTable( ScheduleTableType, TickType )
// OS R3.0:
StartScheduleTableRel( ScheduleTableType, TickType )
StopScheduleTable( ScheduleTableType )
```

10.5 Calibration

RTA-RTE supports the following calibration methods:

- “None” – no software support is generated (RTE APIs for accessing calibration parameters access the allocated data directly). The RTE generator expects calibration parameters to be updated externally, e.g. by direct memory access from calibration hardware.
- “Single-pointered” – RTA-RTE generates a reference table (in RAM) that locates calibration data. The calibration values returned by individual generated RTE APIs can be modified by changing pointers in the reference table to point to alternate values in either RAM or ROM.
- “Double-pointered” – RTA-RTE generates a reference table (in ROM) and a reference base (in RAM) that points to the table. The calibration values returned by all generated RTE APIs can be modified by changing the base pointer to reference a different table.
- “Init-RAM” – RTA-RTE generates a RAM copy of all parameters and an initializing ROM block. Generated RTE APIs access the RAM copy.
- “Single-pointered2” – RTA-RTE generates individual reference pointers (in RAM) that locate calibration data. The calibration values returned by individual generated RTE APIs can be modified by changing those reference pointers. This is a non-AUTOSAR extension.

For the pointered methods, all calibration parameters within an SW-C that reference the same `swAddrMethod` are grouped within a *calibration parameter group*. Grouping can also optionally be enabled for method “None” using the `--deviate-group-calibration-none` command-line option.

If no <SW-ADDR-METHOD> is referenced the name `nullSWAddr` is used.

To be able to import or instantiate the required calibration parameter groups, RTA-RTE must define both the type and the instance names.

For the “Init-RAM” method the calibration parameters are also grouped in the same way but the groups are further collected into a structure for allocation of the RAM and ROM blocks. This allows a single copy operation to initialize all RAM blocks. To be able to instantiate the RAM and ROM blocks RTA-RTE must define the type and instance names of the RAM and ROM block structures, the type names for calibration parameter groups and the order and names of the elements in the RAM/ROM block structures.

10.5.1 Type Name

Non-grouped Parameters

When using the calibration method “None” RTA-RTE does not normally create calibration parameter groups and hence instantiated parameters simply uses the data type of the calibration parameter. When the `--deviate-group-calibration-none` command-line option is enabled the group naming follows the same rules as for the pointered methods.

Grouped Parameters

A calibration parameter group is implemented as a C structure. The type is declared within `Rte_Type.h` and the type name is:

```
Rte_CGT<loc>_<swci>_<swAddr>
```

Where:

- `<loc>` is an RTA-RTE symbol depending on the location of the calibration parameter definitions:
  - c – calibration parameters within a CalPrmComponentType.
  - u – calibration parameters within an unconnected RPort within an ApplicationComponentType.
  - s – shared calibration parameters within a SWC Type.
  - `i` – per-instance parameters within a SWC Type.
- `<swci>` is as follows:
  - For calibration parameters within a CalPrmComponentType, calibration parameters within unconnected RPorts of an ApplicationComponentType and per-instance calibration parameters declared within a SWC type, this field is the SWC instance internal name[1].
  - For shared calibration parameters declared within an SWC type, this field is the SWC type name.
- `<swAddr>` is the `swAddrMethod` name, except for calibration parameters within an unconnected RPort when it is the port name followed by the `swAddrMethod` name, separated by an underscore.

[1]The RTE configuration constants file includes support for associating component prototypes and the RTE’s internal name. See Sections 10.5.5 and **??**.

The type of each group is also described, using AUTOSAR XML, in the McSupportData file created by RTA-RTE.

Note that the type name of the data structure is based on the internal RTA-RTE SWCI identifier and a data type is declared for each instance of the SWC type because different `SwAddrMethods` may be applied to different instances using `FlatInstanceDescriptors` and hence the groups may be different for each instance.

10.5.2 Instance Name

Non-grouped Parameters

When using the calibration method "none" the instance name is taken from the flatmap instance if one is available for the calibration parameter instance. The name of the flatmap instance must be globally unique.

