Document Title	Specification of RTE Software
Document Owner	AUTOSAR GbR
Document Responsibility	AUTOSAR GbR
Document Version	1.0.1
Document Status	Final

Document Change History

Date	Version	Changed by	Change Description
2006-05-05	1.0.0	AUTOSAR Administration	Initial release.
2006-07-18	1.0.1	AUTOSAR Administration	Second release. Additional features integrated, adapted to updated version of meta-model.

Disclaimer

Advice to users of AUTOSAR Specification Documents:

Table of Contents

Bibliography

[1] Glossary
AUTOSAR_Glossary.pdf

[2] Methodology
AUTOSAR_Methodology.pdf

[3] Requirements on Communication
AUTOSAR_SRS_COM.pdf

[4] Requirements on ECU Configuration
AUTOSAR_RS_ECU_Configuration.pdf

[5] Requirements on Operating System
AUTOSAR_SRS_OS.pdf

[6] Specification of Communication
AUTOSAR_SWS_COM.pdf

[7] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[8] Specification of ECU State Manager
AUTOSAR_SWS_ECU_StateManager.pdf

[9] Specification of Interoperability of Authoring Tools
AUTOSAR_InteroperabilityAuthoringTools.pdf

[10] Specification of I/O Hardware Abstraction
AUTOSAR_SWS_IO_HWAbstraction.pdf

[11] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

[12] Specification of Operating System
AUTOSAR_SWS_OS.pdf

[13] Specification of Standard Types
AUTOSAR_SWS_StandardTypes.pdf

[14] Specification of the Virtual Functional Bus
AUTOSAR_VirtualFunctionBus.pdf

[15] Specification of System Template
AUTOSAR_SystemTemplate.pdf

[16] DTD File
AUTOSAR_DTD_File.dtd

[17] Template Modeling Guide
AUTOSAR_TemplateModelingGuide.pdf

[18] Software Component Template
AUTOSAR_SoftwareComponentTemplate.pdf

Note on XML examples

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

1 Introduction

This document contains the software specification of the AUTOSAR Run-Time Environment (*RTE*). Basically, the RTE together with the OS, AUTOSAR COM and other Basic Software Modules is the implementation of the Virtual Functional Bus concepts (*VFB*, [14]). The RTE implements the AUTOSAR Virtual Functional Bus interfaces and thereby realizes the communication between AUTOSAR software-components.

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

1.1 Scope

This document is intended to be the main reference for developers of an RTE generator tool or of a concrete RTE implementation respectively. Chapters 4, 5, and Appendix B are probably of most interest for this group of readers. The document is also the reference for developers of AUTOSAR software-components and basic software modules that interact with the RTE, since it specifies the application programming interface of the RTE and therefore the mechanisms for accessing the RTE functionality. Especially Chapters 2, 3, 5 are important for these developers. Furthermore, this specification should be read by the AUTOSAR working groups that are closely related to the RTE (see Section 1.2 below), since it describes the interfaces of the RTE to these modules as well as the behavior / functionality the RTE expects from them. The most important part for these readers would be Chapters 4 and 5, as well as Appendix B and C.

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

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

The appendix of this document consists of five parts: Appendix A lists the restrictions to the AUTOSAR metamodel that this version of the RTE specification relies on. Appendix B describes the input that is needed for the RTE generation process and where this input is assumed to come from. Appendix C explicitly lists all external requirements, i.e. all requirements that are not about the RTE itself but specify the assumptions on the environment and the input of an RTE generator. In Appendix D some HIS MISRA rules are listed that are likely to be violated by RTE code, and the rationale why these violations may occur. Finally, Appendix E lists the COM API and COM Callback functions that are used by the RTE

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

1.2 Dependency to other AUTOSAR specifications

The main documents that served as input for the specification of the RTE are the specification of the Virtual Functional Bus [14] and the specification of the Software Component Template [18]. Also of primary importance are the specifications of those Basic Software modules that closely interact with the RTE (or vice versa). These are especially the communication module [6] and the operating system [12]. The main input of an RTE generator is described (among others) in the ECU Configuration Description. Therefore, the corresponding specification [4] is also important for the RTE specification. Furthermore, as the process of RTE generation is an important part of the overall AUTOSAR Methodology, the corresponding document [2] is also considered.

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

- Specification of the Virtual Functional Bus [14]
- Specification of the Software Component Template [18]
- Specification of AUTOSAR COM [6]
- Specification of AUTOSAR OS [12]
- Specification of ECU State Manager and Communication Manager [8]
- Specification of ECU-Configuration Description / Generation [4]
- Specification of System Description / Generation [15]
- AUTOSAR Methodology [2]
- Documents relevant for the AUTOSAR Metamodel [17, 16]

1.3 Acronyms and Abbreviations

All abbreviations used throughout this document – except the ones listed here – can be found in the official AUTOSAR glossary [1].

1.4 Document Conventions

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

Requirements in the SWS are marked with **[rte_sws_<n>]** as the first text in a paragraph. The scope of the requirement is the entire paragraph.

Requirements on the input of the RTE specified in terms of the meta model are marked with **[rte_sws_in_<n>]** accordingly.

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

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

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

1.5 Requirements Traceability

Requirement	Satisfied by
[BSW00300] Module naming convention	rte_sws_1171 rte_sws_1157 rte_sws_1158 rte_sws_1003 rte_sws_1161 rte_sws_1169
[BSW00304] AUTOSAR integer data types	rte_sws_1175 rte_sws_1215 rte_sws_1176 rte_sws_1212 rte_sws_1177 rte_sws_1178 rte_sws_1179 rte_sws_1180 rte_sws_1181 rte_sws_1182 rte_sws_1183 rte_sws_1184 rte_sws_1185
[BSW00305] Self-defined data types naming convention	rte_sws_3713 rte_sws_3714 rte_sws_3733 rte_sws_2301 rte_sws_3731 rte_sws_1055 rte_sws_1150
[BSW00307] Global variables naming convention	rte_sws_1171 rte_sws_3712
[BSW00308] Definition of global data	not testable
[BSW00310] API naming convention	rte_sws_1071 rte_sws_1072 rte_sws_2631 rte_sws_1206 rte_sws_1083 rte_sws_1091 rte_sws_1092 rte_sws_1102 rte_sws_1111 rte_sws_1118 rte_sws_1252 rte_sws_3741 rte_sws_3744 rte_sws_3800 rte_sws_3550 rte_sws_3553 rte_sws_3560 rte_sws_3565 rte_sws_1120 rte_sws_1123 rte_sws_2569
[BSW00312] Shared code shall be reentrant	rte_sws_1172 rte_sws_3590 rte_sws_3749
[BSW00326] Transition from ISRs to OS tasks	rte_sws_3600 rte_sws_3530 rte_sws_3531 rte_sws_3532
[BSW00327] Error values naming convention	rte_sws_1058 rte_sws_1060 rte_sws_1064 rte_sws_1317 rte_sws_1061 rte_sws_1065 rte_sws_2571

[BSW00330] Usage of macros / inline functions instead of functions	rte_sws_1274
[BSW007] HIS MISRA C	rte_sws_1168 rte_sws_3715
[RTE00003] Tracing of sender-receiver communication	rte_sws_1357 rte_sws_1238 rte_sws_1240 rte_sws_1241 rte_sws_1242
[RTE00004] Tracing of client-server communication	rte_sws_1357 rte_sws_1238 rte_sws_1240 rte_sws_1241 rte_sws_1242
[RTE00005] Support for 'production' and 'trace' build	rte_sws_1320 rte_sws_1322 rte_sws_1323 rte_sws_1327 rte_sws_1328
[RTE00008] VFB tracing configuration	rte_sws_1320 rte_sws_1236 rte_sws_1321 rte_sws_1322 rte_sws_1323 rte_sws_1324 rte_sws_1325 rte_sws_1235
[RTE00011] Support for multiple AUTOSAR software-component instances	rte_sws_2000 rte_sws_2001 rte_sws_2018 rte_sws_2008 rte_sws_2009 rte_sws_2002 rte_sws_2017 rte_sws_3711 rte_sws_1012 rte_sws_1013 rte_sws_3806 rte_sws_3793 rte_sws_3713 rte_sws_3718 rte_sws_3719 rte_sws_1349 rte_sws_3720 rte_sws_3721 rte_sws_3716 rte_sws_3717 rte_sws_3722 rte_sws_1148 rte_sws_1016
[RTE00012] Multiply instantiated AUTOSAR software-components delivered as binary code shall share code	rte_sws_3015 rte_sws_2017 rte_sws_1007
[RTE00013] Static memory sections	rte_sws_3790 rte_sws_2303 rte_sws_2304 rte_sws_3789 rte_sws_3782 rte_sws_2305 rte_sws_2301 rte_sws_2302
[RTE00017] Rejection of inconsistent component implementations	rte_sws_3755 rte_sws_4504 rte_sws_3764 rte_sws_1004 rte_sws_1276
[RTE00018] Rejection of invalid configurations	rte_sws_5508 rte_sws_2254 rte_sws_2102 rte_sws_2310 rte_sws_2051 rte_sws_2009 rte_sws_2204 rte_sws_1313
[RTE00019] RTE is the communication infrastructure	rte_sws_6000 rte_sws_6011 rte_sws_5500 rte_sws_6025 rte_sws_4527 rte_sws_6023 rte_sws_4526 rte_sws_6024 rte_sws_3760 rte_sws_3761 rte_sws_3762 rte_sws_4515 rte_sws_4516 rte_sws_4520 rte_sws_4522 rte_sws_2527 rte_sws_2528 rte_sws_3769 rte_sws_1048 rte_sws_1264 rte_sws_3795 rte_sws_3796 rte_sws_1231 rte_sws_3007 rte_sws_3008 rte_sws_3000 rte_sws_3001 rte_sws_3002 rte_sws_3775 rte_sws_2612 rte_sws_2610 rte_sws_3004 rte_sws_3005 rte_sws_3776 rte_sws_2611
[RTE00020] Access to OS	rte_sws_4014 rte_sws_2250
[RTE00021] Per-ECU RTE customization	rte_sws_5000 rte_sws_1316
[RTE00022] Interaction with call-backs	rte_sws_1165

[RTE00024] Source-code AUTOSAR software components	rte_sws_1195 rte_sws_1315 rte_sws_1000
[RTE00025] Static communication	rte_sws_6026
[RTE00027] VFB to RTE mapping shall be semantic preserving	rte_sws_2200 rte_sws_2201 rte_sws_1274
[RTE00028] 1:n Sender-receiver communication	rte_sws_6023 rte_sws_4526 rte_sws_6024 rte_sws_1071 rte_sws_1072 rte_sws_2631 rte_sws_1077 rte_sws_1081 rte_sws_2633 rte_sws_2635 rte_sws_1082 rte_sws_1091 rte_sws_1092 rte_sws_1135
[RTE00029] n:1 Client-server communication	rte_sws_6019 rte_sws_4519 rte_sws_4517 rte_sws_3763 rte_sws_3770 rte_sws_3767 rte_sws_3768 rte_sws_2579 rte_sws_3769 rte_sws_1102 rte_sws_1109 rte_sws_1133 rte_sws_1359 rte_sws_1166
[RTE00031] Multiple runnable entities	rte_sws_2202 rte_sws_1015 rte_sws_1126 rte_sws_1132 rte_sws_1016 rte_sws_1130 rte_sws_3749
[RTE00032] Data consistency mechanisms	rte_sws_3514 rte_sws_3500 rte_sws_3502 rte_sws_3503 rte_sws_3504 rte_sws_3507 rte_sws_3516 rte_sws_3517 rte_sws_3519 rte_sws_3739 rte_sws_3740 rte_sws_1122
[RTE00033] Serialization of server runnables	rte_sws_4515 rte_sws_4518 rte_sws_4522 rte_sws_2527 rte_sws_2528 rte_sws_2529 rte_sws_2530
[RTE00044] Production build	rte_sws_1323 rte_sws_1327 rte_sws_1235
[RTE00045] Standardized VFB tracing interface	rte_sws_1319 rte_sws_1250 rte_sws_1251 rte_sws_1321 rte_sws_1326 rte_sws_1238 rte_sws_1239 rte_sws_1240 rte_sws_1241 rte_sws_1242 rte_sws_1243 rte_sws_1244 rte_sws_1245 rte_sws_1246 rte_sws_1247 rte_sws_1248 rte_sws_1249
[RTE00046] Support for ’runnable runs inside’ exclusive areas	rte_sws_3500 rte_sws_3515 rte_sws_1120 rte_sws_1122 rte_sws_1123
[RTE00048] RTE Generator input	rte_sws_5001
[RTE00049] Construction of task bodies	rte_sws_2251 rte_sws_2254 rte_sws_2204

[RTE00051] RTE API mapping	rte_sws_3014 rte_sws_3706 rte_sws_3707 rte_sws_1143 rte_sws_1348 rte_sws_1155 rte_sws_1156 rte_sws_1153 rte_sws_1146 rte_sws_1159 rte_sws_1009 rte_sws_1276 rte_sws_1266 rte_sws_1197 rte_sws_1335 rte_sws_3718 rte_sws_3719 rte_sws_1349 rte_sws_3720 rte_sws_3721 rte_sws_3716 rte_sws_3717 rte_sws_3723 rte_sws_3733 rte_sws_2608 rte_sws_2588 rte_sws_1363 rte_sws_1364 rte_sws_2607 rte_sws_1365 rte_sws_1366 rte_sws_3734 rte_sws_2607 rte_sws_2589 rte_sws_1367 rte_sws_2301 rte_sws_2302 rte_sws_3737 rte_sws_3738 rte_sws_3739 rte_sws_3740 rte_sws_2616 rte_sws_2617 rte_sws_3731 rte_sws_3732 rte_sws_3730 rte_sws_2620 rte_sws_2621 rte_sws_1055 rte_sws_3726 rte_sws_2618 rte_sws_1343 rte_sws_1342 rte_sws_1053 rte_sws_3725 rte_sws_3752 rte_sws_2623 rte_sws_3791 rte_sws_1269 rte_sws_1148 rte_sws_2619 rte_sws_2613 rte_sws_2614 rte_sws_2615 rte_sws_1354 rte_sws_1355 rte_sws_1280 rte_sws_1281 rte_sws_2632 rte_sws_1282 rte_sws_1283 rte_sws_1284 rte_sws_1285 rte_sws_1286 rte_sws_1287 rte_sws_1289 rte_sws_1291 rte_sws_1292 rte_sws_1313 rte_sws_1288 rte_sws_1290 rte_sws_1293 rte_sws_1294 rte_sws_1295 rte_sws_1296 rte_sws_1297 rte_sws_1298 rte_sws_1312 rte_sws_1299 rte_sws_1119 rte_sws_1300 rte_sws_1254 rte_sws_1255 rte_sws_1301 rte_sws_1268 rte_sws_3743 rte_sws_1302 rte_sws_3746 rte_sws_3747 rte_sws_3801 rte_sws_1303 rte_sws_3552 rte_sws_1304 rte_sws_3557 rte_sws_3559 rte_sws_3555 rte_sws_1305 rte_sws_3562 rte_sws_3563 rte_sws_3564 rte_sws_1306 rte_sws_3567 rte_sws_3568 rte_sws_1307 rte_sws_1123 rte_sws_1308 rte_sws_1132 rte_sws_1309 rte_sws_1310
[RTE00052] Initialization and finalization of components	rte_sws_2503 rte_sws_2562

[RTE00053] AUTOSAR data types	rte_sws_1160 rte_sws_2648 rte_sws_1163 rte_sws_1175 rte_sws_1215 rte_sws_1176 rte_sws_1212 rte_sws_1177 rte_sws_1178 rte_sws_1179 rte_sws_1180 rte_sws_1181 rte_sws_1182 rte_sws_1183 rte_sws_1184 rte_sws_1185 rte_sws_1186 rte_sws_1187 rte_sws_1188 rte_sws_1265 rte_sws_1214 rte_sws_1189 rte_sws_1190 rte_sws_1191 rte_sws_1192 rte_sws_1161 rte_sws_1162 rte_sws_3559 rte_sws_3564
[RTE00055] Use of global namespace	rte_sws_1171
[RTE00056] Pre-defined primitive data types cannot be redefined	rte_sws_1263
[RTE00059] RTE API passes 'in' primitive data types by value	rte_sws_1017
[RTE00060] RTE API shall pass 'in' complex data types by reference	rte_sws_1018
[RTE00061] 'in/out' and 'out' parameters	rte_sws_1019 rte_sws_1020
[RTE00062] Local access to basic software components	rte_sws_2102 rte_sws_2310 rte_sws_2051
[RTE00064] AUTOSAR Methodology	rte_sws_5002
[RTE00065] Deterministic generation	rte_sws_2514
[RTE00068] Signal initial values	rte_sws_2517
[RTE00069] Communication timeouts	rte_sws_6002 rte_sws_6013 rte_sws_3754 rte_sws_3758 rte_sws_3759 rte_sws_3763 rte_sws_3770 rte_sws_3773 rte_sws_3771 rte_sws_3772 rte_sws_3767 rte_sws_3768 rte_sws_1064 rte_sws_1085 rte_sws_1095 rte_sws_1107 rte_sws_1209 rte_sws_1114
[RTE00070] Invocation order of runnables	rte_sws_2207
[RTE00072] Activation of runnable entities	rte_sws_3526 rte_sws_3527 rte_sws_3530 rte_sws_3531 rte_sws_3532 rte_sws_3523 rte_sws_3520 rte_sws_3524 rte_sws_2203 rte_sws_1131 rte_sws_2512 rte_sws_1133 rte_sws_1359 rte_sws_1166 rte_sws_1135 rte_sws_1137
[RTE00073] Data items are atomic	rte_sws_4527
[RTE00075] API for accessing static memory sections	rte_sws_1118 rte_sws_1119
[RTE00077] Instantiation of static memory sections	rte_sws_3790 rte_sws_2303 rte_sws_2304 rte_sws_3789 rte_sws_3782 rte_sws_2305

[RTE00078] Support for INVALIDATE attribute	rte_sws_5024 rte_sws_2607 rte_sws_2607 rte_sws_2589 rte_sws_2590 rte_sws_2609 rte_sws_2594 rte_sws_2525 rte_sws_1206 rte_sws_1282 rte_sws_1231 rte_sws_2626 rte_sws_3800 rte_sws_3801 rte_sws_3802 rte_sws_3778 rte_sws_2599 rte_sws_2600 rte_sws_2603 rte_sws_2629
[RTE00079] Single asynchronous client-server interaction	rte_sws_3765 rte_sws_3766 rte_sws_3771 rte_sws_3772 rte_sws_1109 rte_sws_1133 rte_sws_1359 rte_sws_1166
[RTE00080] Multiple requests of servers	rte_sws_4516 rte_sws_4520
[RTE00082] Standardized communication protocol	rte_sws_3575 rte_sws_6025 rte_sws_6028 rte_sws_6029 rte_sws_6027 rte_sws_2579
[RTE00083] Optimization for source-code components	rte_sws_1152 rte_sws_1274
[RTE00084] Support infrastructural errors	rte_sws_2593 rte_sws_1318
[RTE00087] Application Header File	rte_sws_1000 rte_sws_3786 rte_sws_1004 rte_sws_1001 rte_sws_1006 rte_sws_1263 rte_sws_1009 rte_sws_1132
[RTE00089] Independent access to interface elements	rte_sws_6008
[RTE00091] Inter-ECU Marshalling	rte_sws_4505 rte_sws_4506 rte_sws_4507 rte_sws_4508 rte_sws_2557 rte_sws_6025 rte_sws_4527
[RTE00092] Implementation of VFB model waitpoints	rte_sws_1358 rte_sws_3010 rte_sws_3018
[RTE00093] Externally triggered RTEEvents	rte_sws_2500 rte_sws_2562 rte_sws_2512
[RTE00094] Communication and Resource Errors	rte_sws_2524 rte_sws_2525 rte_sws_1034 rte_sws_1318 rte_sws_2571 rte_sws_1073 rte_sws_1074 rte_sws_1207 rte_sws_1339 rte_sws_1084 rte_sws_1085 rte_sws_3774 rte_sws_1086 rte_sws_1093 rte_sws_2598 rte_sws_1094 rte_sws_1095 rte_sws_2572 rte_sws_1103 rte_sws_1104 rte_sws_1105 rte_sws_1106 rte_sws_1107 rte_sws_1209 rte_sws_1112 rte_sws_1113 rte_sws_1114 rte_sws_2578 rte_sws_3803 rte_sws_2602 rte_sws_1356 rte_sws_1261 rte_sws_1262 rte_sws_1259 rte_sws_1260
[RTE00098] Explicit Transmission	rte_sws_3011 rte_sws_6011 rte_sws_6016 rte_sws_1071
[RTE00099] Decoupling of interrupts	rte_sws_3600 rte_sws_3530 rte_sws_3531 rte_sws_3532
[RTE00100] Compiler independent API	rte_sws_1314

[RTE00107] Support for INFORMATION_TYPE attribute	rte_sws_6010 rte_sws_4500 rte_sws_2516 rte_sws_2518 rte_sws_2520 rte_sws_2521 rte_sws_2522 rte_sws_2523 rte_sws_2524 rte_sws_2525 rte_sws_2571 rte_sws_2572 rte_sws_1135 rte_sws_1137
[RTE00108] Support for INIT_VALUE attribute	rte_sws_4525 rte_sws_6009 rte_sws_4501 rte_sws_4502 rte_sws_2517 rte_sws_1268
[RTE00109] Support for RECEIVE_MODE attribute	rte_sws_3018 rte_sws_6002 rte_sws_6012 rte_sws_2519
[RTE00110] Support for BUFFERING attribute	rte_sws_2515 rte_sws_2522 rte_sws_2523 rte_sws_2524 rte_sws_2525 rte_sws_2526 rte_sws_2527 rte_sws_2529 rte_sws_2530 rte_sws_2571 rte_sws_2572
[RTE00111] Support for CLIENT_MODE attribute	rte_sws_1293 rte_sws_1294 rte_sws_1295
[RTE00115] API for data consistency mechanism	rte_sws_1120 rte_sws_1307 rte_sws_1122 rte_sws_1308
[RTE00116] RTE Initialization, finalization and resumption	rte_sws_2513 rte_sws_2535 rte_sws_2536 rte_sws_2538 rte_sws_2544 rte_sws_2626 rte_sws_1139 rte_sws_2569 rte_sws_2582 rte_sws_2585 rte_sws_2570 rte_sws_2583 rte_sws_2584
[RTE00121] Support for FILTER attribute	rte_sws_5503 rte_sws_5500 rte_sws_5501
[RTE00122] Support for SUCCESS attribute	rte_sws_5504 rte_sws_3754 rte_sws_3756 rte_sws_3757 rte_sws_3758 rte_sws_1080 rte_sws_1083 rte_sws_1283 rte_sws_1284 rte_sws_1285 rte_sws_1286 rte_sws_1287 rte_sws_1084 rte_sws_1086 rte_sws_1137 rte_sws_3002 rte_sws_3775 rte_sws_2612 rte_sws_2610 rte_sws_3005 rte_sws_3776 rte_sws_2611
[RTE00123] Forwarding of application level errors	rte_sws_2593 rte_sws_2576 rte_sws_1103 rte_sws_2577 rte_sws_2578
[RTE00124] APIs for application level server errors	rte_sws_2573 rte_sws_2575 rte_sws_1103 rte_sws_1130
[RTE00125] Interaction of 1:n communication with the SUCCESS attribute	rte_sws_5506
[RTE00126] C support	rte_sws_1167 rte_sws_1005 rte_sws_3709 rte_sws_3710 rte_sws_1162 rte_sws_1169 rte_sws_3724
[RTE00128] Implicit Reception	rte_sws_3012 rte_sws_3013 rte_sws_6000 rte_sws_6001 rte_sws_6004 rte_sws_6011 rte_sws_1005 rte_sws_3709 rte_sws_3710 rte_sws_3741 rte_sws_1268

[RTE00129] Implicit Transmission	rte_sws_3011 rte_sws_6011 rte_sws_3570 rte_sws_3571 rte_sws_3572 rte_sws_3573 rte_sws_3744 rte_sws_3746
[RTE00131] n:1 Sender-receiver communication	rte_sws_3760 rte_sws_3761 rte_sws_3762 rte_sws_1071 rte_sws_1072 rte_sws_2631 rte_sws_1077 rte_sws_1081 rte_sws_2633 rte_sws_2635 rte_sws_1091 rte_sws_1092 rte_sws_1135
[RTE00133] No parallel execution of runnable instance	rte_sws_3523 rte_sws_3590
[RTE00134] Runnable entity categories supported by the RTE	rte_sws_3016 rte_sws_6003 rte_sws_6005 rte_sws_6006 rte_sws_6007 rte_sws_3574 rte_sws_6017
[RTE00136] AUTOSAR interface of basic software	rte_sws_2100 rte_sws_3807 rte_sws_3808 rte_sws_2050
[RTE00137] API for mismatched ports	rte_sws_1368 rte_sws_1369
[RTE00138] C++ support	rte_sws_1011 rte_sws_1370 rte_sws_1162 rte_sws_1169 rte_sws_3724
[RTE00139] API for unconnected ports	rte_sws_1329 rte_sws_1330 rte_sws_1331 rte_sws_1336 rte_sws_1344 rte_sws_1345 rte_sws_1332 rte_sws_3783 rte_sws_1346 rte_sws_1347 rte_sws_3784 rte_sws_3785 rte_sws_2638 rte_sws_2639 rte_sws_2640 rte_sws_2641 rte_sws_2642 rte_sws_1333 rte_sws_1337 rte_sws_1334 rte_sws_1338
[RTE00140] Binary-code AUTOSAR software components	rte_sws_1195 rte_sws_1315 rte_sws_1000
[RTE00141] Explicit Reception	rte_sws_3013 rte_sws_6011 rte_sws_1072 rte_sws_1091 rte_sws_1092
[RTE00142] InterRunnableVariables	rte_sws_3518 rte_sws_3588 rte_sws_3591 rte_sws_3589 rte_sws_3516 rte_sws_3517 rte_sws_3582 rte_sws_3583 rte_sws_3584 rte_sws_3519 rte_sws_3580 rte_sws_2636 rte_sws_1350 rte_sws_1351 rte_sws_3550 rte_sws_1303 rte_sws_3581 rte_sws_3552 rte_sws_3556 rte_sws_3558 rte_sws_3553 rte_sws_1304 rte_sws_3557 rte_sws_3559 rte_sws_3555 rte_sws_3560 rte_sws_1305 rte_sws_3562 rte_sws_3563 rte_sws_3564 rte_sws_3565 rte_sws_1306 rte_sws_3567 rte_sws_3568 rte_sws_3569
[RTE00143] Mode switches	rte_sws_2500 rte_sws_2503 rte_sws_2504 rte_sws_2544 rte_sws_2626 rte_sws_2562 rte_sws_2563 rte_sws_2564 rte_sws_2587 rte_sws_2630 rte_sws_2546 rte_sws_2631 rte_sws_2634 rte_sws_2512

[RTE00144] Mode switch notification via AUTOSAR interfaces	rte_sws_2544 rte_sws_2626 rte_sws_2549 rte_sws_2586 rte_sws_2508 rte_sws_2566 rte_sws_2624 rte_sws_2567 rte_sws_2546 rte_sws_2627 rte_sws_2568 rte_sws_2628
[RTE00145] Compatibility mode	rte_sws_1151 rte_sws_1216 rte_sws_1234 rte_sws_1257 rte_sws_3794 rte_sws_1279 rte_sws_1326 rte_sws_1277
[RTE00146] Vendor mode	rte_sws_1234
[RTE00147] Support for communication infrastructure time-out notification	rte_sws_5020 rte_sws_5021 rte_sws_3759 rte_sws_5022 rte_sws_2607 rte_sws_2607 rte_sws_2589 rte_sws_2590 rte_sws_2609 rte_sws_2627 rte_sws_2599 rte_sws_2600 rte_sws_2604 rte_sws_2629
[RTE00148] Support ’Specification of Memory Mapping’	rte_sws_3788
[RTE00149] Support ’Specification of Compiler Abstraction’	rte_sws_3787 rte_sws_1164
[RTE00150] Support ’Specification of Platform Types’	rte_sws_1164
[RTE00151] Support RTE section of the ’General Requirements on Basic Software Modules’	see [BSW...] entries in this table
[RTE00152] Support for port-defined argument values	rte_sws_1360 rte_sws_3780 rte_sws_3779 rte_sws_3781

2 RTE Overview

2.1 The RTE in the Context of AUTOSAR

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

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

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

2.2 AUTOSAR Concepts

This section introduces fundamental AUTOSAR concepts and how they are understood within the context of the RTE.

2.2.1 AUTOSAR Software-components

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

An AUTOSAR software-component is defined by a *type* definition that defines the component’s interfaces. A component type is instantiated when the component is deployed

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

to an ECU. A component type can be instantiated more than once on the same ECU in which case the component type is said to be “multiply instantiated”. The RTE supports per-instance memory sections that enable each component instance to have private states.

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

Details of AUTOSAR software-components in relation to the RTE are presented in Section 4.1.2.

2.2.2 Basic Software Modules

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

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

2.2.3 Communication

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

2.2.3.1 Communication Models

The AUTOSAR VFB Specification [14] defines two communication models within the RTE core services; sender-receiver (signal passing) and client-server (function invocation). Each communication model can be applied to either intra-ECU software-component distribution (which includes both intra-task and inter-task distribution) and inter-ECU software-component distribution. Intra-task communication occurs between runnable entities that are mapped to the same OS task whereas inter-task communication occurs between runnable entities mapped to different tasks and can therefore involve a context switch and possibly cross memory protection boundaries. In contrast, inter-ECU communication occurs between runnable entities in components that have been mapped to different ECUs and so is inherently concurrent and involves potentially unreliable communication.

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

2.2.3.2 Communication Modes

The RTE supports two modes for sender-receiver communication:

- **Explicit** — A component uses explicit RTE API calls to send and receive data elements [RTE00098].
- **Implicit** — The RTE automatically reads a specified set of data elements before a runnable is invoked and automatically writes (a different) set of data elements after the runnable entity has terminated [RTE00128] [RTE00129]. The term “implicit” is used here since the runnable does not actively initiate the reception or transmission of data.

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

2.2.3.3 Static Communication

[rte_sws_6026] The RTE shall support static communication only.

Static communication includes only those communication connections where the source(s) and destination(s) of all communication is known at the point the RTE is generated. [RTE00025]. Dynamic reconfiguration of communication is not supported due to the run-time and code overhead which would therefore limit the range of devices for which the RTE is suitable.

2.2.3.4 Multiplicity

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

- When using sender-receiver communication, the RTE supports both “1:n” (single sender with multiple receivers) [RTE00028] and “n:1” (multiple senders and a single receiver) [RTE00131] communication.

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

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

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

2.2.4 Concurrency

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

The AUTOSAR VFB specification [14] defines a runnable entity as a “sequence of instructions that can be started by the Run-Time Environment”. A component provides one[2] or more runnable entities [RTE00031] and each runnable entity has exactly one entry point. An entry point defines the *symbol* within the software-component’s code that provides the implementation of a runnable entity.

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

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

The RTE supports runnable entities in any component that has an AUTOSAR interface - this includes AUTOSAR software-components and basic software modules.[3]

Runnable entities are divided into multiple categories with each catgory supporting different facilities. The categories supported by the RTE are described in Section 4.2.2.2.

2.3 The RTE Generator

The RTE generator is one of a set of tools[4] that create the realization of the AUTOSAR virtual function bus for an ECU based on information in the *ECU Configuration Description*. The RTE Generator is responsible for creating the AUTOSAR software-component API functions that link AUTOSAR software-components to the OS and manage communication between AUTOSAR software-components and between AUTOSAR software-components and basic software modules.

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

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

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

The RTE generation process consists of two distinct phases:

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

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

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

2.4 Design Decisions

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

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

placed on the use of the meta-model. The restrictions are described in Appendix A.

3 RTE Generation Process

This chapter describes the methodology of the RTE generation. For a detailed description of the overall AUTOSAR methodology refer to methodology document [2].

[rte_sws_5002] The RTE Generation tools shall support the AUTOSAR Methodology [2].

[rte_sws_2514] The RTE generator shall produce the same RTE API and RTE code when the input information is the same.

The RTE-Generator gets involved in the AUTOSAR Methodology twice. In the following section the two applications of the RTE-Generator are described.

In Figure 3.1 the overall AUTOSAR Methodology is outlined with respect to the RTE.

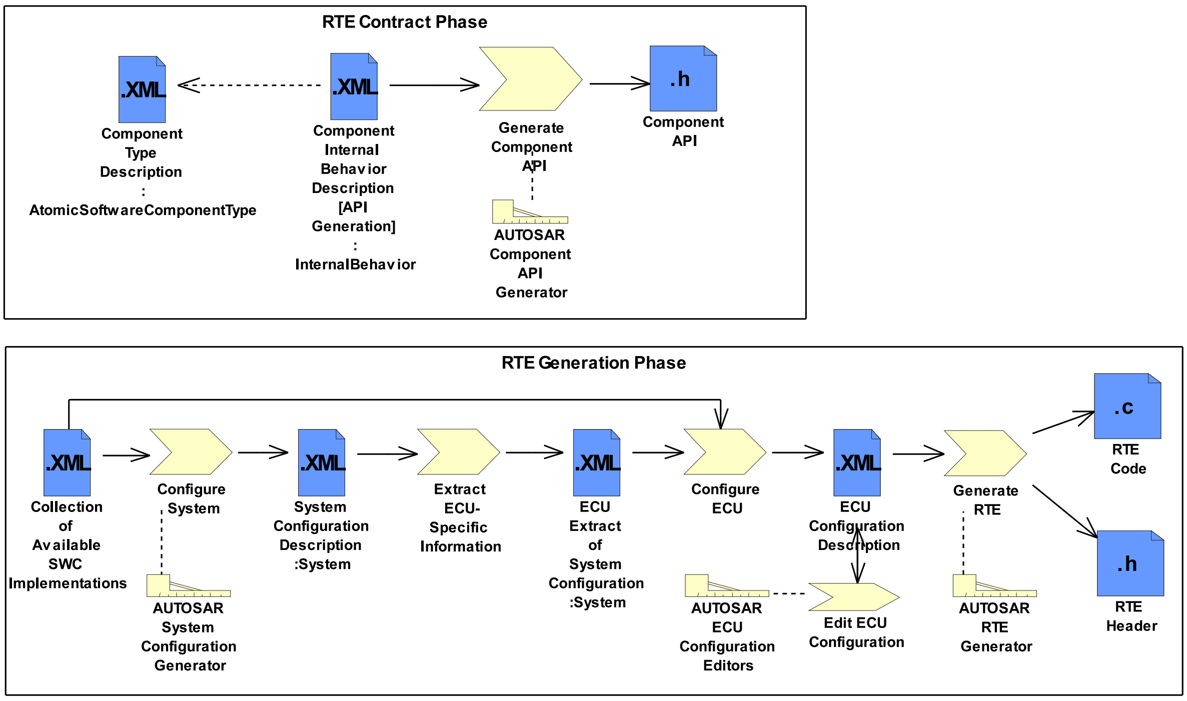

RTE Contract Phase
.XML
Component Type Description : AtomicSoftwareComponentType
.XML
Component Internal Behavior Description [API Generation] : InternalBehavior
Generate Component API
AUTOSAR Component API Generator
.h
Component API
RTE Generation Phase
.XML
Collection of Available SWC Implementations
Configure System
AUTOSAR System Configuration Generator
.XML
System Configuration Description :System
Extract ECU-Specific Information
.XML
ECU Extract of System Configuration :System
Configure ECU
AUTOSAR ECU Configuration Editors
.XML
ECU Configuration Description
Edit ECU Configuration
Generate RTE
AUTOSAR RTE Generator
.c
RTE Code
.h
RTE Header

Figure 3.1: System Build Methodology

For the development of AUTOSAR Software Components it is essential that the 'Component API Generator Tool' [1] produces the 'Component API' file in the so called 'RTE Contract Phase' (see section 3.1).

The whole vehicle functionality is described with means of Composite SW-Components and Atomic SW-Components. In the Composite SW-Component descriptions the connections between the SW-Component's ports are also defined. Such a collection of SW-Components connected to each other, without the mapping on actual ECUs, is called the VFB view.

[1]The 'Component API Generator Tool' might be a separate tool or the RTE-Generator might be operated in a special mode to achieve the same functionality. This specification does not require how the tool is implemented.

During the ’Configure System’ step the ’System Configuration Generator’ gets the information about the needed SW-Components, the available ECUs and the System Constraints. Now the Atomic SW-Components are mapped on the available ECUs.

Since in the VFB view the communication relationships between the Atomic SW-Components have been described and the mapping of each Atomic SW-Component to a specific ECU has been fixed, the communication matrix can be generated. In the SW-Component descriptions the signals that are exchanged through ports are defined in an abstract way. Now the ’System Configuration Generator’ needs to define system signals (including the actual signal length and the frames in which they will be transmitted) to be able to transmit the application signals over some network. COM signals that correspond to the system signals will be later used by the ’RTE Generator’ to actually transmit the application signals.

In the next step the ’System Configuration Description’ is split into descriptions for each individual ECU. The extract only contains information necessary to configure each ECU individually and it is fed into the ECU Configuration for each ECU.

[rte_sws_5000] The RTE is configured and generated for each ECU individually.

The ’ECU Configuration Editors’ (see also Section 3.2) are working iteratively on the ’ECU Configuration Description’ until all configuration issues are resolved. There will be the need for several configuration editors, each specialized on a specific part of ECU Configuration. So one editor might be configuring the COM stack (not the communication matrix but the interaction of the individual modules) while another editor is used to configure the RTE.

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

Only when the configuration issues are resolved the ’RTE Generator’ will be used to generate the actual RTE code (see also Section 3.3) which will then be compiled and linked together with the other Basic-SW modules and the SW-Components code.

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

[rte_sws_5001] The RTE Generation tools needs to support input according to the Interoperability of AUTOSAR Authoring Tools document [9].

This is just a rough sketch of the main steps necessary to build an ECU with AUTOSAR and how the RTE is involved in this methodology. For a more detailed description of the AUTOSAR Methodology please refer to the methodology document [2]. In the next sections the steps with RTE interaction are explained in more detail.

3.1 RTE Contract Phase

To be able to support the SW-Component development with RTE-specific APIs the ’Component API’ (application header file) is generated from the ’SW-Component Internal Behavior Description’ (see Figure 3.1) by the RTE-Generator in the so called ’RTE Contract Phase’ (see Figure 3.2).

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

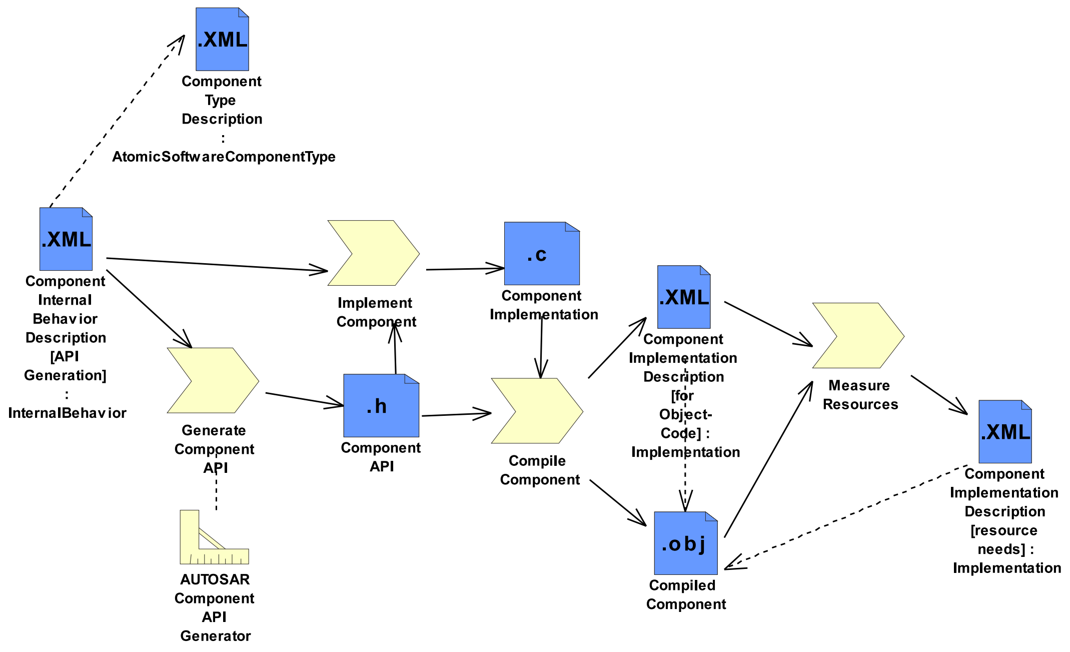

.XML
Component Type Description : AtomicSoftwareComponentType
.XML
Component Internal Behavior Description [API Generation] : InternalBehavior
Implement Component
.c
Component Implementation
.XML
Component Implementation Description [for Object-Code] : Implementation
Measure Resources
.XML
Component Implementation Description [resource needs] : Implementation
Generate Component API
.h
Component API
Compile Component
.obj
Compiled Component
AUTOSAR Component API Generator

Figure 3.2: RTE Contract Phase

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

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

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

So when a SW-Component is delivered it will consist of the following parts:

- Component Type Description
- Component Internal Behavior Description
- The actual source and/or object code
- Component Implementation Description

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

3.2 RTE Configuration Editing

During the configuration of an ECU the RTE also needs to be configured. This is mainly divided into two sections: The configuration of the RTE and the request for configuration of other modules.