If no flatmap instance is available then RTA-RTE uses the name of the calibration parameter. RTA-RTE will raise an error if it detects two calibration parameters are declared with the same name. To avoid this error either use a flatmap instance to rename one ot both of the parameters or enable generation of calibration parameter groups using the appropriate command-line option.

Grouped Parameters

RTA-RTE instantiates calibration parameter groups as required.

The name of allocated/imported instance of the calibration parameter group is:

```
Rte_CG<loc>_<swci>_<swAddr>
```

Where:

- `<loc>` is an RTA-RTE symbol depending on the location of the calibration parameter definitions, as for the calibration parameter group type:
  - `c` – calibration parameters within a CalPrmComponentType.
  - `u` – calibration parameters within an unconnected RPort within an ApplicationComponentType.
  - `s` – shared calibration parameters within a SWC Type.
  - `i` – per-instance parameters within a SWC Type.
- `<swci>` is as follows:
  - For calibration parameters within a CalPrmComponentType, calibration parameters within unconnected RPorts of an ApplicationComponentType and per-instance calibration parameters declared within a SWC type, this field is the SWC instance internal name[2].

[2] The RTE configuration constants file includes support for associating component prototypes and the RTE's internal name. See Sections 10.5.5 and **??**.

    - For shared calibration parameters declared within an SWC type, this field is the SWC type name.
- `<swAddr>` is the `swAddrMethod` name, except for calibration parameters within an unconnected RPort when it is the port name followed by the `swAddrMethod` name, separated by an underscore.

10.5.3 Init-RAM Structures

When using calibration method “Init-RAM” RTA-RTE creates a single combined RAM block containing space for all the calibration parameters and a single combined ROM block with the initializer for every calibration parameter. Both blocks use the following structure type:

```
Rte_CalprmInitRAMType
```

The RAM block is instantiated as a variable of that type called:

```
Rte_CalprmInitRAM
```

and the ROM block is instantiated as a constant of that type called:

```
Rte_CalprmInitROM
```

This scheme allows a single copy operation to perform the initialization of all parameters at once.

Within the blocks the parameters are grouped together following the same scheme as for the pointered calibration methods, namely by SW-C and `swAddrMethod`. Each element of the `Rte_CalprmInitRAMType` structure is itself a structure, for a calibration parameter group. The names of the calibration parameter group structure types follow the same rules as for the pointered calibration methods, as set out in section 10.5.1. The element names are:

```
CG<loc>_<swci>_<swAddr>
```

Where:

- `<loc>` is an RTA-RTE symbol depending on the location of the calibration parameter definitions:
    - `c` – calibration parameters within a CalPrmComponentType.
    - `u` – calibration parameters within an unconnected RPort within an ApplicationComponentType.
    - `s` – shared calibration parameters within a SWC Type.
    - `i` – per-instance parameters within a SWC Type.
- `<swci>` is as follows:

  - For calibration parameters within a CalPrmComponentType, calibration parameters within unconnected RPorts of an ApplicationComponentType and per-instance calibration parameters declared within a SWC type, this field is the SWC instance internal name[3].
  - For shared calibration parameters declared within an SWC type, this field is the SWC type name.
- `<swAddr>` is the `swAddrMethod` name, except for calibration parameters within an unconnected RPort when it is the port name followed by the `swAddrMethod` name, separated by an underscore.

The elements appear in the `Rte_CalprmInitRAMType` structure sorted by name.

10.5.4 Examples

The following examples illustrate the allocation of calibration parameters to groups and the import of parameters by RTA-RTE.

Shared Calibration Parameters

Consider a SWC type `swcA` that declares shared calibration parameters `A` and `B` both of which reference `swAddrMethod` `Md`.

Since both calibration parameters reference the same `swAddrMethod` and are declared in the same SWC type they are allocated to the same calibration parameter group. The type definition within `Rte_Type.h` is therefore:

```
typedef struct {
   <type>  A;
   <type>  B;
} Rte_CGTs_swcA_Md;
```

And the instance name is:

```
Rte_CGs_swcA_Md
```

For the “Init-RAM” method the type definition is unchanged but the group is instantiated within the RAM and ROM block structures, appearing as an element of the structure as follows:

```
typedef struct {
   Rte_CGTs_swcA_Md CGs_swcA_Md;
} Rte_CalprmInitRAMType;
```

CalPrmComponent Types

Consider a CalPrmComponent type `CalData` that provides a single port pd categorized by a calibration interface that declares calibration parameters A and B both of which reference `swAddrMethod` `Md`.

[3] The RTE configuration constants file includes support for associating component prototypes and the RTE’s internal name. See Sections 10.5.5 and **??**.

Since both calibration parameters reference the same swAddrMethod and are declared in the same component type they are allocated to the same calibration parameter group. However, unlike the example above the names within the generated structure are prefixed with the port name to ensure uniqueness if the same interface categorizes more than one port. The type definition within Rte_Type.h is therefore:

```
typedef struct {
   <type>  pd_A;
   <type>  pd_B;
} Rte_CGTc_<i>_Md;
```

and the instance names is:

```
Rte_CGc_<i>_Md
```

where <i> is the RTA-RTE internal name for the instance of the calibration component type.

For the “Init-RAM” method the type definition is unchanged but the group is instantiated within the RAM and ROM block structures, appearing as an element of the structure as follows:

```
typedef struct {
   Rte_CGTc_<i>_Md CGc_<i>_Md;
} Rte_CalprmInitRAMType;
```

Where <i> is the RTA-RTE internal name for the instance of the calibration component type.

10.5.5 SWC Instance Internal Names

To ensure uniqueness when creating imported instance names for calibration parameters, RTA-RTE uses an internally allocated SWC instance name. A mapping between a user-visible name for the component prototype and the internal name is defined in Rte_Const.h.

For each instantiated software component prototype, Rte_Const.h includes a #define that maps the full path to the prototype to the internal name. For example, a component prototype C1 within composition Comp1 within package pkg will have a path of _pkg_Comp1_C1. Where nested compositions are used the path includes the name of each level of the hierarchy.

10.5.6 McSupportData

RTA-RTE writes an XML description of the calibration data to the file Rte_McSupportData.arxml. Third-party tools may use this to generate an A2L file for use in calibration and measurement. To cause calibration parameters to be reported to the McSupportData, each must have a FlatInstanceDescription referencing it.

In the absence of a relevant FlatInstanceDescription, RTA-RTE emits a warning and there will be no calibration information emitted for that parameter.

10.5.7 Special Treatment of Arrays of Curves and Maps

As an addition to AUTOSAR-specified behavior, RTA-RTE supports a shorthand way to associate input variables with axes held in an array or structure, or axes with curves or maps held in an array or structure.

When a calibration parameter is typed by a complex `ApplicationDataType` whose leaf elements have `category` `CURVE`, `MAP`, `COM_AXIS`, or `RES_AXIS`, it is not necessary to write a `SwCalPrmAxisSet` for each leaf element.

If an `InstantiationDataDefProps` references the whole complex calibration parameter, then RTA-RTE will attempt to apply its `SwCalPrmAxisSet` to each member of the complex calibration parameter. Additionally, if this is the case, the `SwCalPrmAxisSet` may reference, instead of the valid axis or input variable, an array or structure of the same shape as that containing the curve, map or axis, and RTA-RTE will walk both complex types, associating the elements pairwise.

For example, an array of ten curves having a standard axis in their `SwCalPrmAxisSet` might be referenced by an `InstantiationDataDefProps` containing a `SwCalPrmAxisSet` having a `swVariableRef` referencing port data characterized by a scalar type. In this case, that port data instance will be used as the input variable for all ten curves.

Alternatively, the `InstantiationDataDefProps.swVariableRef` might reference an array of ten scalars, in which case RTA-RTE will treat the first scalar as the input variable for the first curve, the seconds scalar as the input variable for the second curve, etc.