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

When the 'RTE Configuration Editor' has gathered all necessary information and built its internal structure it can start to place requirements on the configuration of other modules like COM and OS.

One extremely important point is the mapping of application signals from SW-Component's ports to COM signals. A mapping of the application signals to system signals has already been defined by the 'System Configuration Generator' (see Figure 3.1). The 'RTE Configuration Editor' now has to substantiate this system-level mapping by mapping the application signals to COM signals for the ECU. This application signal to COM signal mapping has to respect the mapping from application signals to system signals done at system generation time. The link between the ECU-specific communication objects and the system-level communication objects is thereby provided by a reference of the Pdu object in the ECU configuration to the frame object in the system configuration.

The usage of 'ECU Configuration Editors' covering different parts of the 'ECU Configuration Description' will – if there are no cyclic dependencies – converge to a stable configuration and then the ECU Configuration process is finished. A detailed description of the ECU Configuration can be found in [7]. The next phase is the generation of the actual RTE.

3.3 RTE Generation Phase

After the ECU has been entirely configured the generation of the actual RTE can be performed. Since all the relationships to and from the other Basic-SW modules have been already resolved during the ECU Configuration phase, the generation can be performed in parallel for all modules (see Figure 3.3).

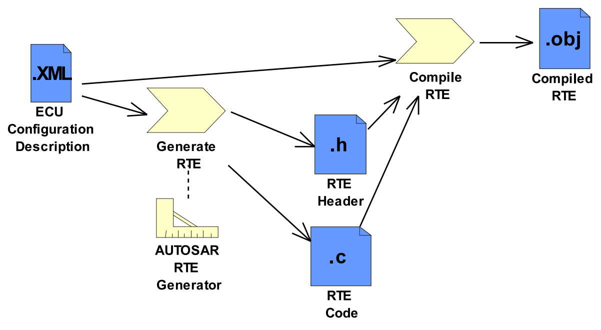


Figure 3.3: RTE Generation Phase

The actual SW-Component and Basic-SW modules code will be linked together with the RTE code to build the entire ECU software.

4 RTE Functional Specification

4.1 Architectural concepts

4.1.1 Scope

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

The Software-Component Template [18] defines the kinds of SW-Components within the AUTOSAR context. These are shown in Figure 4.1. The abstract `ComponentType` can not be instantiated, so there can only be either a `CompositionType` or an `AtomicSoftwareComponentType` of which the `SensorActuatorSoftwareComponent` is a specialization.

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

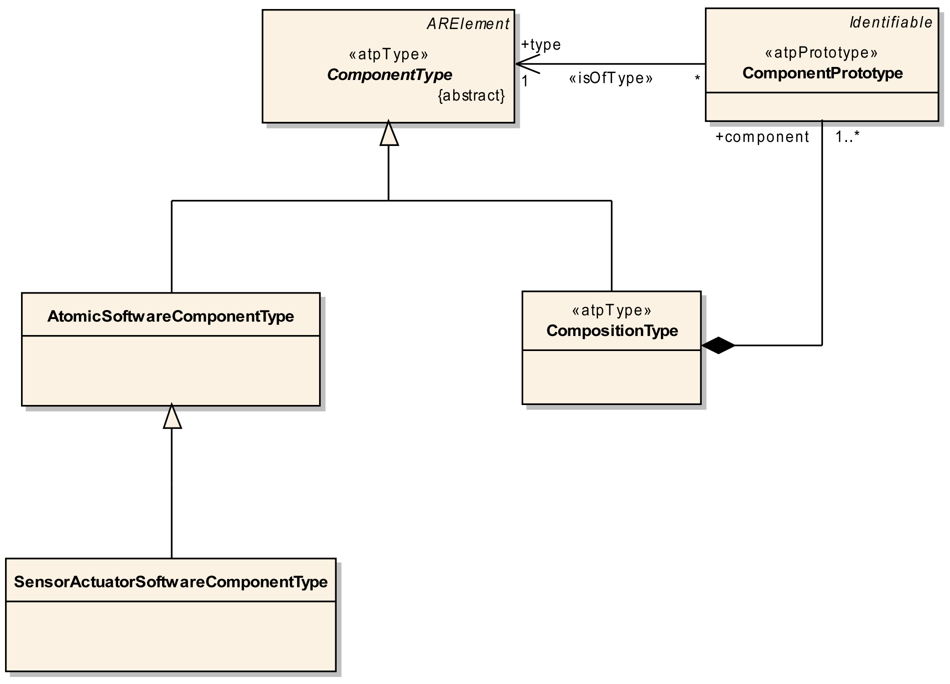


Figure 4.1: AUTOSAR SW-Component classification

The SW-Components shown in Figure 4.1 are located above the RTE in the architectural Figure 4.2.

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

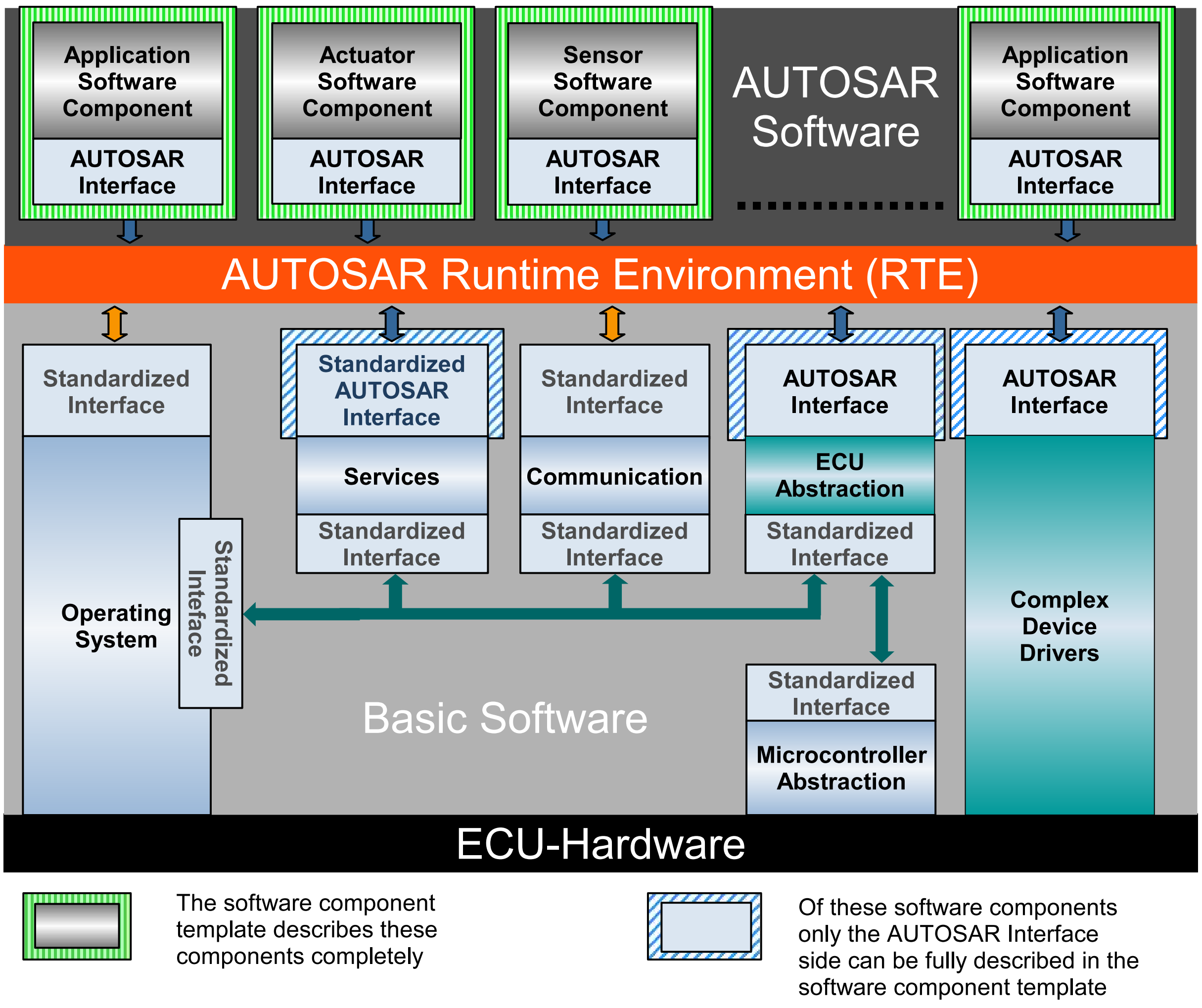

Application Software Component
AUTOSAR Interface
Actuator Software Component
AUTOSAR Interface
Sensor Software Component
AUTOSAR Interface
AUTOSAR Software
Application Software Component
AUTOSAR Interface
AUTOSAR Runtime Environment (RTE)
Standardized Interface
Standardized AUTOSAR Interface
Standardized Interface
AUTOSAR Interface
AUTOSAR Interface
Operating System
Standardized Inteface
Services
Standardized Interface
Communication
Standardized Interface
ECU Abstraction
Standardized Interface
Complex Device Drivers
Basic Software
Standardized Interface
Microcontroller Abstraction
ECU-Hardware
The software component template describes these components completely
Of these software components only the AUTOSAR Interface side can be fully described in the software component template

Figure 4.2: AUTOSAR ECU architecture diagram

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

4.1.2 RTE and AUTOSAR Software-Components

The description of a SW-Component is divided into the sections

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

which will be addressed separately in the following sections.

4.1.2.1 Structure of SW-Components

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

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

4.1.2.2 Ports, Interfaces and Connections

Each SW-Component is providing and/or requiring ports to communicate with other SW-Components. This is shown in Figure 4.3. The Interface determines if the port is a sender/receiver or a client/server port.

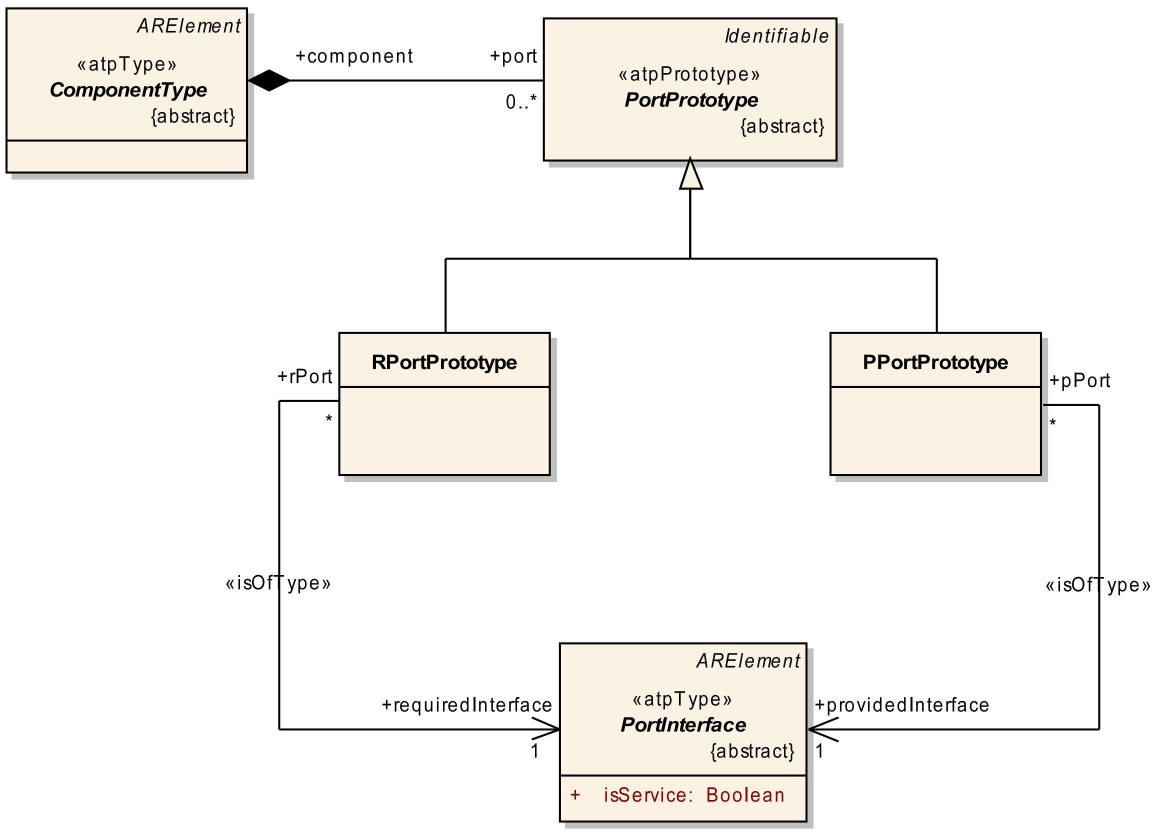


Figure 4.3: SW-Components and Ports

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

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

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

[rte_sws_2201] The RTE shall implement the semantic of the communication attributes given by the SW-Component description (see [RTE00027]). The semantic of the given communication mechanism shall not change regardless of whether the communication partner is located on the same ECU or remote, the communication is done by COM or the RTE.

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

4.1.2.3 Internal Behavior

Only for AtomicSoftwareComponents the internal structure is exposed in the Internal Behavior description. Here the definition of the Runnable Entities and used RTEEvents is done (see Figure 4.4).

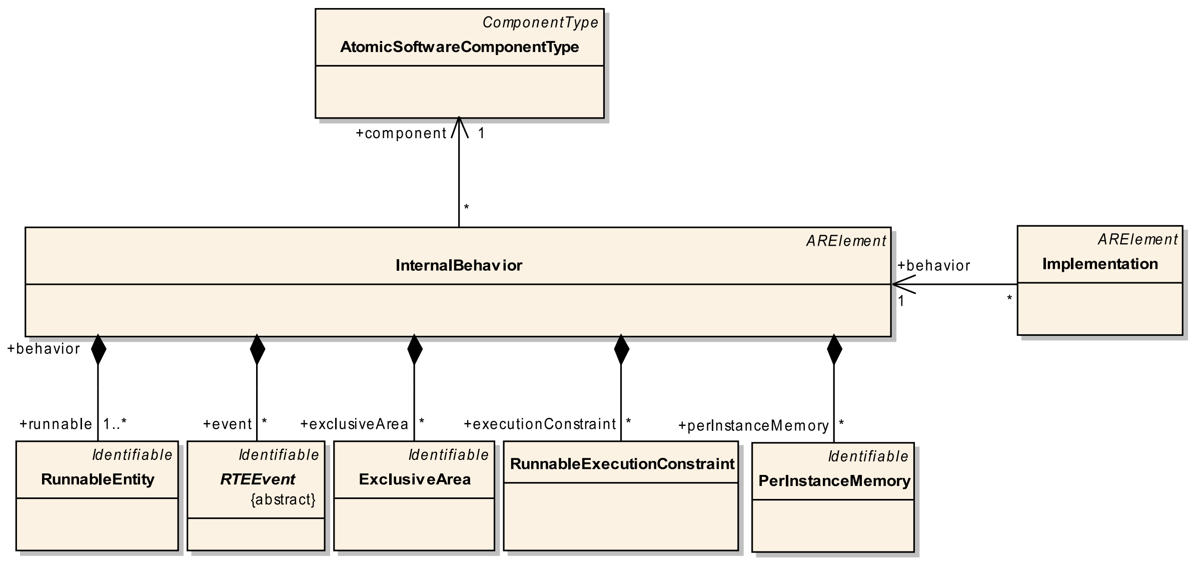


Figure 4.4: SW-Component internal behavior

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

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

[rte_sws_2202] The RTE shall support multiple Runnable Entities in AUTOSAR SW-Components (see [RTE00031]).

Runnable Entities are executed in the context of an OS task, their execution is triggered by RTEEvents. Section 4.2.2.2 gives a more detailed description of the concept of Runnable Entities, Section 4.2.2.4 discusses the problem of mapping Runnable Entities to OS tasks. RTEEvents and the activation of Runnable Entities by RTEEvents is treated in Section 4.2.2.3.

[rte_sws_2203] The RTE shall trigger the execution of Runnable Entities in accordance with the connected RTEEvent (see [RTE00072] and [RTE00093]).

[rte_sws_2204] The RTE-Generator shall reject configurations where not all Runnable Entities are mapped to OS tasks (see [RTE00049] and [RTE00018]). The only exception is a Runnable Entitiy that implements a reentrant server. If it can be invoked via a direct function call, it does not have to be mapped to a task.

[rte_sws_2207] The RTE shall respect the configured execution order of Runnable Entities within one OS task (see [RTE00070]).

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

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

4.1.2.4 Implementation

In the Implementation description an actual implementation of a SW-Component is described including the memory consumption (see Figure 4.6).

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

4.1.3 Instantiation

4.1.3.1 Scope and background

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

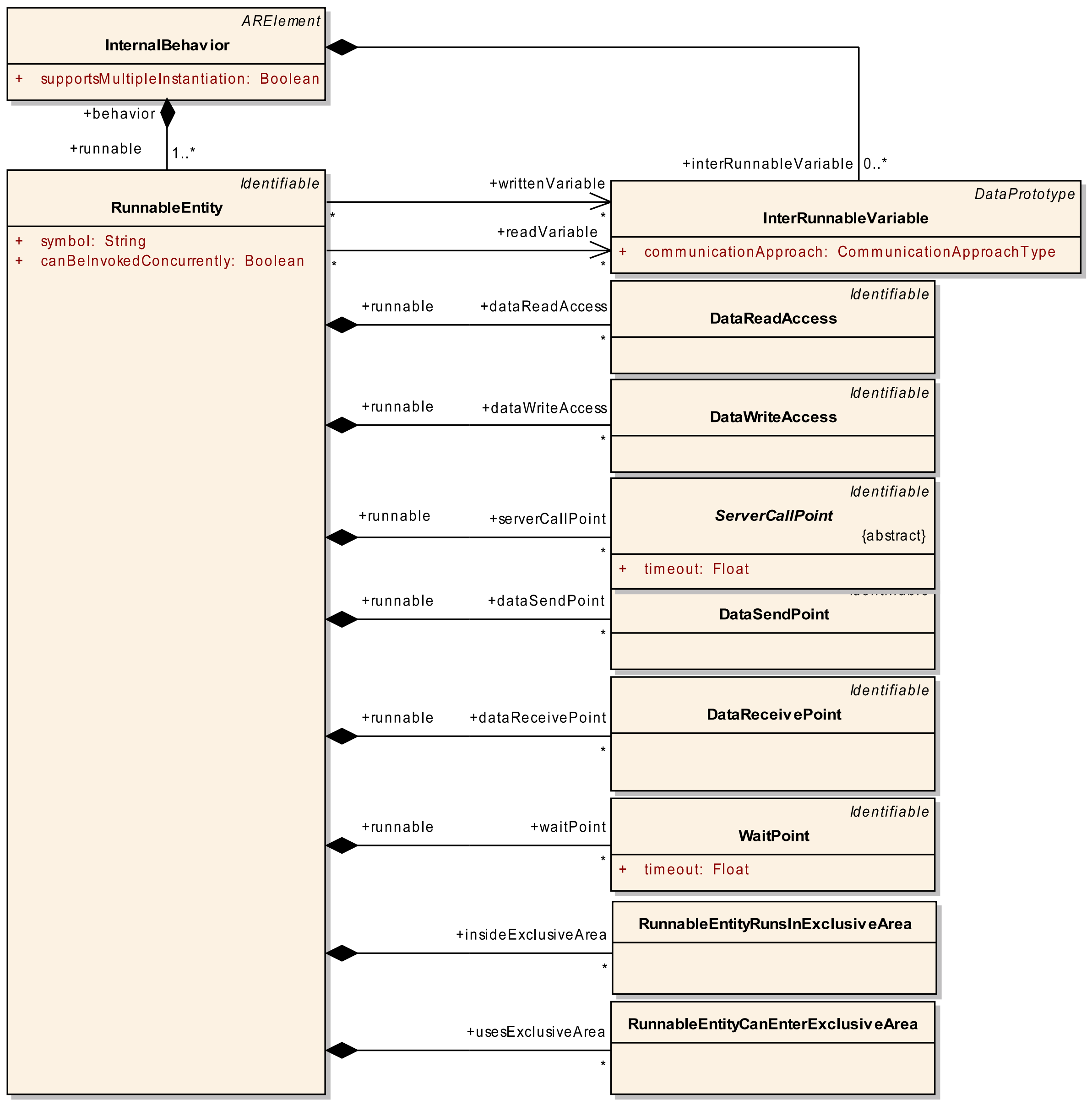


Figure 4.5: SW-Component runnable entity

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

The level $M3$ describes the concepts used to derive an AUTOSAR meta model of level $M2$. This meta model at level $M2$ defines a language in order to be able to describe specific attributes of a model at level $M1$, e.g., to be able to describe an specific type of an AUTOSAR software component. E.g., one part of the AUTOSAR meta model is called *Software Component Template* or *SW-C-T* for short and specified in [18]. It is discussed more detailed in section 4.1.2.

At level $M1$ engineers will use the defined language in order to design components or interfaces or compositions, say to describe an specific *type* of a *LightManager*. Hereby,

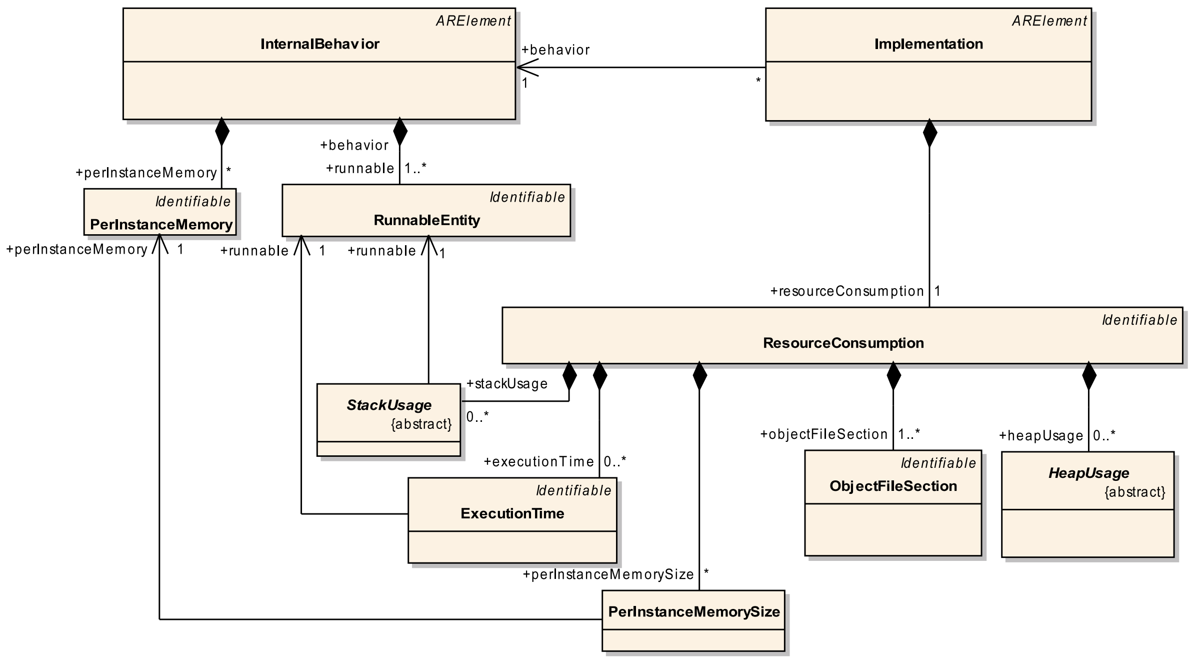


Figure 4.6: SW-Component resource consumption

e.g., the descriptions of the (atomic) software components will also contain an internal behavior as well as an implementation part as mentioned in section 4.1.2.

Those descriptions are input for the RTE-Generator in the so-called 'Contract Phase' (see section 3.1). Out of this information specific APIs (in a programming language) to access ports and interfaces will be generated.

Software components generally consist of a set of Runnable Entities. They can now specifically be described in a programming language which can be refered to as "implementation". As one can seen in section 4.1.2 these "implementation" then correspond exactly to one implementation description as well as to one internal behavior description. However, they are still blueprints on $M1$.

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

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

With respect to this more narrative description the *RTE* as the *run time environment* shall enable the process of instantiation. Thereby the term *instantiation* throughout the document shall refer to the process of deriving $M0$ from $M1$. Therefore, this section will address the problems which can arise out of the instantiation process and will specify the needs for AUTOSAR components and the AUTOSAR RTE respectively.

[rte_sws_2000] The RTE-Generator shall be able to instantiate AUTOSAR software components out of an AUTOSAR software component description.

4.1.3.2 Concepts of instantiation

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

Generally, there are two different kinds of instantiations possible:

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

[rte_sws_2001] The RTE shall be able to cope with one or more AUTOSAR software component instances out of a single AUTOSAR software component description.

[rte_sws_2018] The RTE-Generator and the generated RTE shall be able to cope with the instantiation of “code” and “data”.

[rte_sws_2008] The RTE-Generator shall evaluate the attribute *supportsMultipleInstantiation* of the *InternalBehavior* of an AUTOSAR software component description.

[rte_sws_2009] The RTE-Generator shall reject configurations where multiple instantiation is required, but the value of the attribute *supportsMultipleInstantiation* of the *InternalBehavior* of an AUTOSAR software component description is set to *FALSE*.

4.1.3.3 Single instantiation

Single instantiation refers to the easiest case of instantiation.

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

Example 4.1

Runnable entity R1 called out of a task context:

```
TASK(Task1){
    ...
    R1();
    ...
}
```

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

4.1.3.4 Multiple instantiation

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

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

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

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

[rte_sws_2017] Multiple instantiation shall be achieved by sharing code.

Multiple instances can share the same code, if the code is reentrant.

4.1.3.4.1 Reentrant code

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

Thus, they affect each other. This would lead to data consistency problems on one hand. On the other – and that is even more important – it would introduce a new communication mechanism to AUTOSAR and this is forbidden. AUTOSAR software components can only communicate via ports.

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

In order to be reentrant the following requirements shall be satisfied, though the RTE generator cannot check whether the input satisfies them:

[rte_sws_ext_2006] The code of a runnable entity shall never modify itself, if reentrancy is required.

[rte_sws_ext_2010] The usage of global variables within runnable entities shall be prohibited, if reentrancy is required.

[rte_sws_ext_2011] The usage of local *static* variables within runnable entities shall be prohibited, if reentrancy is required.

4.1.3.4.2 Unambiguous object identification

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

4.1.3.4.3 Multiple instantiation and Per-instance memory

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

PerInstanceMemory API principles are explained in Section 5.2.5.

The API for PerInstanceMemory is specified in Section 5.6.11.

4.1.4 RTE and AUTOSAR Services

According to the AUTOSAR glossary [1] "an AUTOSAR service is a logical entity of the Basic Software offering general functionality to be used by various AUTOSAR software components. The functionality is accessed via standardized AUTOSAR interfaces".

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

[rte_sws_2100] The RTE shall connect ports of AUTOSAR services to ports of AUTOSAR software components of the same ECU, if required.

However, AUTOSAR services are also part of the BSW and can therefore use other BSW modules APIs directly.

[rte_sws_2102] The RTE shall never connect ports of AUTOSAR services to ports of AUTOSAR services of the same or of different ECUs. The RTE-Generator shall reject corresponding input information and/or configurations.

[rte_sws_2310] The RTE shall never connect ports of AUTOSAR services to ports of AUTOSAR software components of different ECUs. The RTE-Generator shall reject corresponding input information and/or configurations.

When connecting AUTOSAR service ports to ports of AUTOSAR software components the RTE has to map standard RTE API calls to the API calls defined for the AUTOSAR services within the BSW. The key technique to distinguish ECU dependent identifiers for the AUTOSAR services is called "port-defined argument values", which is described in Section 4.3.2.4.

[rte_sws_3807] The RTE shall not pass an instance handle to the *C*-based API of AUTOSAR services.

[rte_sws_3808] If a PortInterface is marked as a service port rte_sws_in_0069, the RTE Generator shall map the port-based RTE API of AUTOSAR software components onto the respective API of the AUTOSAR services by using the ‘port-defined argument values” technique.

4.1.5 RTE and ECU Abstraction

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

See the AUTOSAR ECU architecture in figure 4.2. From an architectural point of view the ECU Abstraction is part of the *Basic Software* layer and offers AUTOSAR interfaces to AUTOSAR software components. The *ECU Abstraction* is classified as firmware and will mostly interact with sensor and actuator software components.

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

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

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

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

[rte_sws_ext_2054] The RTE-Generator expects only one instance of the ECU Abstraction.

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

[rte_sws_2051] The RTE-Generator shall reject configurations which require a communication path from a AUTOSAR software component to an ECU Abstraction located on a remote ECU.

Further information about the ECU Abstraction can be found in the corresponding specification document [10].

4.1.6 RTE and Complex Device Driver

A Complex Device Driver does have an AUTOSAR Interface but otherwise does not comply to any further AUTOSAR standard. Therefore the RTE can only deal with the communication on the Complex Device Driver's ports and has no further interaction with its internals.

4.2 RTE Implementation Aspects

4.2.1 Scope

This section describes some specific implementation aspects of an AUTOSAR RTE. It will mainly address

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

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

- the module *AUTOSAR Operating System* [5, 12]
- the module *AUTOSAR COM* [3, 6]

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

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

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

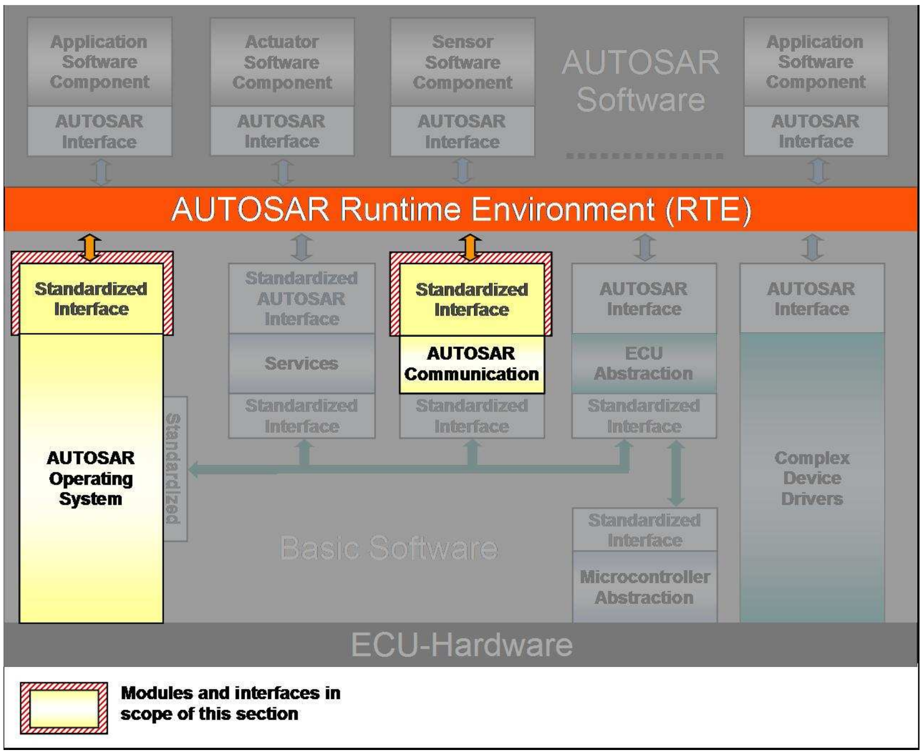


Figure 4.7: Scope of the section on Basic Software modules

With respect to this view the RTE is *thirdly* **not** the abstraction layer for AUTOSAR OS and AUTOSAR COM! Only the RTE offers the same *interface* to the AUTOSAR Software Components like the VFB. For a specific ECU the RTE implements in conjunction with the modules of the Basis Software the entire functionality of the VFB (for that specific ECU). Hence, AUTOSAR OS and AUTOSAR COM are specific modules of a specific implementation of the VFB for a specific ECU. They shall be able to support the implementation of the VFB functionality, but the functionality of the modules are neither known by the AUTOSAR software components nor offered to them per se.

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

[rte_sws_2251] The RTE-Generator shall construct task bodies for those tasks which contain Runnable Entities (see [RTE00049]).

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

[rte_sws_2254] Missing input information for the RTE-Generator regarding the mapping of Runnable Entities to tasks or the construction of tasks bodies shall be taken as an invalid configuration and shall be rejected (see [RTE00049] and [RTE00018]).

4.2.2 OS

This chapter describes the interaction between the RTE and the AUTOSAR OS. The interaction is realized via the standardized interface of the OS - the AUTOSAR OS API. See Figure 4.7.

The OS is statically configured by the ECU-Configuration and not by the RTE generator. The RTE generator is not allowed to create tasks and other OS objects, which are necessary for the runtime environment. Also the mapping of runnable entities to tasks is not the job of the RTE generator. This mapping has to be done in a configuration step before, in the RTE-Configuration phase. The RTE generator is responsible for the generation of task bodies, which contain the calls for the runnable entities. The runnable entities themselves are OS independent and are not allowed to use OS service calls. The RTE has to encapsulate such calls via the standardized RTE API.

As described above, the RTE generation toolkit may be divided in two parts.

- RTE Configurator as one of the AUTOSAR ECU Configuration editors
- RTE Generator

At least all configuration issues, which have impacts to other AUTOSAR BSW, have to be described in the ECU Configuration. An example therefore is the existence of an OS Task, because both, the OS and the RTE have to deal with the task.

4.2.2.1 OS Objects

Tasks

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

OS applications

- The RTE SWS of AUTOSAR Release 2.0 does not support memory protection.

Events

- The RTE may use OS Events for the implementation of the abstract RTEEvents.

- The RTE therefore may call the OS service functions `SetEvent()`, `WaitEvent()`, `GetEvent()` and `ClearEvent()`.
- The used OS Events are part of the input information of the RTE generator.
- The RTE configurator has to allocate the necessary events in the OS configuration.

Resources

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

Interrupt Processing

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

Alarms

- The RTE may use Alarms for timeout monitoring of asynchronous client/server calls. The RTE is responsible for Timeout handling.
- The RTE may setup cyclic alarms for periodic triggering of runnable entities (runnable entity activation via RTEEvent TimingEvent)
- The used Alarms are part of the input information of the RTE generator.
- The RTE configurator has to allocate the necessary alarms in the OS configuration.

Schedule Tables

- The RTE may setup schedule tables for cyclic task activation (runnable entity activation via RTEEvent TimingEvent)
- The used schedule tables are part of the input information of the RTE generator.
- The RTE configurator has to allocate the necessary schedule tables in the OS configuration.

Memory Protection (SCC3/SCC4)

The RTE specification of AUTOSAR Release 2.0 does not support features of the AUTOSAR OS memory protection mechanisms. Nevertheless for future versions of the specification, the RTE is responsible to transfer the data of sender/receiver communication as well as for client-server communication over protection boundaries (OS applications).

Common OS features

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

[rte_sws_4014] The RTE Configurator shall allocate all OS objects in the ECU configuration which are necessary for the generated RTE.

Hook functions

The AUTOSAR OS Specification defines hook functions as follows:

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

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

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

The RTE SWS of AUTOSAR Release 2.0 does not support memory protection. Therefore, only the system-specific hooks are relevant. In the SRS the requirements to implement the system-specific hook functions are rejected [RTE00001], [RTE00101], [RTE00102] and [RTE00105]. The reason for the rejection is the system (ECU) global scope of those functions. The RTE is not the only user of those functions. Other BSW modules might have requirements to use hook functions as well. This is the reason why the RTE is not able to generate these functions without the necessary information of the BSW configuration.

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

4.2.2.2 Runnable Entities

The following chapter describes the runnable entities, their categories and their task-mapping aspects. The prototypes of the functions implementing runnable entities are described in Chapter 5.7

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

All runnable entities are activated by the RTE as a result an RTEEvent. Possible activation events are described in the meta-model by using RTEEvents (see Figure 4.8. RTEEvents are described in the following chapter. If no RTEEvent is specified as StartOnEvent for the runnable entity, the runnable entity is never activated by the RTE.

The runnable entities are categorized as follows. **Category 1**
Category 1 runnable entities do not have *WaitPoints* and have to terminate in *finite* time. With respect to some constraints, category 1 runnable entities can be mapped to *Basic Tasks* of the AUTOSAR OS. The VFB Specification [14] distinguishes between Category 1A and Category 1B runnable entities. For mapping aspects, both sub categories can be handled equally and therfore the term *Category 1* is used instead.

Category 2
In contrast to category 1 runnable entities, runnable entities of category 2 always have at least one *WaitPoint*. Category 2 runnable entities are intended to be mapped into *Extended Tasks*, because only extended tasks provide the task state WAITING. The existence of at least one waitpoint classifies the runnable entity to a category 2 runnable.

Category 3
Runnable entities of category 3 are described in the VFB-Specification [14] in Chapter 4.5.4.4 but are currently out of scope of the RTE Specification. This restriction is also described in Section A.

4.2.2.3 RTE Events

The meta model describes the following RTE events.

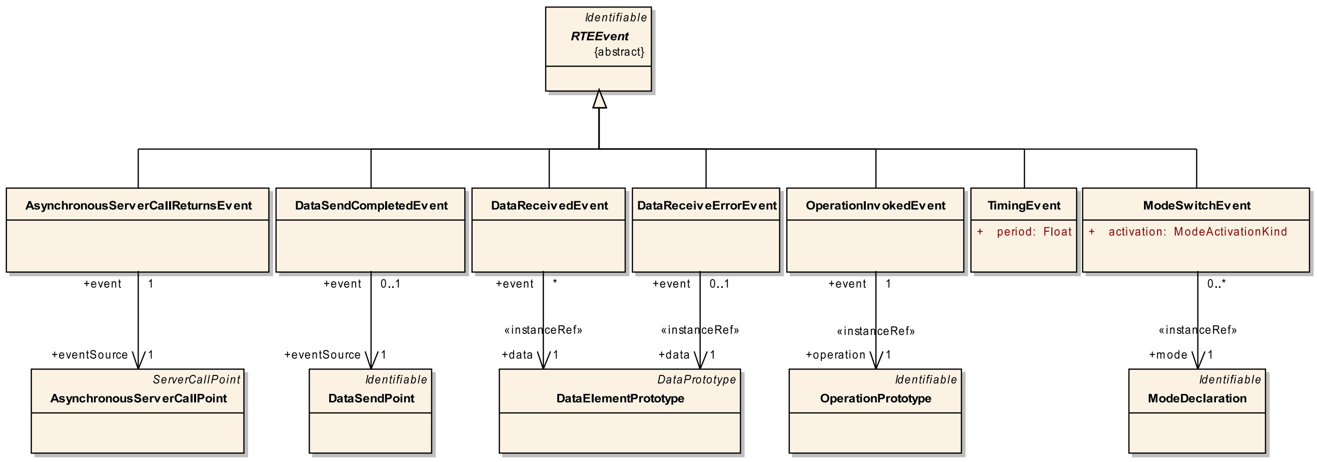


Figure 4.8: Different kinds of RTE-Events

T	TimingEvent
DR	DataReceivedEvent (S/R Communication only)
DRE	DataReceiveErrorEvent (S/R Communication only)
DSC	DataSendCompletedEvent (S/R Communication only)
OI	OperationInvokedEvent (C/S Communication only)
ASCR	AsynchronousServerCallReturnsEvent (C/S communication only)
MS	ModeSwitchEvent

According to the meta model it is possible that all kinds of RTEEvents can either
1.) activate a runnable entity or
2.) wakeup a runnable entity at its waitpoints

The meta model makes no restrictions. As a consequence RTE API functions would be necessary to set up the waitpoints for all kinds of RTEEvents.