11 Parameters of Implementation

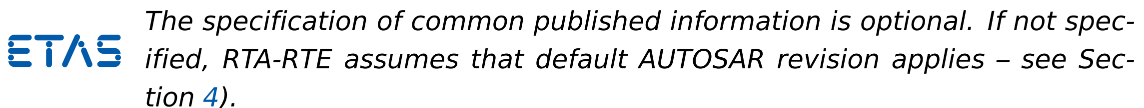
This chapter provides details of limits and constraints imposed by RTA-RTE.

11.1 AUTOSAR Common Published Information

RTA-RTE supports the following values for the Common Published Information (Section 4.19.2).

Parameter	R4.0
ArMajorVersion	4
ArMinorVersion	0
ArPatchVersion	1, 2 or 3
ModuleId	Not used
SwMajorVersion	1
SwMinorVersion	0
SwPatchVersion	0
VendorId	Not used

RTA-RTE will accept RTE generator version numbers up to and including the specified version. Higher version numbers are rejected with an error.



The specification of common published information is optional. If not specified, RTA-RTE assumes that default AUTOSAR revision applies – see Section 4).

11.2 API Legitimacy

RTE API calls, other than the RTE Lifecycle API functions, are invoked by runnable entities from within task bodies generated by RTA-RTE.

The RTE Lifecycle API functions can be invoked as follows:

RTE API	Task	ISR (Cat 2)	OS Startup hook
`Rte_Start`	✓	✓	✓[1]
`Rte_Stop`	✓	✓	✓[2]
`Rte_MainFunction`	✓	✓	✗

Category 1 ISRs must not invoke RTE API functions.

11.3 Tasks and Runnable Entities

[1] When OSEK OS is used `Rte_Start` cannot be invoked from the StartupHook since it invokes the OSEK API `SetRelAlarm`.

[2] When OSEK OS is used `Rte_Stop` cannot be invoked from the StartupHook since it invokes the OSEK API `CancelAlarm`.

Parameter	Limit
Maximum number of tasks	256 (subject to support from underlying operating system).
Maximum number of runnable entities mapped to an OS task.	No limit imposed by RTA-RTE.
Maximum number of TimingEvents.	No limit imposed by RTA-RTE. However the use of non-harmonic periods may result in large schedule tables that exceed the limits of the underlying operating system.
Maximum number of runnable entities activated during a mode switch.	No limit imposed by RTA-RTE.

11.4 Queued Communication

Parameter	Limit
Maximum number of entries in a queue (for queued communication).	65535
Maximum size of each entry in a queue (intra-ECU communication).	65535 bytes
Maximum size of each entry in a queue (inter-ECU communication).	8 bytes

11.5 Scheduling

Parameter	Limit
Minimum supported TimingEvent period	1 μs[3]
Maximum number of expiry points	10000

11.6 Modes and Mode Switches

Parameter	Limit
Maximum number of mode declarations per mode declaration group.	32
Maximum size of a mode switch queue.	255

[3]This value is the minimum supported by RTA-RTE. The practical minimum supported timing event period is dependent on the underlying OS implementation and will typically be significant larger.

11.7 Inter-ECU Communication

Parameter	Limit
Maximum length of signal for a client-server sequence counter	16 bits.

RTA-RTE always uses 16-bit types to maintain counter state since System signals do not include the length.

12 AUTOSAR Revision Support

This chapter provides details of optional behavior enabled using the `--sws` command-line option.

The `--sws` command-line option can be used to select the following AUTOSAR revision specific behavior:

Behavior	–sws	Notes
Mode definitions within application types header or BSW module interlink types header files. Type definitions and preprocessor symbol definitions for modes are wrapped in an include guard symbol.	4.0.1 or 4.0.2	Definition uses a type-cast and parenthesis.
	4.0.3	No type-cast or parenthesis are present but instead the definition uses a U suffix to indicate that it is an unsigned value.
Use of BSW module description short-name for entry point prototype definitions.	4.0.1	Memory mapping and compiler abstraction macros use short-name unmodified.
	4.0.2 and above	Memory mapping and compiler abstraction macros convert short-name to upper-case before use.