Nevertheless in some cases it seems to make no sense to implement all possible combinations of the general meta model. E.g. setting up a waitpoint, which should be resolved by a cyclic TimingEvent. Therefore the RTE SWS of AUTOSAR Release 2.0 makes some restrictions, which are also described in Section A.

The meta model also allows, that the same runnable entity can be triggered by several RTEEvents. For the current approach of the RTE and restrictions see Section 4.2.5.

	T	**DR**	**DRE**	**DSC**	**OI**	**ASCR**	**MS**
Activation of runnable entity	x	x	x	x	x	x	x
Wakeup of waitpoint		x		x		x	

The table shows, that *activation of runnable entity* is possible for all kinds of RTEEvents. For runnable entity activation, no explicit RTE API is necessary. The RTE itself is responsible for the activation of the runnable entity depending on the configuration in the SW-C Description.

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

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

- DataReceivedEvent: `Rte_Receive()`
- DataSendCompletedEvent: `Rte_Feedback()`
- AsynchronousServerCallReturnsEvent: `Rte_Result()`

4.2.2.4 Mapping of runnable entities to tasks

One of the main requirements of the RTE is ”Construction of task bodies” [RTE00049]. The necessary input information e.g. the mapping of runnable entities to tasks must be provided by the ECU configuration description.

The ECU configuration description (or an extract of it) is the input for the RTE-Generator (see Figure 3.3). It is also the purpose of this document to define the necessary input information. Therefore the following scenarios may help to derive requirements for the ECU-Configuration Template as well as for the RTE-generator itself.
Note: The scenarios do not cover all possible combinations.
The RTE-Configurator configures parts of the ECU-Configuration, e.g. the mapping of runnable entities to tasks. In this configuration process the RTE-Configurator also allocates those OS-objects (e.g. Tasks, Events, Alarms...) which are used in the generated RTE. The RTE-Configurator must be the **owner** of these configuration items. Other configurators, e.g. the OS Configurator, should not be able to change these settings.

Some figures for better understanding use the following conventions:

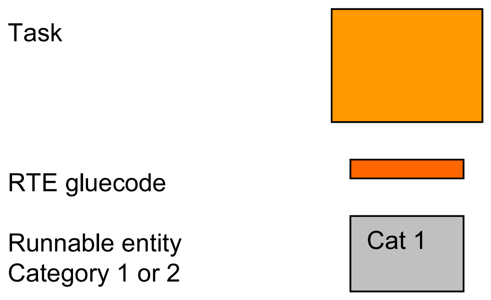


Figure 4.9: Element description

4.2.2.4.1 Scenario for mapping of runnable entities to tasks

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

- Runnable entities using (implicit) DataReadAccess/DataWriteAccess have to terminate.
- Runnable entities using (implicit) DataReadAccess/DataWriteAccess are category 1 runnables (1A or 1B). Runnable entities of category 2 do not allow (implicit) DataReadAccess/DataWriteAccess.
- Runnable entities of category 1 can be mapped either to basic or extended tasks. (see next subsection).
- Runnable entities using at least one Waitpoint are of category 2.
- Runnables of category 2 which contain WaitPoints will be typically mapped to extended tasks.

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

4.2.2.4.1.1 Scenario 1

Runnable entity category 1A: ”runnable1”

- Ports: only S/R with DataReadAccess / DataWriteAccess
- RTEEvents: TimingEvent
- no sequence of runnable entities specified
- no explicit DataSendPoint
- no WaitPoint

Possible mappings of ”runnable1” to tasks:

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

The ECU-Configuration-Template has to provide the sequence of runnable entities mapped to the same task, see rte_sws_in_0014.

Figure 4.10 shows the possible mappings of runnable entities into a basic task. Only if a sequence order is specified can more than one runnable entity be mapped into a basic task.

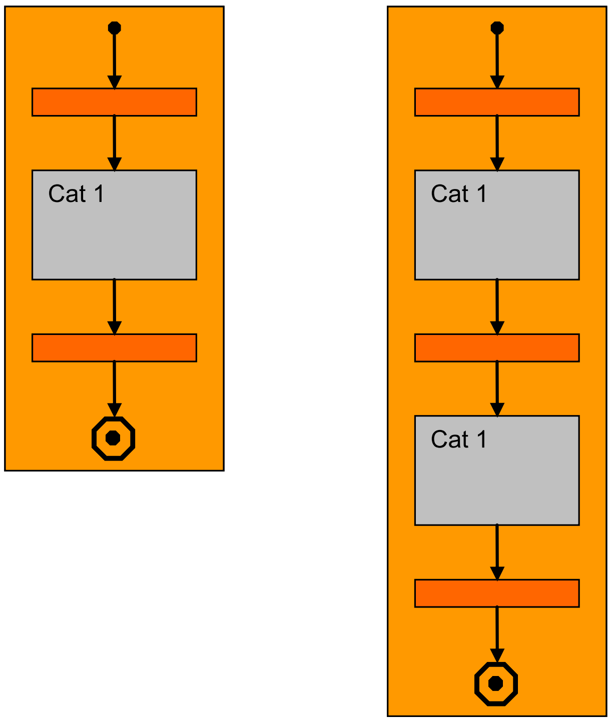


Figure 4.10: Mapping of Category 1 runnable entities to Basic Tasks

Extended Task

If more than one runnable entity is mapped to the same task and the special condition (same cycletime) does not fit, an extended task is used.

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

The ECU-Configuration has to provide the OS events assigned to the RTEEvents triggering the runnable entities that are mapped to an extended task, see rte_sws_in_0039.

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

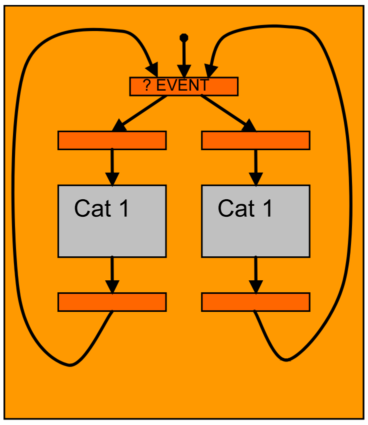


Figure 4.11: Mapping of Category 1 runnable entities to Extended Tasks

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

The ECU-Configuration has to provide the name of the task, see rte_sws_in_5012.

The ECU-Configuration has to provide the task type (BASIC or EXTENDED), which can be determined from the presence or absence of OS Events associated with that task, see rte_sws_in_0040.

4.2.2.4.1.2 Scenario 2

Runnable entity category 1B: ”runnable2”

- Ports: S/R with DataSendPoints.
- RTEEvents: TimingEvent
- no sequence of runnables specified
- no WaitPoint

Possible mappings of ”runnable2” to tasks:

The following figure shows the different mappings:

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

The gluecode to realize the DataReadAccess and DataWriteAccess respectively before entering the runnable and after exiting is not necessary.

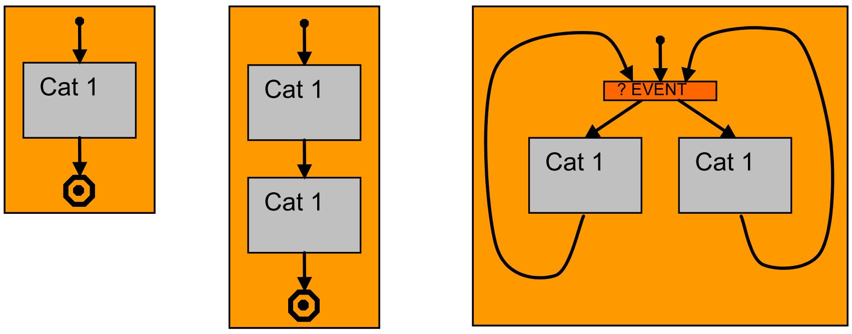


Figure 4.12: Mapping of Category 1 runnable entities using no DataReadAccess / DataWriteAccess

4.2.2.4.1.3 Scenario 3

Runnable entity category 1A: ”runnable3”

- Ports: S/R with DataReadAccess / DataWriteAccess
- RTEEvents: Runnable is activated by a DataReceivedEvent
- no sequence of runnables specified
- no DataSendPoint
- no WaitPoint

There is no difference between Scenario 1. Only the RTEEvent that activates the runnable entity is different.

4.2.2.4.1.4 Scenario 4

Runnable entity category 2: ”runnable4”

- Ports: S/R with DataReceivePoint and WaitPoint (blocking read)
- RTEEvents: WaitPoint referencing a DataReceivedEvent

- no sequence of runnables specified

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

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

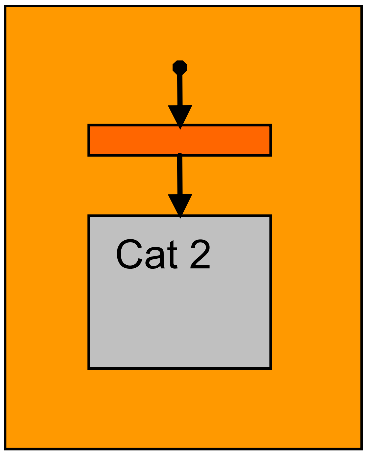


Figure 4.13: Mapping of Category 2 runnable entities to Extended Tasks

4.2.2.4.1.5 Scenario 5

There are two runnable entities (category 1B) implementing a client and a server for synchronous C/S communication and the timeout attribute of the ServerCallPoint is 0.

There are two ways to invoke a server synchronously:

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

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

4.2.2.4.1.6 Scenario 6

There are two runnable entities (category 1B) implementing a client and a server for synchronous C/S communication and the timeout attribute of the ServerCallPoint is greater than 0.

There are again two ways to invoke a server synchronously:

- Simple function call for intra-ECU C/S communication if the canBeInvokedConcurrently attribute of the server runnable is set. In that case the server runnable is executed in the same task context (same stack) as the client runnable that has invoked the server and no timeout monitoring is performed (see rte_sws_3768).
- The server runnable is mapped to its own task. If the canBeInvokedConcurrently attribute is not set, the server runnable must be mapped to a task.

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

4.2.2.4.1.7 Scenario 7

Runnable entity category 2: ”runnable7”

- Ports: only C/S with AsynchronousServerCallPoint and WaitPoint
- RTEEvents: AsynchronousServerCallReturnsEvent (C/S communication only)
- no sequence of runnables specified

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

4.2.3 Notifications

4.2.3.1 Basic notification principles

Several BSW modules exist which contain functionality which is not directly activated, triggered or called by AUTOSAR software-components but by other circumstances, like digital input port level changes, complex driver actions, CAN signal reception, etc. In most cases interrupts are a result of those circumstances. For a definition of interrupts, see the VFB [14].

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

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

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

1. **BSW with Standardized interface**. Used with COM and OS.
   Basic-SW modules with Standardized interfaces cannot create RTEEvents. So another mechanism must be chosen: ”**callbacks**”
   The typical callback realization in a C/C++ environment is a function call.

2. **BSW with AUTOSAR interface**: Used in all the other BSW modules.
   Basic-SW modules with AUTOSAR-Interfaces have their interface specified in an AUTOSAR BSW description XML file which contains signal specifications according to the AUTOSAR specification. The BSW modules can employ RTE API calls like Rte_Send – see *5.6.4*). **RTEEvents** may be connected with the RTE API calls, so realizing AUTOSAR SW-C activation.

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

4.2.3.2 Interrupts

The AUTOSAR concept as stated in the VFB specification [14] does not allow AUTOSAR software components to run in interrupt context. Only the Microcontroller Abstraction Layer, Complex Device Drivers and the OS are allowed to directly interact with interrupts and implement interrupt service routines (see Requirement BSW164). This ensures hardware independency and determinism.

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

Furthermore, AUTOSAR software components are not allowed to explicitly block interrupts as a means to ensure data consistency. They have to use RTE functions for this purpose instead, see Section 4.2.4.

4.2.3.3 Decoupling interrupts on RTE level

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

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

Example 1:

Situation:

- An interrupt routine calls an RTE callback function

Intention:

- Start a runnable

RTE job:

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

Example 2:

Situation:

- An interrupt routine calls an RTE callback function

Intention:

- Make a runnable wake up from a wait point

RTE job:

- RTE sets an OS event

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

[rte_sws_3600] The RTE shall prevent runnable entities of AUTOSAR software-components to run in interrupt context.

4.2.3.3.1 Interrupt decoupling for COM

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

[rte_sws_3530] The RTE has to provide callback functions to allow COM to signal COM events to the RTE.

[rte_sws_3531] The RTE has to support runnable activation by COM callbacks.

[rte_sws_3532] The RTE has to support cat2 runnables to wake up from a wait point as a result of COM callbacks.

See RTE callback API in chapter 5.9.

4.2.4 Data Consistency

4.2.4.1 General

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

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

The ”lost update” example in Figure 4.14 illustrates the problem for concurrent read-modify-write accesses:
There are two tasks. Task A has higher priority than task B. A increments the commonly accessed counter X by 2, B increments X by 1. So in both tasks there is a read (step1) – modify (step2) – write (step3) sequence. If there are no atomic accesses (fully completed read-modify-write accesses without interruption) the following can happen:

1. Assume X=5.
2. B makes read (step1) access to X and stores value 5 in an intermediate store (e.g. on stack or in a CPU register).
3. B cannot continue because it is preempted by A.

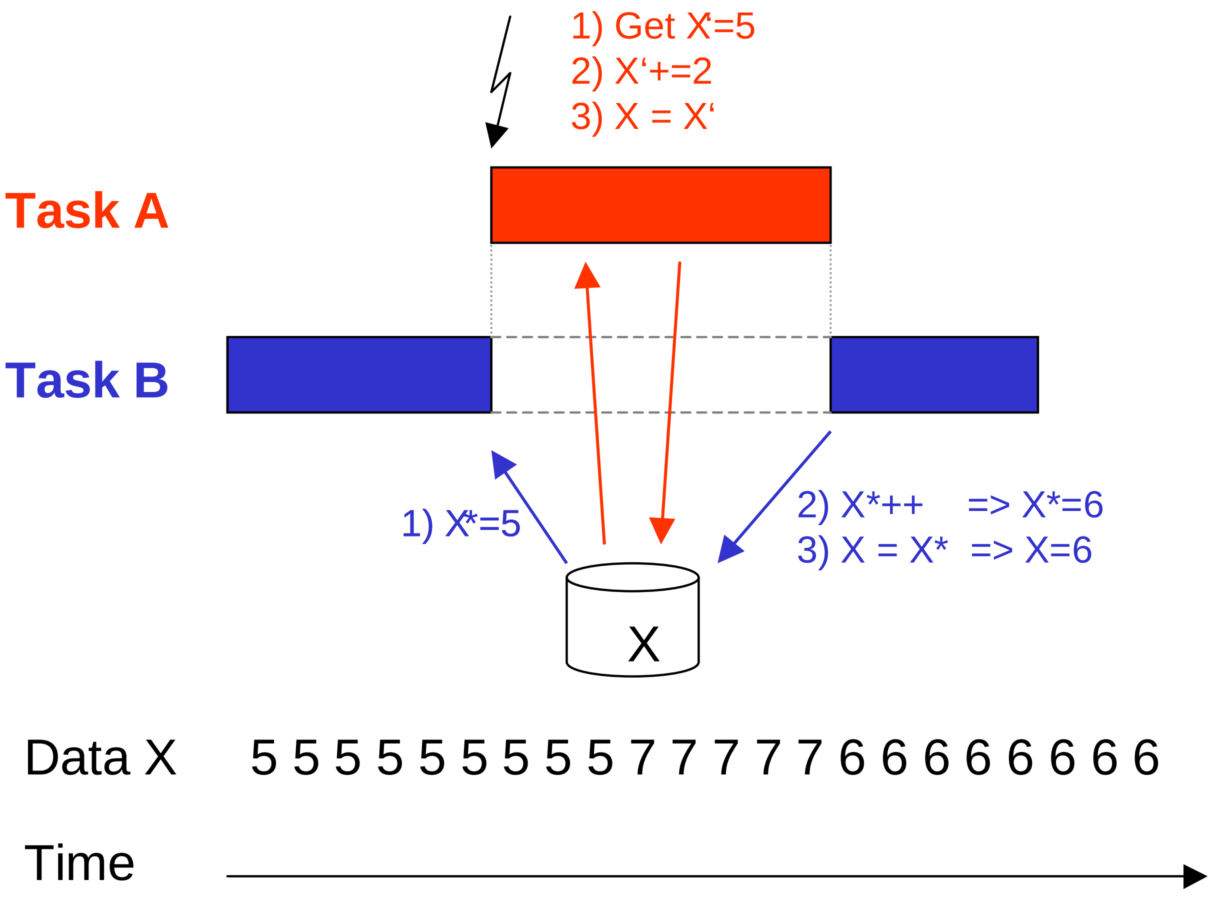


Figure 4.14: Data inconsistency example - lost update

4. A does its read (step1) – modify (step2) – write (step3) sequence, which means that A reads the actual value of X, which is 5, increments it by 2 and writes the new value for X, which is 7. (X=5+2)
5. A is suspended again.
6. B continues where it has been preempted: with its modify (step2) and write (step3) job. This means that it takes the value 5 form its internal store, increments it by one to 6 and writes the value 6 to X (X=5+1).
7. B is supended again.

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

4.2.4.2 Communications to look at

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

The following communication mechanisms can be distinguished:

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

[rte_sws_3514] The RTE has to guarantee data consistency for communication via AUTOSAR interfaces.

4.2.4.3 Concepts

In the AUTOSAR SW-C Template [18] chapter ”Interaction between runnables within one component”, the concepts of

1. ExclusiveAreas *(see section 4.2.4.5 below)*
2. InterRunnableVariables *(see section 4.2.4.6 below)*

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

The AUTOSAR SW-C template specification [18] also states that AUTOSAR SW-Cs may define **PerInstanceMemory**, allowing reservation of static (permanent) need of global RAM for the SW-C. Nothing is specified about the way Runnables might access this memory. RTE only provides a reference to this memory *(see section 5.6)* but doesn't guarantee data consistency for it.

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

4.2.4.4 Mechanisms to guarantee data consistency

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

The data consistency guaranteeing mechanisms listed here are derived from AUTOSAR SW-C Template and from further discussions. There might be more.
The RTE has the responsibility to apply such mechanisms if required. The details how to apply the mechanisms are left open to the RTE supplier.

Mechanisms:

- ***Sequential scheduling strategy***
  The activation code of Runnables is sequentially placed in one task so that no interference between them is possible because one Runnable is only activated after the termination of the other. Data consistency is guaranteed.

- ***Task blocking strategy***
  OS prohibits mutually preemption of tasks with Runnables accessing common data to prohibit data inconsistencies. If required, the RTE has to assign an access blocking mechanism around the complete Runnable.

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

- ***Cooperative Runnable placement strategy***
  The principle is that tasks containing Runnables to be protected by ”Cooperative Runnable placement strategy” are not allowed to preempt other tasks also containing Runnables to be protected by ”Cooperative Runnable placement strategy” when one of the Runnables to protect is active - but are allowed between Runn-

able executions. The RTE's job is to create appropriate task bodies and use OS services or other mechanisms to achieve the required behavior.

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

Example to explain the cooperative mechanism:

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

This setup results in this behavior:

- T4 can always preempt all other tasks (Higher prio than all others).
- T3b can preempt T2 (higher prio of T3b, no cooperative restriction)
- T3a cannot preempt T2 (Higher prio of T3a but same cooperative context). So data access of Runnable R2 to common data cannot interfere with data access by Runnable R3a. Nevertheless if both tasks T3a and T2 are ready to run, it's guaranteed that T3a is running first.
- T1 can never preempt one of the other tasks because of lowest assigned prio.

• ***Interrupt blocking strategy***
Simple interrupt blocking can be an appropriate means if collision avoidance is required for a very short amount of time. This might be done by disabling or resuming all interrupts or - if hardware supports it - by only disabling some interrupt levels. In general this mechanism must be applied with care because it might influence the timing of the complete system.

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

For AUTOSAR SW-C internal comunication this strategy is addressed by the ***InterRunnableVariables with implicit behavior*** (see 4.2.4.6.1)

How it can work:

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

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

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

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

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

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

Further issues to be taken into account:

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

4.2.4.5 Exclusive Areas

The concept of ExclusiveArea is more a working model. It's not a concrete implementation approach, although concrete possible mechanisms are listed in AUTOSAR SW-C template specification [18].

Focus of the ExclusiveArea concept is to block potential concurrent accesses to get data consistency.

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

- If an AUTOSAR SW-C requests the RTE to look for data consistency for it's internally used data (for a part of it or the complete one) using the ExclusiveArea concept, the SW designer can use the API calls ”Rte_Enter()” in 5.6.21 and ”Rte_Exit()” in 5.6.22 to specify where he wants to have the protection by RTE applied.
  ”Rte_Enter()” defines the begin and ”Rte_Exit()” defines the end of the code sequence containing data accesses the RTE has to guarantee data consistency for.
- If the SW designer wants to have the mutual exclusion for complete Runnables he can specify this by setting the attribute ”RunnableEntityRunsInExclusiveArea” in the AUTOSAR SW-C description.

Principly the ExclusiveArea concept can handle the access to single data items as well as the access to several data items realized by a group of instructions. It also doesn't matter if one Runnable is completely running in an ExclusiveArea and another Runnable only temporarily enters the same ExclusiveArea. The RTE has to guarantee data consistency.

[rte_sws_3500] The RTE has to guarantee data consistency for arbitrary accesses to data items accessed by Runnables marked with the same ExclusiveArea.

[rte_sws_3515] RTE has to provide an API enabling the SW-Cs to access and leave ExclusiveAreas.

[rte_sws_3502] If Runnables accessing same ExclusiveArea are assigned to be executing in different task contexts, the RTE can apply e.g. task blocking means to guarantee data consistency for data accesses in the common ExclusiveArea; however, for optimisation purposes, the RTE is still free to select another data consistency guaranteeing mechanism. In spite of that, if specials attributes are set requiring special data consistency mechanisms, the RTE is forced to apply the selected mechanism.

4.2.4.5.1 Strategy to assign data consistency mechanisms for ExclusiveAreas

An AUTOSAR SW-C designer can specify that RTE should apply a special data consistency mechanism by setting the ExclusiveArea attribute *executionOptimization*. ”Should” because the attribute content is meant as a hint for the ECU integrator what the SW-C designer thinks what is best for the SW-C out of his view.

[rte_sws_3503] If an ExclusiveArea is marked with the attribute *executionOptimization* set to *”codeSize”* the RTE generator has to generate appropriate code according the

Cooperative Runnable placement strategy to guarantee data consistency if data inconsistency could occur.

Requirement rte_sws_3503 has to be seen in combination with requirement rte_sws_3507.

Strategy is described in section 4.2.4.4. CodeSize efficiency is reached because additional RAM, code, runtime needs inside of Runnables to guarantee data consistency are avoided even if a lot of complex data is commonly accessed by Runnables in different tasks. (E.g. compared to applying the mechanism of interrupt blocking.) Costs might be some few instructions between the Runnable activation calls.

[rte_sws_3504] If an ExclusiveArea is marked with the attribute *executionOptimization* set to *"executionTime"* the RTE generator has to use the mechanism of *Interrupt blocking* to guarantee data consistency if data inconsistency could occur.

Requirement rte_sws_3504 has to be seen in combination with requirement rte_sws_3507.

Mechanism is described in section 4.2.4.4. Addressed is to optimize overall system timing, not the execution time of the Runnable itself. So get fastest activation of Runnables running in task context with higher priorities. This is reached by reducing the time Runnables in tasks with lower priority might block Runnables in tasks with higher priority to a minimum.

Requirements rte_sws_3503 and rte_sws_3504 have the limitation "if data inconsistency could occur" because it makes no sense to apply a data consistency mechanism if no potential data inconsistency may occur. This can be the case if e.g. the "Sequential scheduling strategy" (described in section 4.2.4.4) still has solved the item by the ECU integrator defining an appropriate runnable-to-task mapping.

[rte_sws_3507] The RTE has to apply data consistency mechanisms for each ExclusiveArea according following sequence:

1. At first the RTE generator has to check its input from ECU configuration if a specific data consistency mechanism is defined. If a mechanism is defined it has to be applied.
2. If ECU configuration doesn't define a mechanism, secondly the RTE generator has to check the AUTOSAR SW-C if the attribute *executionOptimization* is set. If it is set the RTE generator shall apply the requested mechanism.
3. If *executionOptimization* isn't set the RTE generator has to reject the configuration

Requirement rte_sws_3507 is based on these ideas:

- If the attribute *executionOptimization* has been set in AUTOSAR SW-C description the ECU configuration description gets it as an input from AUTOSAR SWC description initially. As mentioned before this is meant as a hint for the ECU integrator.

- The ECU integrator with his ECU wide view is able to define the same or another another mechanism as *executionOptimization* proposes, so overruling the *executionOptimization* attribute proposal. This is done by using the RTE configurator during ECU configuration.

4.2.4.6 InterRunnableVariables

A non-composite AUTOSAR SW-C can reserve InterRunnableVariables which can be accessed by the Runnables of this one AUTOSAR SW-C (also see section 4.3.3.1). Read and write accesses are possible.

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

[rte_sws_3516] The RTE has to guarantee data consistency for communication between Runnables of one AUTOSAR SW-Component using InterRunnableVariables.

Next the two kinds of InterRunnableVariables are treated:

1. InterRunnableVariables with **implicit** behavior
2. InterRunnableVariables with **explicit** behavior

4.2.4.6.1 InterRunnableVariables with implicit behavior

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

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

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

[rte_sws_3517] The RTE shall guarantee data consistency for *InterRunnableVariables with implicit behavior* by avoiding concurrent accesses to data items specified by InterRunnableVariables using one or more copies and redirecting accesses to the copies.

Compared with Sender/Receiver communication

- Like with DataReadAccess/DataWriteAccess the Runnable IN data is stable during Runnable execution, which means that during an Runnable execution several read accesses to an InterRunnableVariable always deliver the same data item value.

- Like with DataWriteAccess/DataWriteAccess the Runnable OUT data is forwarded to other Runnables not before Runnable execution has terminated, which means that during an Runnable execution write accesses to InterRunnableVariable are not visible to other Runnables.

This behavior requires that Runnable execution terminates.

[rte_sws_3582] Several read accesses to *InterRunnableVariables with implicit behavior* during a Runnable execution shall always deliver the same data item value.

[rte_sws_3583] Several write accesses to *InterRunnableVariables with implicit behavior* during a Runnable execution shall result in only one update of the InterRunnableVariable content visible to other Runnables with the last written value.

[rte_sws_3584] The update of *InterRunnableVariables with implicit behavior* done during a Runnable execution shall be made available to other Runnables after the Runnable execution has terminated.

The usage of *InterRunnableVariables with implicit behavior* shall be valid for category 1a and 1b Runnable entities. Usage in category 2 (and 3) Runnables is not allowed because there Runnable termination is not guaranteed and so it's not guaranteed that other Runnables will ever get the updated data. See also requirement rte_sws_3518.

For API of *InterRunnableVariables with implicit behavior* see sections 5.6.17 and 5.6.18.

For more details how this mechanism could work see "Copy strategy" in section 4.2.4.4.

4.2.4.6.2 InterRunnableVariables with explicit behavior

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

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

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

The mechanism behind is the same as in the ExclusiveArea concept (see section 4.2.4.5). But although ExclusiveAreas can handle single data item accesses too, their API is made to make the RTE to apply data consistency means for a group of in-

structions accessing several data items as well. So when using an ExclusiveArea to protect accesses to one single common used data item each time two RTE API calls grouped around are needed. This is very inconvenient and might lead to faults if the calls grouped around might be forgotton.
The solution is to support *InterRunnableVariables with explicit behavior*.

[rte_sws_3519] The RTE shall guarantee data consistency for *InterRunnableVariables with explicit behavior* by blocking concurrent accesses to data items specified by InterRunnableVariables.

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

[rte_sws_3580] The RTE shall supply different APIs for *InterRunnableVariables with implicit* behavior and *InterRunnableVariables with explicit* behavior.

For API of *InterRunnableVariables with explicit behavior* see sections 5.6.19 and 5.6.20.

4.2.5 Multiple trigger of Runnables

Concurrent activation

The AUTOSAR SW-C template specification [18] states that runnable entities (further called "Runnables") might be invoked concurrently several times if the Runnables attribute "canBeInvokedConcurrently" is set. It's then in the responsability of the AUTOSAR SW-C designer that no data might be corrupted when the Runnable is activated several times in parallel.

[rte_sws_3523] The RTE has to support concurrent activation of the same instance of a runnable entity if the associative attribute "canBeInvokedConcurrently" is set to TRUE. This includes concurrent activation in several tasks. If the attribute is not set resp. set to FALSE, concurrent activation of the runnable entity is forbidden.

Activation by several RTEEvents

Nevertheless a Runnable whose attribute "canBeInvokedConcurrently" is NOT set might be still activated by several RTEEvents if activation configuration guarantees that concurrent activation can never occur. This includes activation in different tasks. A standard use case is the activation of same instance of a runnable in different modes.

[rte_sws_3520] The RTE supports activation of same instance of a runnable entity by multiple RTEEvents.

RTEEvents are triggering Runnable activation and may supply 0..several role parameters, see *section 5.7.3*. Role parameters are not visible in the Runnables signature - except in those triggered by an OperationInvokedEvent. With the exception of the

RTEEvent *OperationInvokedEvent* all role parameters can be accessed by user with implicit or explicit Receiver API.

[rte_sws_3524] The RTE supports activation of same instance of a runnable entity by RTEEvents of different kinds.

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

4.3 Communication Models

AUTOSAR supports two basic communication patterns: Client-Server and Sender-Receiver. AUTOSAR software-components communicate through well defined ports and the behavior is statically defined by attributes. Some attributes are defined on the modeling level and others are closely related to the network topology and must be defined on the implementation level.

The RTE provides the implementation of these communication patterns. For inter-ECU communication the RTE uses the functionalities provided by COM. For intra-ECU communication the RTE can use the services of COM, but may as well implement the functionality on its own if that is more efficient.

With Sender-Receiver communication there are two main principles: Data Distribution and Event Distribution. When data is distributed, the last received value is of interest (last-is-best semantics). When events are distributed the whole history of received events is of interest, hence they must be queued on receiver side. Therefore an 'isQueued' attribute of the data element is used to distinguish between Data and Event Distribution. [1] If a data element has event semantics, the isQueued attribute is set to true, if the data element has data semantics, the isQueued attribute is set to false.

4.3.1 Sender-Receiver

4.3.1.1 Introduction

Sender-receiver communication involves the transmission and reception of signals consisting of atomic data elements that are sent by one component and received by one or more components. A sender-receiver interface can contain multiple data elements. Sender-receiver communication is one-way - any reply sent by the receiver is sent as a separate sender-receiver communication.

[rte_sws_5508] The RTE generator shall reject the configuration when an r-port is connected to an r-port or a p-port is connected to a p-port. A require port (r-port) of a component typed by an AUTOSAR sender-receiver interface can read data elements of this interface. A provide port (p-port) of a component typed by an AUTOSAR sender-receiver interface can write data elements of this interface.

[1] The isQueued attribute corresponds to the VFB attribute INFORMATION_TYPE.

4.3.1.2 Receive Modes

The RTE supports multiple receive modes for passing data to receivers. The four possible receive modes are:

- **"Implicit data read access"** – when the receiver's runnable executes it shall have access to a "copy" of the data that remains unchanged during the execution of the runnable.

  [rte_sws_6000] For data elements specified with implicit data read access, the RTE shall make the receive data available to the runnable through the semantics of a copy [RTE00128].

  [rte_sws_6001] For data elements specified with implicit data read access the receive data shall not change during execution of the runnable [RTE00128].

  When "implicit data read access" is used the RTE is required to make the data available as a "copy". It is not necessarily required to use a unique copy for each runnable. Thus the RTE may use a unique copy of the data for each runnable entity or may, if several runnables (even from different components) need the same data, share the same copy between runnables. Runnable entities can only share a copy of the same data when the scheduling structure can make sure the contents of the data is protected from modification by any other party.

  [rte_sws_6004] The RTE shall read the data elements specified with implicit data read access before the associated runnable entity is invoked [RTE00128].

  Complex data types shall be handled in the same way as primitive data types, i.e. RTE shall make a "copy" available for the runnable.

  [rte_sws_6003] The "implicit data read access" receive mode shall be valid for category 1A and 1B runnable entities [RTE00134].

- **"Explicit data read access"** – the RTE generator creates a non-blocking API call to enable a receiver to poll (and read) data. This receive mode is an "explicit" mode since an explicit API call is invoked by the receiver.

  [rte_sws_6005] The explicit "data read access" receive mode shall only be valid for category 1B or 2 runnable entities [RTE00134].

- **"wake up of wait point"** – the RTE generator creates a blocking API call that the receiver invokes to read data.

  [rte_sws_6002] The "wake up of wait point" receive mode shall support a timeout to prevent infinite blocking if no data is available [RTE00109].

  [rte_sws_6006] The "wake up of wait point" receive mode shall only be valid for a category 2 runnable entity [RTE00134].

  A category 2 runnable entity is required since the implementation may need to suspend execution of the caller if no data is available.

- **"activation of runnable entity"** – the receiving runnable entity is invoked automatically by the RTE whenever new data is available. To access the new data, the runnable entity either has to use “implicit data read access” or “explicit data read access”, i.e. invoke an `Rte_Read` or `Rte_Receive` call, depending on the input configuration. This receive mode differs from “implicit data read access” since the receiver is invoked by the RTE in response to a DataReceivedEvent.

[rte_sws_6007] The “activation of runnable entity” receive mode shall be valid for category 1A, 1B and 2 runnable entities [RTE00134].

The validity of receive modes in conjunction with different categories of runnable entity is summarized in Table 4.1.

Receive Mode	**Cat 1A**	**Cat 1B**	**Cat 2**
Implicit Data Read Access	Yes	Yes	No
Explicit Data Read Access	No	Yes	Yes
Wake up of wait point	No	No	Yes
Activation of runnable entity	Yes	Yes	Yes

Table 4.1: Receive mode validity

The category of a runnable entity is not an inherent property but is instead determined by the features of the runnable. Thus the presence of explicit API calls makes the runnable at least category 1B and the presence of a wait point forces the runnable to be category 2.

4.3.1.2.1 Applicability

The different receive modes are not just used for receivers in sender-receiver communication. The same semantics are also applied in the following situations:

- **Success feedback** – The mechanism used to return transmission acknowledgments to a component. See Section 5.2.6.7.
- **Asynchronous client-server result** – The mechanism used to return the result of an asynchronous client-server call to a component. See Section 5.7.5.3.

4.3.1.2.2 Representation in the Software Component Template

The following list serves as a reference for how the RTE Generator determines the Receive Mode from its input [RTE00109]. Note that references to “the DataElementPrototype” within this sub-section will implicitly mean “the DataElementPrototype for which the API is being generated”.

- **All** –
- **“wake up of wait point”** – A *DataReceivePoint* references the *DataElementPrototype* and a *WaitPoint* references the *DataReceivedEvent* which in turn references the *DataElementPrototype*.

- **“activation of runnable entity”** – a *DataReceivedEvent* refences the *DataElementPrototype* and a runnable entity.
- It is an input error if a *WaitPoint* references a *DataReceivedEvent* that references a runnable entity.
- **“implicit data read access”** – A *DataReadAccess* references the *DataElementPrototype* but no *DataReceivedEvent* has to reference the *DataElementPrototype*.
- **“explicit data read access”** – A *DataReceivePoint* references the *DataElementPrototype* but no *DataReceivedEvent* has to reference the *DataElementPrototype*.

  For details of how “implicit data read access” is distinguished from “explicit data read access” see Section 4.3.1.5.

4.3.1.3 Multiple Data Elements

A sender-receiver interface can contain one or more data elements. The transmission and reception of elements is independent – each data element, eg. AUTOSAR signal, can be considered to form a separate logical data channel between the “provide” port and a “require” port.

[rte_sws_6008] Each data element in a sender-receiver interface shall be sent separately [RTE00089].

> **Example 4.2**
>
> Consider an interface that has two data elements, `speed` and `freq` and that a component template defines a provide port that is typed by the interface. The RTE generator will then create two API calls; one to transmit `speed` and another to transmit `freq`.

Where it is important that multiple data elements are sent simultaneously they should be combined into a complex data structure (Section 4.3.1.11.1). The sender then creates an instance of the data structure which is filled with the required data before the RTE is invoked to transmit the data.

4.3.1.3.1 Initial Values

[rte_sws_6009] For each data element in an interface specified with data semantics (isQueued = false), the RTE shall support the `initValue` attribute [RTE00108].

The `initValue` attribute is used to ensure that AUTOSAR software-components always access valid data even if no value has yet been received. This information is required for both inter-ECU and intra-ECU communication. For inter-ECU communication initial values can be handled by COM but for intra-ECU communication RTE has to guarantee that `initValue` is handled.

The specification of an init value is mandatory for each data element prototype with isQueued = FALSE, see [18].

[rte_sws_6010] When isQueued is specified as false, the RTE shall use any specified initial value to prevent the receiver performing calculations based on invalid (i.e. uninitialized) values [RTE00107].

The above requirement ensures that RTE API calls return the initialized value until a “real” value has been received, possibly via the communication service. The requirement does *not* apply when the isQueued attribute is set to true, i.e. when “event” semantics are used since the implied state change when the event data is received will mean that the receiver will not start to process invalid data and would therefore never see the initialized value.

[rte_sws_4500] An initial value cannot be specified when the isQueued attribute is specified as true [RTE00107].

For senders, an initial value is not used directly by the RTE (since an AUTOSAR SW-C must supply a value using `Rte_Send`) however it may be needed to configure the communication service - for example, an un-initialised signal can be transmitted if multiple signals are mapped to a single frame and the communication service transmits the whole frame when any contained signal is sent by the application. Note that it is not the responsibility of the RTE generator to configure the communication service.

It is permitted for an initial value to be specified for either the sender or receiver. In this case the same value is used for both sides of the communication.

[rte_sws_4501] If a sender specifies an initial value and the receiver does not (or *vice versa*) the same initial value is used for both sides of the communication [RTE00108].

It is also permitted for both sender and receiver to specify an initial value. In this case it is defined that the receiver’s initial value is used by the RTE generator for both sides of the communication.

[rte_sws_4502] If both receiver and sender specify an initial value the specification for the *receiver* takes priority [RTE00108].

4.3.1.4 Multiple Receivers and Senders

Sender-receiver communication is not restricted to communication connections between a single sender and a single receiver. Instead, a communication connection can have multiple senders (’n:1’ communication) or multiple receivers (’1:m’ communication).

The RTE does not impose any co-ordination on senders – the behavior of senders is independent of the behavior of other senders. For example, consider two senders `A` and `B` that both transmit data to the same receiver (i.e. ’n:1’ communication). Transmissions by either sender can be made at any time and there is no requirement that the senders co-ordinate their transmission. However, while the RTE does not impose any co-ordination on the senders it does ensure that simultaneous transmissions do not conflict.