13 Contact, Support and Problem Reporting

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries	www.etas.com/en/contact.php
ETAS technical support	www.etas.com/en/hotlines.php

The RTA hotline is available to all RTA-RTE users with a valid support contract.

rta.hotline.uk@etas.com
+44 (0)1904 562624. (0900-1730 GMT/BST)

Please provide support with the following information:

- Your support contract number.
- Your AUTOSAR XML and/or OS configuration files.
- The command line that results in an error message.
- The version of the ETAS tools you are using.

Index

Symbols

--append-name-to-buffer, 23
--atomic-assign, 24
--bit-pack-type, 25
--bsw-scope-limit-defns, 27
--bsw, 26
--calibration-disable, 28
--calibration-instantiation, 29, 214
--calibration-method, 30, 193
--client-server-global-optimization, 31
--com-symbolic-sigs, 32
--com-version, 33
--contract, 34
--deviate-allow-supportsmulti-sharedmemorys, 39
--deviate-allow-unmapped-swci-config, 35
--deviate-appl-impl-compu-method, 36
--deviate-appl-impl-display-format, 37
--deviate-bsw-any-partition, 38, 49
--deviate-enum-cast, 40
--deviate-group-calibration-none, 41, 270, 271
--deviate-ignore-datatype-semantics, 42
--deviate-implicit-cat2-mdd, 43
--deviate-implicit-modify-for-loopbacks, 44
--deviate-memmap-decls=0, 262
--deviate-memmap-decls, 45
--deviate-omit-implicit-cds, 46
--deviate-physical-dimension-compatibility, 47
--deviate-prefer-no-empty-executions, 48
--deviate-split-swci-support, 49
--deviate-trace-implicit-api, 51
--deviate-unconnected-pmode-behavior, 52
--disable-warning, 53, 108
--error-as-warning, 54, 108
--error-report, 55
--exclusive-area-optimization, 56
--fast-init, 57, 247
--file, 58
--force-basic-tasks, 59
--have-64bit-int-types, 60
--help, 61
--implicit-allocation-method, 62, 263
--implicit-read-return-const, 63
--implicit-use-global-buffers, 64
--incremental-build, 65
--initial-value-rounding, 66

--ioc-header, 67
--ioc-xml-namespace, 68
--local-mcsd, 69
--makedep, 70
--mcore-spinlocks-always, 71
--mcsd-policy, 72
--measurement, 73
--memory-sections, 74
--notimestamps, 75
--operating-system, 76
--optimize, 77, 193
--os-define-osenv, 78
--os-fp, 79
--os-header, 80
--os-output-param, 81
--os-permit-extended-tasks, 82
--os-task-as-function, 83
--os-xml-namespace, 84
--output, 14, 19, 85
--period, 86, 228
--preferred-intra-core-protection-scheme, 87
--protection-threshold-copy-bytes, 88
--quiet, 89
--report, 90
--rte, 91, 113, 114
--samples, 92, 262
--strict-config-check, 93
--strict-initial-values-check, 94
--strict-unconnected-rport-check, 95
--sws, 96, 280
--task-recurrence, 97
--template-path, 98
--terminate-background-tasks, 99
--test-license, 100
--text-value-spec-policy, 101
--toolchain-significant-len, 102
--use-partition-sections, 103, 259, 260
--variability-also-bind, 104
--version, 105
--vfb-trace, 106, 193
--warn-directive, 107
--warning-as-error, 108
--xfrm-ignore-inplace, 109
--, 22

A

AFL, 203
API
BSW API
Rte_GetMirror, 153
Rte_MainFunction, 228
Rte_NvMNotifyInitBlock, 153
Rte_NvMNotifyJobFinished, 153
Rte_SetMirror, 153
Rte_Start, 239
Rte_Stop, 239
Rte_Tick_Timeouts, 241
SWC API
Rte_Call, 212
Rte_CData, 214
Rte_DRead, 234
Rte_Enter, 215
Rte_Exit, 216
Rte_Feedback, 218