In the same way that the RTE does not impose any co-ordination on senders there is no co-ordination imposed on receivers. For example, consider two receivers `P` and `Q` that both receive the same data transmitted by a single sender (i.e. ’1:m’ communication). The RTE does not guarantee that multiple receivers see the data simultaneously even when all receivers are on the same ECU.

4.3.1.5 Implicit and Explicit Data Reception and Transmission

[rte_sws_6011] The RTE shall support ’explicit’ and ’implicit’ data reception and transmission.

Implicit data access transmission means that a runnable does not actively initiate the reception or transmission of data. Instead, the required data is received automatically when the runnable starts and is made available for other runnables when it terminates.

Explicit data reception and transmission means that a runnable employs an explicit API call to send or receive certain data elements. Depending on the category of the runnable and on the configuration of the according ports, these API calls can be either blocking or non-blocking.

4.3.1.5.1 Implicit

DataReadAccess

For the implicit reading of data, called *DataReadAccess* [RTE00128], the data is made available when the runnable starts using the semantics of a copy operation and the RTE ensures that the ’copy’ will not be modified until after the runnable terminates.

When a runnable *R* is started, the RTE reads all data elements marked ’DataReadAccess’ if the data elements may be changed by other runnables a copy is created that will be available to runnable *R*. The runnable *R* can read the data element by using the RTE APIs for implicit read (see the API description in Sect. 5.6.13). That way, the data is guaranteed not to change (e.g. by write operations of other runnables) during the entire lifetime of *R*. If several runnables (even from different components) need the data, they can share the *same* buffer. This is only applicable when the scheduling structure can make sure the contents of the data is protected from modification by any other party.

Note that this concept implies that the runnable does in fact terminate. Therefore, *DataReadAccess* is only allowed for category 1A and 1B runnable entities which are guaranteed to have a finite execution time whereas category 2 runnables may run forever.

DataWriteAccess

Implicit sending, called *DataWriteAccess* [RTE00129], is the opposite concept. Data elements marked as 'DataWriteAccess' are sent by the RTE after the runnable terminates. The runnable can write the data element by using the RTE APIs for implicit write (see the API description in Sect. 5.6.14). The sending is independent from the position in the execution flow in which the `Rte_IWrite` is performed inside the Runnable. When performing several write accesses during runnable execution to the same data element, only the last one will be recognized. Here we have a last-is-best semantic.

[rte_sws_3570] For DataWriteAccess the RTE shall make the send data available to others (other runnables, other AUTOSAR SWCs, Basic SW, ..) with the semantics of a copy [RTE00129].

[rte_sws_3571] For DataWriteAccess the RTE shall make the send data available to others (other runnables, other AUTOSAR SWCs, Basic SW, ..) after the execution of the runnable has terminated [RTE00129].

[rte_sws_3572] For DataWriteAccesses several accesses to the same data element performed inside a runnable during one runnable execution shall lead to only one transmission of the data element [RTE00129].

[rte_sws_3573] If several DataWriteAccesses to the same data element are performed inside a runnable during the runnable execution, the RTE shall use the last value written. (last-is-best semantics) [RTE00129]

DataWriteAccess is only applicable to runnables that are guaranteed to terminate, i.e. category 1A and 1B. It is not allowed – and does not make sense – to use DataWriteAccess for a category 2 runnable which may have infinite execution time.

[rte_sws_3574] DataWriteAccess shall be valid for category 1A and 1B runnable entities [RTE00134].

4.3.1.5.2 Explicit

The behavior of explicit reception depends on the category of the runnable and on the configuration of the according ports.

An explicit API call can be either non-blocking or blocking. If the call is non-blocking (i.e. there is a *DataReceivePoint* referencing the DataElementPrototype for which the API is being generated, but no *WaitPoint* referencing the *DataReceivePoint*), the API call immediately returns the next value to be read and, if the communication is queued (event reception), it removes the data from the receiver-side queue, see Section 4.3.1.10

[rte_sws_6012] A non-blocking RTE API “read” call shall indicate if no data is available [RTE00109].

In contrast, a blocking call (i.e. there is a *WaitPoint* referencing the *DataReceivePoint* for which the API is being generated) will suspend execution of the caller until new data arrives (or a timeout occurs) at the according port. When new data is received, the RTE resumes the execution of the waiting runnable. ([RTE00092])

To prevent infinite waiting, a blocking RTE API call can have a timeout applied. The RTE monitors the timeout and if it expires without data being received returns a particular error status.

[rte_sws_6013] A blocking RTE API "read" call shall indicate the expiry of a timeout [RTE00069].

The "timeout expired" indication also indicates that no data was received before the timeout expired.

Blocking reception of data ("wake up of wait point" receive mode as described in Section 4.3.1.2) is only applicable for category 2 runnables whereas non-blocking reception ("explicit data read access" receive mode) can be employed by runnables of category 2 or 1B. Neither blocking nor non-blocking explicit reception is applicable for category 1A runnable because they must not invoke functions with unknown execution time (see table 4.1).

[rte_sws_6016] The RTE API call for explicit sending (*DataSendPoint*, [RTE00098]) shall be non-blocking.

Using this API call, the runnable can explicitly send new values of the according data element.

[rte_sws_6017] Explicit writing is valid for runnables of category 1b and 2 [RTE00134].

For the same reason stated above, explicit writing is not allowed for a category 1A runnable.

Although the API call for explicit sending is non-blocking, it is possible for a category 2 runnable to block waiting for a notification whether the (explicit) send operation was successful. This is specified by the AcknowledgementRequest attribute and occurs by a separate API call Rte_Feedback. If the feedback method is 'wake_up_of_wait_point', the runnable will block and be resumed by the RTE either when a positive or negative acknowledgement arrives or when the timeout associated with the wait point expires.

4.3.1.5.3 Concepts of data access

Tables 4.2 and 4.3 summarize the characteristics of implicit versus explicit data reception and transmission.

Implicit Read	**Explicit Read**
Receiving of data element values is performed only once when runnable starts	Runnable decides when and how often a data element value is received
Values of data elements do not change while runnable is running.	Runnable can always decide to receive the latest value
Several API calls to the same signal always yield the same data element value	Several API calls to the same signal may yield different data element values

Runnable must terminate (cat. 1A or 1B)	Runnable is of cat. 1B or 2

Table 4.2: Implicit vs. explicit read

Implicit Write	Explicit Write
Sending of data element values is only done once after runnable returns	Runnable can decide when sending of data element values is done via the API call
Several usages of the API call inside the runnable cause only one data element transmission	Several usages of the API call inside the runnable cause several transmissions of the data element content. (Depending on the behavior of COM, the number of API calls and the number of transmissions are not necessarily equal.)
Runnable must terminate (cat. 1A or 1B)	Runnable is cat. 1B or 2

Table 4.3: Implicit vs. explicit write

4.3.1.6 Transmission Acknowledgement

When `AcknowledgementRequest` is specified, the RTE will inform the sending component if the signal has been sent correctly or not. Note that there is no insurance that the signal has actually been *received* correctly by the corresponding receiver AUTOSAR software-component. Thus, only the RTE on the sender side is involved in supporting `AcknowledgementRequest`.

In case of mode switch communication (see Section 4.4), the communication is local to one ECU. The transmission acknowledgement will be sent, when the mode switch is executed by the RTE, see rte_sws_2587.[2]

[rte_sws_5504] The RTE shall support the use of `AcknowledgementRequest` independently for each data item of an AUTOSAR software-component's AUTOSAR interface [RTE00122].

[rte_sws_5506] The RTE generator shall reject specification of the `AcknowledgementRequest` attribute for transmission acknowledgement for 1:n communication [RTE00125], except for mode switch communication. Restriction: In some cases, when more than one receiver is connected via one physical bus, this can not be discovered by the RTE generator.

[2]Currently, no mode switch acknowledgement is defined. If a mode switch acknowledement will be defined in future releases, it shall be used instead of the transmission acknowledgement.

The result of the feedback can be collected using “wake up of wait point”, “explicit data read access” or “activation of runnable entity”.

The AcknowledgementRequest attribute allows to specify a timeout.

[rte_sws_3754] If AcknowledgementRequest is specified, the RTE shall ensure that timeout monitoring is performed, regardless of the receive mode of the acknowledgement.

For inter-ECU communication, AUTOSAR COM provides the necessary functionality, for intra-ECU communication, the RTE has to implement the timeout monitoring.

If a WaitPoint is specified to collect the acknowledgement, two timeout values have to be specified, one for the AcknowledgementRequest and one for the WaitPoint.

[rte_sws_3755] If different timeout values are specified for the AcknowledgementRequest for a DataElementPrototype and for the WaitPoint associated with the DataSendCompletedEvent for the DataSendPoint for that DataElementPrototype, the configuration shall be rejected by the RTE generator.

The DataSendCompletedEvent associated with the DataSendPoint for a DataElementPrototype shall indicate that the transmission was successful or that the transmission was not successful. The status information about the success of the transmission shall be available as the return value of the generated RTE API call.

[rte_sws_3756] For each transmission of a DataElementPrototype only one acknowledgement shall be passed to the sending component by the RTE. The acknowledgement indicates either that the transmission was successful or that the transmission was not successfull.

[rte_sws_3757] The status information about the success or failure of the transmission shall be available as the return value of the RTE API call to retrieve the acknowledgement.

[rte_sws_3758] If the timeout value of the AcknowledgementRequest is 0, no timeout monitoring shall be performed.

4.3.1.7 Communication Time-out

When sender-receiver communication is performed using some physical network there is a chance this communication may fail and the receiver does not get an update of data (in time or at all). To allow the receiver of a `data element` to react appropriately to such a condition the SW-C template allows the specification of a time-out which the infrastructure shall monitor and indicate to the interested software components.

A “data element” is the actual information exchanged in case of sender-receiver communication. In the COM specification this is represented by a `ComSignal`. In the SW-C template a data element is represented by the instance of a `DataElementPrototype`.

[rte_sws_5020] When present, the `aliveTimeout` attribute[3] rte_sws_in_0067 enables the monitoring of the timely reception of the `data element` with data semantics (isQueued = false) transmitted over the network.

The monitoring functionality is provided by the COM module, the RTE transports the event of reception time-outs to software components as “data element outdated”. The software components can either subscribe to that event (activation of runnable entity) or get that situation passed by the implicit and explicit status information (using API calls).

[rte_sws_5021] If `aliveTimeout` is present, but the communication is local to the ECU, time-out monitoring is disabled and no notification of the software components will occur.

Therefore the Software Component shall not rely in its functionality on the time-out notification, because for local communication the notification will never occur. Time-out notification is intended as pure error reporting.

[rte_sws_3759] If the `aliveTimeout` attribute is 0, no timeout monitoring shall be performed.

[rte_sws_5022] If a time-out has been detected, the last correctly received value shall be provided to the software components (preserving the last-is-best-semantics, see Section 4.3.1.10.1).

The time-out support (called “deadline monitoring” in COM) provided by COM has some restrictions which have to be respected when using this mechanism. Since the COM module is configured based on the System Description the restrictions mainly arise from the `data element` to I-PDU mapping. This already has to be considered when developing the System Description and the RTE Generator can only provide warnings when inconsistencies are detected. Therefore the RTE Generator needs to have access to the configuration information of COM.

In case time-out is enabled on a `data element` with update bit, there shall be a separate time-out monitoring for each `data element` with an update bit [COM292].

There shall be an I-PDU based time-out for `data element`s without an update bit [COM290]. For all data elements without update bits within the same I-PDU, the smallest configured time-out of the associated data elements is chosen as time-out for the I-PDU[COM291]. The notification from COM to RTE is performed per data element.

In case one `data element` coming from COM needs to be distributed to several SW-Components the SW-C template allows to specify different `aliveTimeout` values at each Port. But COM does only support one `aliveTimeout` value per `data element`, therefore the smallest `aliveTimeout` value shall be used for the notification of the time-out to several SW-Components.

[3]This attribute is called “LIVELIHOOD” in the VFB specification

4.3.1.8 Data Element Invalidation

The Software Component template allows to specify whether a `data element`, defined in an AUTOSAR Interface, can be invalidated by the sender. The communication infrastructure shall provide means to set a data element to invalid and also indicate an invalid data element to the receiving software components. This functionality is called “data element invalidation”.

[rte_sws_5024] On sender side the `canInvalidate` attribute rte_sws_in_5023 (when present) enables the invalidation support for this `data element`. The actual value used to represent the invalid data element shall be specified in the Data Semantics part of the data element definition defined in rte_sws_in_5031[4].

[rte_sws_5032] On receiver side the definition of an invalid value in rte_sws_in_5031 enables the invalidation support[5].

[rte_sws_5033] Data element invalidation is only supported for data elements with the isQueued attribute set to false rte_sws_in_0045.

The API to set a `data element` to invalid shall be provided to the runnable entities on data element level.

In case an invalidated data element is received a software component can be notified using the activation of runnable entity. If an invalidated data element is read by the SW-C the invalid status shall be indicated in the status code of the API.

4.3.1.8.1 Data Element Invalidation in case of Inter-ECU communication

If `canInvalidate` is enabled and the communication is Inter-ECU:

- explicit data transmission: data element invalidation will be performed by COM (COM needs to be configured properly).
- implicit data transmission: data element invalidation will be performed by RTE.

[rte_sws_5026] If a data element has been invalidated in case of Inter-ECU communication the query of the value shall return the value provided by COM together with an indication of the invalid case.

If the COM layer does perform an “invalid value substitution” (`Com_DataInvalidAction` is `Replace` [COM314]) the RTE shall not consider the substitution and provide the value returned by COM. If no substitution takes place the invalid value will be provided.

[4]When canInvalidate is enabled but there is no invalidValue specified it is considered an invalid configuration.

[5]The receiver side RTE generation tool might only have access to the ECU Extract of the receiving ECU, therefore can not determine the canInvalidate attribute of the sender

4.3.1.8.2 Data Element Invalidation in case of Intra-ECU communication

[rte_sws_5025] If `canInvalidate` is enabled, and the communication is Intra-ECU, data element invalidation can be implemented by the RTE or the RTE may utilize the implementation of the AUTOSAR COM module.

In case of implicit data transmission the RTE shall always implement the data element invalidation and therefore provide an API to set the data element's value to the invalid value. The actual invalid value is specified in the SW-C template rte_sws_in_5031.

[rte_sws_5030] If a data element has been invalidated in case of Intra-ECU communication the query of the value shall return the same value as if COM would have handled the invalidation (copy COM behavior).

The RTE does not support the "invalid value substitution" in case of local communication[6], therefore the query must always return the invalid value.

4.3.1.9 Filters

By means of the `filter` attribute [RTE00121] an additional filter layer can be added on the receiver side. *Value-based* filters can be defined, i.e. only signal values fulfilling certain conditions are made available for the receiving component. The possible filter algorithms are taken from OSEK COM version 3.0.2. They are listed in the meta model (see [18], Sect. 'Communication specification of data filters'). According to the SW-C template [18], filters are only allowed for signals that are compatible to C language unsigned integer types (i.e. characters, unsigned integers and enumerations). Thus, filters cannot be applied to complex data types like records or arrays.

[rte_sws_5503] The RTE shall provide value-based filters on the receiver-side as specified in the SW-C template [18], Section 'Communication specification of data filters'.

[rte_sws_5500] For inter-ECU communication, the RTE shall use the filter implementation of the COM layer [RTE00121]. For intra-ECU communication, the RTE can use the filter implementation of COM, but may also implement the filters itself for efficiency reasons, without using COM.

[rte_sws_5501] The RTE shall support a different filter specification for each data element in a component's AUTOSAR interface [RTE00121].

4.3.1.10 Buffering

[rte_sws_2515] The buffering of sender-receiver communication shall be done on the receiver side. This does not imply that COM does no buffering on the sender side. On the receiver side, two different approaches are taken for the buffering of 'data' and of 'events', depending on the value of the isQueued attribute of the data element.

[6]The reason for this restriction is that currently only the configuration of the COM module allows for this substitution. Therefore the input information for local data element passing is missing to configure the RTE

4.3.1.10.1 Last-is-Best-Semantics for ‘data’ Reception

[rte_sws_2516] On the receiver side, the buffering of ‘data’ (isQueued = false) shall be realized by the RTE by a single data set for each data element instance.

The use of a single data set provides the required semantics of a single element queue with overwrite semantics (new data replaces old). Since the RTE is required to ensure data consistency, the generated RTE should ensure that non-atomic reads and writes of the data set (e.g. for complex data) are protected from conflicting concurrent access. RTE may use lower layers like COM to implement the buffer.

[rte_sws_2517] Depending on the ports attributes, the RTE shall initialize this data set with a startup value.

[rte_sws_2518] Implicit or explicit read access shall always return the last received data.

Requirement rte_sws_2518 applies whether or not there is a DataReceivedEvent referencing the DataElementPrototype for which the API is being generated.

[rte_sws_2519] Explicit read access shall be non blocking in the sense that it does not wait for new data to arrive. The RTE shall provide mutual exclusion of read and write accesses to this data, e.g., by ExclusiveAreas.

[rte_sws_2520] When new data is received, the RTE shall silently discard the previous value of the data, regardless of whether it was read or not.

4.3.1.10.2 Queueing for ‘event’ Reception

The application of event semantics implies a state change. Events usually have to be handled. In many cases, a loss of events can not be tolerated. Hence the isQueued attribute is set to true to indicate that the received ‘events’ have to be buffered in a queue.

[rte_sws_2521] The RTE shall implement a receive queue for each event-like data element (isQueued = true) of a receive port.

The queueLength attribute of the EventReceiverComSpec referencing the event assigns a constant length to the receive queue.

[rte_sws_2522] The events shall be written to the end of the queue and read (consuming) from the front of the queue (i.e. the queue is first-in-first-out).

[rte_sws_2523] If a new event is received when the queue is already filled, the RTE shall discard the received event and set an error flag.

[rte_sws_2524] The error flag shall be reset during the next explicit read access on the queue. In this case, the status value RTE_E_LOST_DATA shall be presented to the application together with the data.

[rte_sws_2525] If an empty queue is polled, the RTE shall return with a status RTE_E_NO_DATA to the polling function, (see chap. 5.5.1).

The minimum size of the queue is 1.

[rte_sws_2526] The RTE generator shall reject a queueLength attribute of an EventReceiverComSpec with a queue length $\leqslant 0$.

4.3.1.11 Operation

4.3.1.11.1 Inter-ECU Mapping

This section describes the mapping from DataElementPrototypes to COM signals or COM signal groups for sender-receiver communication. The mapping is described in the input of the RTE generator, in the DataMappings element of the ECU configuration.

If a DataElementPrototype is mapped to a COM signal or COM signal group but the communication is local, the RTE generator can use the COM signal/COM signal group for the transmission or it can use its own implementation for the transmission.

4.3.1.11.1.1 Primitive Data Types

[rte_sws_4504] If a data element is a primitive type and the communication is inter-ECU, the DataMappings element shall contain a mapping of the data element to at least one COM signal, else the configuration shall be rejected.

The mapping defines all aspects of the signal necessary to configure the communication service, for example, the network signal endianess and the communication bus. The RTE generator only requires the COM signal handle id since this is necessary for invoking the COM API.

[rte_sws_4505] The RTE shall use the handle id of a COM signal when invoking the COM API.

4.3.1.11.1.2 Complex Data Types

When a data element is a complex type the RTE is required to perform more complex actions to marshall the data [RTE00091] than is the case for primitive data types.

The DataMappings element of the ECU configuration contains (or reference) sufficient information to allow the data item or operation parameters to be transmitted. The mapping indicates the COM signals or signal groups to be used when transmitting a given data item of a given port of a given software component instance within the composition.

[rte_sws_4506] If a data element is a complex type and the communication is inter-ECU, the DataMappings element shall contain a mapping of the data element to COM signals such that each element of the complex data type that is a primitive data type is mapped to a separate COM signal(s), else the configuration shall be rejected.

[rte_sws_4507] If a data element is a complex type and the communication is inter-ECU, the DataMappings element shall contain a mapping of the data element to COM signals such that each element of the complex data type that is itself a complex data type shall be recursively mapped to a primitive type and hence to a separate COM signal(s).

The above requirements have two key features; firstly, COM is responsible for endianness conversion (if any is required) of primitive types and, secondly, differing structure member alignment between sender and receiver is irrelevant since the COM signals are packed into I-PDUs by the COM configuration.