Rte_IFeedback, 217
Rte_IInvalidate, 219
Rte_Invalidate, 220
Rte_IRead, 221
Rte_IrTrigger, 242
Rte_IrvIRead, 223
Rte_IrvIWrite, 224
Rte_IrvRead, 225
Rte_IrvWrite, 226
Rte_IStatus, 227
Rte_IsUpdated, 227
Rte_IWrite, 222
Rte_IWriteRef, 222
Rte_MainFunction, 16, 86
Rte_Mode, 229
Rte_NPorts, 230
Rte_Pim, 232
Rte_Port, 231
Rte_Ports, 230
Rte_Prm, 213
Rte_Read, 233
Rte_Receive, 235
Rte_Result, 237
Rte_Send, 238
Rte_Switch, 240
Rte_Trigger, 242
Rte_Write, 243
ApplicationArrayDataType, 129

ApplicationPrimitiveDataType, 128
ApplicationRecordDataType, 129
ApplicationValueSpecification, 139
Array of curve/map
  Calibration, 276
ArrayValueSpecification, 140
Assembly connector, 178
AUTOSAR, 9
  Compiler_Cfg.h, 263
  Formula Language, 203
  Package, 114
  Sub-package, 114

C

C Library, 15, 264
Calibration
  Com-spec, 122
  Configuration, 28, 143, 163, 193
  Data, 144
  Init-RAM Structure, 273
  Initial Values, 122, 164
  Instance name, 272
  Method, 15, 30
  Type name, 271
Client-server
  Application Error, 145
  Call point, 170
  Com-spec, 127
  Configuration, 144, 184, 185
  Port-Defined argument values, 166
  RTE Event, 159
Command-line
  Examples, 20
  Options, 20
Command-line option
  file, 58
Compiler Abstraction, 263
Component Prototype
  Implementation, 189
  Instantiation, 177
  Mapping, 189
Composition
  Assembly connector, 178
  Configuration, 177
  Delegation connector, 178
  Port prototype, 179

Service connector, 179
CompuMethod
of Category IDENTICAL, 135
of Category LINEAR, 135
of Category RAT_FUNC, 136
of Category TEXTTABLE, 136
ConstantReference, 141
ConstantSpecification, 140

D
DataTypeMappingSet, 137
Delegation connector, 178
Deprecated Command-line Options
-O, *see* --optimize
-aa, *see* --atomic-assign
-bpt, *see* --bit-pack-type
-bsld, *see* --bsw-scope-limit-defns
-b, *see* --bsw
-cd, *see* --calibration-disable
-ci, *see* --calibration-instantiation
-cm, *see* --calibration-method
-csgo, *see* --client-server-global-optimization
-cv, *see* --com-version
-cy, *see* --com-symbolic-sigs
-c, *see* --contract
-dv8-constr2028, *see* --deviate-allow-supportsmulti-sharedmemorys
-dv8aicm, *see* --deviate-appl-impl-compu-method
-dv8aidf, *see* --deviate-appl-impl-display-format
-dv8ausc, *see* --deviate-allow-unmapped-swci-config
-dv8bap, *see* --deviate-bsw-any-partition
-dv8ec, *see* --deviate-enum-cast
-dv8gcpn, *see* --deviate-group-calibration-none
-dv8ic2mdd, *see* --deviate-implicit-cat2-mdd
-dv8idts, *see* --deviate-ignore-datatype-semantics
-dv8imflb, *see* --deviate-implicit-modify-for-loopbacks
-dv8md, *see* --deviate-memmap-decls
-dv8omitcds, *see* --deviate-omit-implicit-cds
-dv8pdc, *see* --deviate-physical-dimension-compatibility
-dv8pnee, *see* --deviate-prefer-no-empty-executions
-dv8sss, *see* --deviate-split-swci-support
-dv8tia, *see* --deviate-trace-implicit-api
-dv8upb, *see* --deviate-unconnected-pmode-behavior
-dw, *see* --disable-warning
-eao, *see* --exclusive-area-optimization
-eaw, *see* --error-as-warning
-err, *see* --error-report

-fbt, *see* --force-basic-tasks
-fi, *see* --fast-init
-f, *see* --file
-h, *see* --help
-iam, *see* --implicit-allocation-method
-ib, *see* --incremental-build
-igb, *see* --implicit-use-global-buffers
-int64, *see* --have-64bit-int-types
-iochdr, *see* --ioc-header
-iocxmlns, *see* --ioc-xml-namespace
-irrc, *see* --implicit-read-return-const
-ivr, *see* --initial-value-rounding
-l, *see* --local-mcsd
-mc, *see* --mcsd-policy
-mem, *see* --memory-sections
-ms, *see* --measurement
-m, *see* --makedep
-nts, *see* --notimestamps
-osenv, *see* --os-define-osenv
-osfp, *see* --os-fp
-oshdr, *see* --os-header
-osparam, *see* --os-output-param
-ospet, *see* --os-permit-extended-tasks
-osxmlns, *see* --os-xml-namespace
-os, *see* --operating-system
-o, *see* --output
-picps, *see* --preferred-intra-core-protection-scheme
-ptcb, *see* --protection-threshold-copy-bytes
-p, *see* --period
-q, *see* --quiet
-rn, *see* --append-name-to-buffer
-rp, *see* --report
-r, *see* --rte
-samples, *see* --samples
-scc, *see* --strict-config-check
-siv, *see* --strict-initial-values-check
-spla, *see* --mcore-spinlocks-always
-sws, *see* --sws
-terminate-background-tasks, *see* --terminate-background-tasks
-tf, *see* --os-task-as-function
-tl, *see* --test-license
-tp, *see* --template-path
-tr, *see* --task-recurrence
-tsl, *see* --toolchain-significant-len
-ups, *see* --use-partition-sections
-ur, *see* --strict-unconnected-rport-check

- `-vab`, *see* `--variability-also-bind`
- `-vt`, *see* `--vfb-trace`
- `-v`, *see* `--version`
- `-warn`, *see* `--warn-directive`
- `-we`, *see* `--warning-as-error`
- `-xfrm-ignore-inplace`, *see* `--xfrm-ignore-inplace`
- `-`, *see* `--`, *see* `--text-value-spec-policy`

E

ECU Description, 190
ECU Type
- Configuration, 177
- Instance, 179

ECUC, 9
Error messages, 18
Exclusive areas, 172
- Configuration, 162

F

Files
- Rte.err, 55

Filter, 124
force-basic semantics, 59
forced-basic semantics, 193
Formula Language, 203

I

Implementation (SWC), 174
- Source/object code, 174

ImplementationDataType, 130
- of Category `ARRAY`, 133
- of Category `STRUCTURE`, 134
- of Category `TYPE_REFERENCE`, 131
- of Category `VALUE`, 132

ImplementationDataTypeElement, 135
Implicit
- reception, 168
- transmission, 168

IncludedDataTypeSet, 141
Indirect API, 166
Instance handle, 210
Inter-ECU
- PDU Type, 176
- System signal, 175
- System signal group, 176

Inter-runnable variables
- Configuration, 162

Read, 173
Write, 173
Interface, 142
Calibration, 143
Client-server, 144
Nv-Data, 143
Sender-receiver, 142
Internal Behavior
Calibration, 163
Exclusive area, 162
Inter-runnable variables, 162
Multiple instantiation, 174
NVRAM mapping, 163
Port option, 165
Runnable entity, 167
Internal behavior, 155
RTE Event, 156
Invocation, 11
Exit codes, 18

M
MAKW, 253
McSupportData, 275
Measurement
Configuration, 73, 146
Global enable, 15, 193
Schema, 150
memcpy, 15, 264
Memory Allocation Keyword, 253
Memory Mapping, 253
Generating Rte_MemMap.h, 260
Mode dependency, 161
Mode Manager, 170
Mode manager
Acknowledgment, 121
Com-spec, 120
Mode Switch
Asynchronous, 121, 240
Synchronous, 240
Mode user
Com-spec, 121
Modes
Configuration, 155

N
nativeDeclaration, 133
NumericalValueSpecification, 139

Nv-Data
    Configuration, 143
NVRAM, *see also* BSW API
    Callback API, 153
    Nv-Block Data Mapping, 150
    Read Access, 152
    Role Based Port Assignment, 153
    Write Access, 152
NVRAM mapping, 163

O
Optimization, 77
    Configuration, 193
    VFB tracing, 252
Os
    Alarm, 266
    Counters, 16, 265
    Events, 266
    Resources, 265
Os config, 191
Output
    Console, 17, 55
    Error messages, 18
    File, 55
Output files, 11
    Application header, 12
    Basic Software Module Description file, 13
    COM config, 13, 14
    McSupportData file, 13
    OS config, 13, 14, 16, 264
    Rte.c, 12
    Rte.err, 13
    Rte.h, 12
    Rte_Cbk.h, 12
    Rte_Cfg.h, 248
    Rte_Const.h, 12, 15, 19, 248, 265, 266
    Rte_Hook.h, 12
    Rte_Intl.h, 12
    Rte_Lib.c, 18
    Rte_lib.c, 12
    Rte_Main.h, 12
    Rte_Type.h, 12, 268, 271
    Task body, 13

P
PDU Type
    Configuration, 176

Phase
  BSW, 26
  Configuration, 192
  Contract, 34
  LocalMCSD, 69
  RTE, 91, 248
Port
  Configuration, 116
Port-Defined argument values, 166
Port-options, 165
  Indirect API, 166
  Take address, 166
Pure runnable, 167

R
RecordValueSpecification, 140
Reference
  Absolute, 112
  ECU Instance, 113
  Instance, 113
  Relative, 112
RTA-OSEK, 9
RTE, 9
  Configuration, 110, 191
  Namespace, 209
RTE API, *see* API
RTE Event
  Asynchronous Server Call Returns, 159, 245
  Data Receive Error, 157, 245
  Data Received, 157, 245
  Data Send Completed, 158, 245
  Mode dependency, 161
  Mode Switch, 160, 245
  Mode Switched Ack, 245
  Mode Switched Acknowledge, 160
  Operation Invoked, 159, 245
  Timing, 156, 246
RTE Library, 18
Rte_Activity, 266
Rte_Instance, 210
RTE_LIBC_MEMCPY, 15, 19
Rte_memcpy, 15, 264
Rte_PortHandle, 211
Rte_ScheduleTable, 265
Rte_Tick_Counter, 266
Rte_Timeout, 267

Rte_TOut_Counter, 266
Rte_UserCfg.h, 19
RTE_VFB_TRACE, 248
RteForceBasicTask, 193
Runnable, 167
    Blocking API, 171
    Call point, 170
    Data read access, 168
    Data write access, 168
    Exclusive areas, 172
    Minimum start interval, 173
    Mode switch, 170
    Re-entrant, 167
    Read variables, 173
    Receive point, 169
    Send point, 169
    Signature, 246
    Symbol, 171
    Waitpoint, 171
    Written variables, 173
Runnable Entity Mapping, 195

S
Sender-receiver
    Acknowledgment, 119
    Alive timeout, 124
    Com-spec, 119, 122
    Configuration, 142, 182
    Filter, 124
    Initial Value, 124
    Initial Values, 120
    Invalid value, 120, 139
    Invalidation, 124
    Receive point, 169
    RTE Event, 157, 158
    Send point, 169
    Timeout, 120
Services
    Reference, 195
Splitable stereotype, 207
Std_ReturnType, 211
SWC Type
    Configuration, 115
    Implementation, 174
    Instantiation, 177, 194
    Multiple instantiation, 174

Naming, 209
Port, 116
System
Configuration, 181
System signal
Configuration, 175
Reference to, 177
System signal group
Configuration, 176

T
TextTableMapping, 138
typeEmitter, 131

U
Unit, 138
User Configuration File, 19

V
VFB Trace
COM Notification, 250
Configuration, 248
OS Task Activation, 250
OS Task Dispatch, 250
OS Task Set Event, 250
OS Task Wait Event, 250
OS Task Wait Event Return, 250
RTE API Return, 249
RTE API Start, 248
Runnable Invocation, 251
Runnable Termination, 251
Signal Reception, 249
Signal Transmission, 249
VFB Tracing
Configuration, 106, 192

W
Wait point, 171

X
XML, 9
Merge, 114
Splitable Elements, 115
Vendor specific, 198