The DataMappings shall contain sufficient COM signals to map each primitive element[7] of the AUTOSAR signal.

[rte_sws_4508] If a data element is a complex type and the communication is inter-ECU, the DataMappings element shall contain at least one COM signal for each primitive element of the AUTOSAR signal.

[rte_sws_2557] If a data element is a complex type and the communication is inter-ECU, the DataMappings element shall contain a mapping to at least one signal group for each non-primitive data element prototype of sender receiver communication.

4.3.1.11.1.3 Mapping Algorithm

As described above, when the data item is a complex data type, multiple system signals are required - which must all be within the same signal group to ensure atomicity (see Section 4.3.1.11.2). The meta-model permits multiple sytem signals to be specified, but there is no explicit linking of a particular signal to a particular data element. Consequently, the following mapping algorithm has been chosen:

[rte_sws_6025] The RTE generator shall match signals in the input with elements in the complex type by performing a depth first traversal of the data structure and using the signals in the order that they are given in the input.

Example 4.3

Given the following data type:

```
struct complexType {
   uint16   a;
   uint8    b[4];
   struct {
     uint16 c;
     uint16 d;
   } substruct;
   uint32   e;
};
```

[7] An AUTOSAR signal that is a primitive data type contains exactly one one primitive element whereas a signal that is a complex type contains one or more primitive elements.

There would need to be eight sytem signals referenced in the input. The system signal would be used in the following order:

```
complexType.a,
complexType.b[0]
complexType.b[1]
complexType.b[2]
complexType.b[3]
complexType.substruct.c
complexType.substruct.d
complexType.e
```

4.3.1.11.2 Atomicity

[rte_sws_4527] The RTE is required to treat AUTOSAR signals transmitted using sender-receiver communication atomically [RTE00073]. To achieve this the “signal group” mechanisms provided by COM shall be utilized. See rte_sws_2557 for the mapping.

The RTE decomposes the complex data type into single signals as described above and passes them to the COM module by using the COM API call `Com_UpdateShadowSignal`. As this set of single signals has to be treated as atomic, it is placed in a “signal group”. A signal group has to be placed always in a single I-PDU. Thus, atomicity is established. When all signals have been updated, the RTE initiates transmission of the signal group by using the COM API call `Com_SendSignalGroup`.

As would be expected, the receiver side is the exact reverse of the transmission side: the RTE must first call `Com_ReceiveSignalGroup` precisely once for the signal group and then call `Com_ReceiveShadowSignal` to extract the value of each signal within the signal group.

A signal group has the additional property that COM guarantees to inform the receiver by invoking a call-back about its arrival only after all signals belonging to the signal group have been unpacked into a shadow buffer.

4.3.1.11.3 Fan-out

Fan-out can be divided into two scenarios; “PDU fanout” where the same I-PDU is sent to multiple destinations and “signal fan-out” where the same signal, i.e. data element is sent in different I-PDUs to multiple receivers.

For Inter-ECU communication, the RTE does not perform PDU fan-out. Instead, the RTE invokes `Com_SendSignal` once for a primitive data element and expects the fan-out to multiple destinations to occur lower down in the AUTOSAR communication stack. However, it is necessary for the RTE to support “signal fan-out” since this cannot be performed by any lower level layer of the AUTOSAR communication stack.

[rte_sws_6023] For inter-ECU transmission of a primitive data type, the RTE shall invoke `Com_SendSignal` for each COM signal to which the primitive data element is mapped.

If the data element is a complex data type, RTE invokes `Com_UpdateShadowSignal` for each primitive element in the complex data type and each COM signal to which that primitive element is mapped, and `Com_SendSignalGroup` for each COM signal group to which the data element is mapped.

[rte_sws_4526] For inter-ECU transmission of complex data type, the RTE shall invoke `Com_UpdateShadowSignal` for each COM signal to which an element in the complex data type is mapped and `Com_SendSignalGroup` for each COM signal group to which the complex data element is mapped.

For intra-ECU transmission of data elements, the situation is slightly different; the RTE handles the communication (the lower layers of the AUTOSAR communication stack are not used) and therefore must ensure that the data elements are routed to all receivers.

[rte_sws_6024] For intra-ECU transmission of data elements, the RTE shall perform the fan-out to each receiver [RTE00028].

4.3.1.11.4 Fan-in

When receiving data from multiple senders in inter-ECU communication, either the RTE on the receiver side has to collect data received in different COM signals or COM signal groups and pass it to one receiver or the RTE on the sender side has to provide shared access to a COM signal or COM signal group to multiple senders. The receiver RTE, which has to handle multiple COM signals or signal groups, is notified about incoming data for each COM signal or COM signal group separately but has to ensure data consistency when passing the data to the receiver. The sender RTE, which has to handle multiple senders sharing COM signals or signal groups, has to ensure consistent access to the COM API, since COM API calls for the same signal are not reentrant.

[rte_sws_3760] If multiple senders use different COM signals or signal groups for inter-ECU transmission of a data element prototype with isQueued = false to a receiver, the RTE on the receiver side has to pass the last received value to the receiver component while ensuring data consistency.

[rte_sws_3761] If multiple senders use different COM signals or signal groups for inter-ECU transmission of a data element prototype with isQueued = true to a receiver, the RTE on the receiver side has to queue all incoming values while ensuring data consistency.

[rte_sws_3762] If multiple senders share COM signals or signal groups for inter-ECU transmission of a data element prototype to a receiver, the RTE on the sender side has to ensure that the COM API for those signals is not invoked concurrently.

For intra-ECU transmission, the RTE must handle the fan-in, which is already stated in rte_sws_6024.

4.3.1.11.5 Sequence diagrams of Sender Receiver communication

Figure 4.15 shows a sequence diagram of how Sender Receiver communication for data transmission and non-blocking reception may be implemented by RTE. The sequence diagram also shows the `Rte_Read` API behavior if an `initValue` is specified.

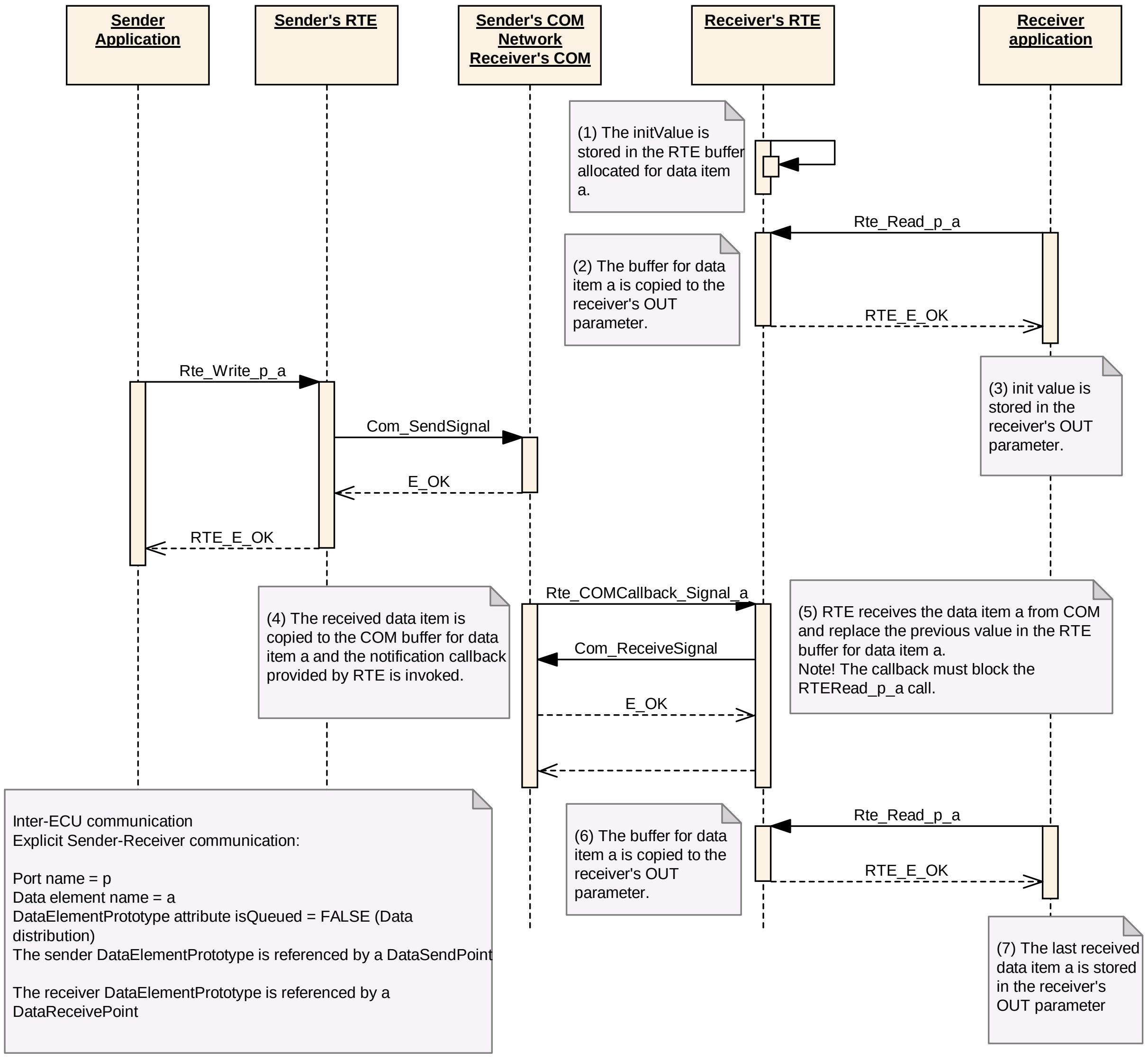

Sender Application
Sender's RTE
Sender's COM Network Receiver's COM
Receiver's RTE
Receiver application
(1) The initValue is stored in the RTE buffer allocated for data item a.
Rte_Read_p_a
(2) The buffer for data item a is copied to the receiver's OUT parameter.
RTE_E_OK
Rte_Write_p_a
(3) init value is stored in the receiver's OUT parameter.
Com_SendSignal
E_OK
RTE_E_OK
Rte_COMCallback_Signal_a
(4) The received data item is copied to the COM buffer for data item a and the notification callback provided by RTE is invoked.
Com_ReceiveSignal
(5) RTE receives the data item a from COM and replace the previous value in the RTE buffer for data item a.
Note! The callback must block the RTERead_p_a call.
E_OK
Inter-ECU communication
Explicit Sender-Receiver communication:
Port name = p
Data element name = a
DataElementPrototype attribute isQueued = FALSE (Data distribution)
The sender DataElementPrototype is referenced by a DataSendPoint
The receiver DataElementPrototype is referenced by a DataReceivePoint
(6) The buffer for data item a is copied to the receiver's OUT parameter.
Rte_Read_p_a
RTE_E_OK
(7) The last received data item a is stored in the receiver's OUT parameter

Figure 4.15: Sender Receiver communication with isQueued false and DataReceivePoint as reception mechanism

Figure 4.16 shows a sequence diagram of how Sender Receiver communication for event transmission and non-blocking reception may be implemented by RTE. The sequence diagram shows the `Rte_Receive` API behavior when the queue is empty.

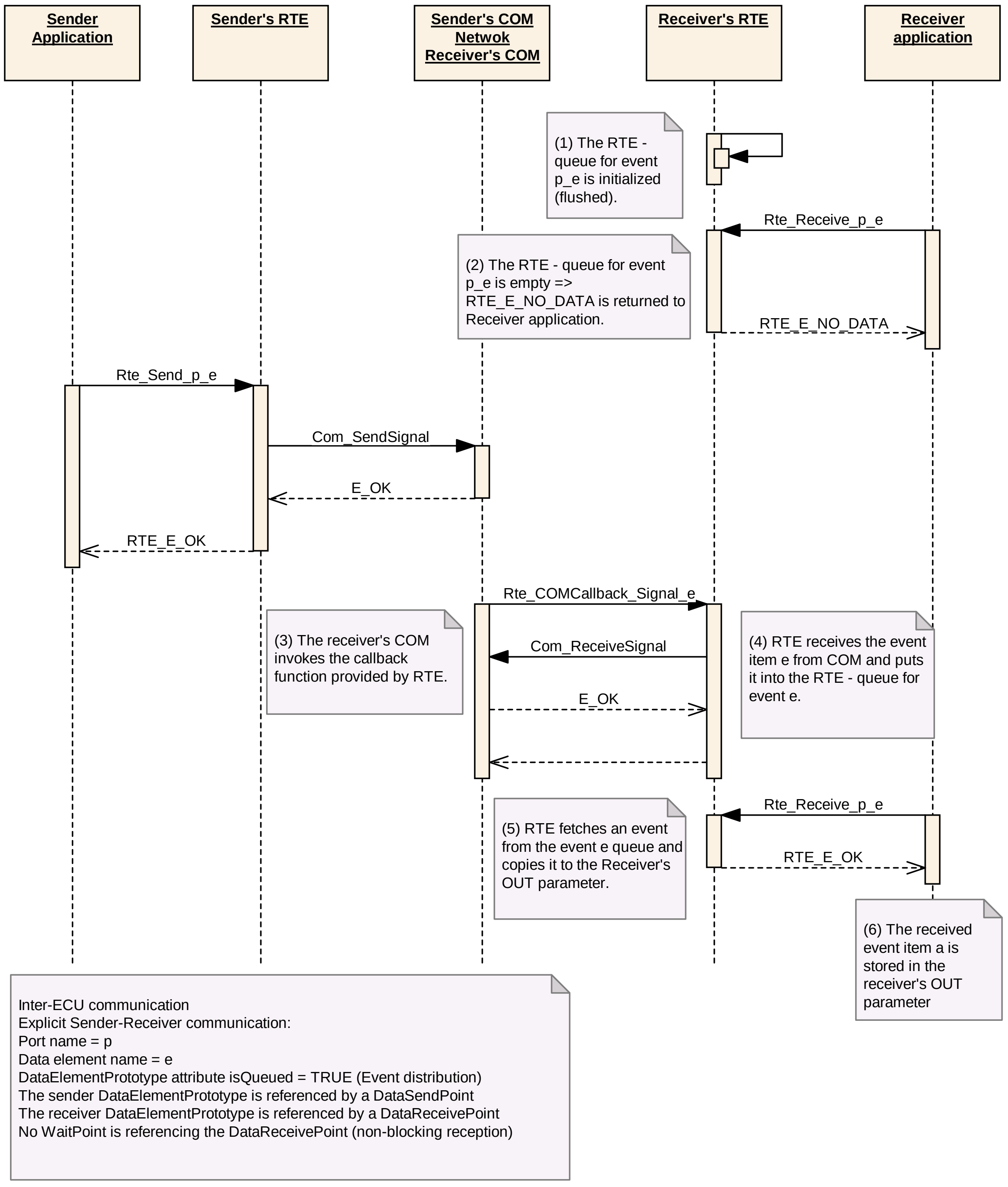

Sender Application
Sender's RTE
Sender's COM Netwok Receiver's COM
Receiver's RTE
Receiver application
(1) The RTE - queue for event p_e is initialized (flushed).
Rte_Receive_p_e
(2) The RTE - queue for event p_e is empty => RTE_E_NO_DATA is returned to Receiver application.
RTE_E_NO_DATA
Rte_Send_p_e
Com_SendSignal
E_OK
RTE_E_OK
Rte_COMCallback_Signal_e
(3) The receiver's COM invokes the callback function provided by RTE.
Com_ReceiveSignal
E_OK
(4) RTE receives the event item e from COM and puts it into the RTE - queue for event e.
Rte_Receive_p_e
RTE_E_OK
(5) RTE fetches an event from the event e queue and copies it to the Receiver's OUT parameter.
(6) The received event item a is stored in the receiver's OUT parameter
Inter-ECU communication
Explicit Sender-Receiver communication:
Port name = p
Data element name = e
DataElementPrototype attribute isQueued = TRUE (Event distribution)
The sender DataElementPrototype is referenced by a DataSendPoint
The receiver DataElementPrototype is referenced by a DataReceivePoint
No WaitPoint is referencing the DataReceivePoint (non-blocking reception)

Figure 4.16: Sender Receiver communication with isQueued true and DataReceivePoint as reception mechanism

Figure 4.17 shows a sequence diagram of how Sender Receiver communication for event transmission and activation of runnable entity on the receiver side may be implemented by RTE.

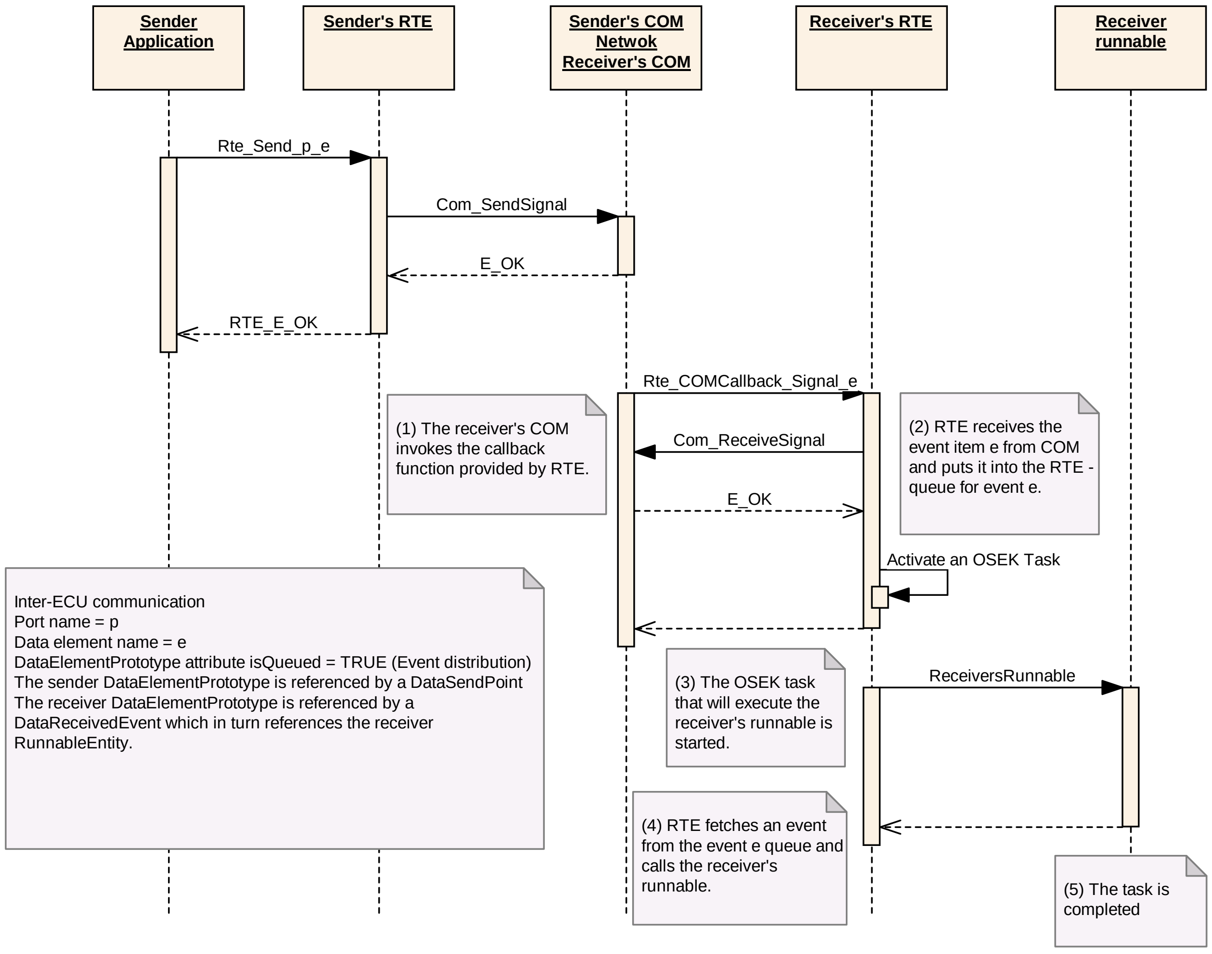

Sender Application
Sender's RTE
Sender's COM Netwok Receiver's COM
Receiver's RTE
Receiver runnable
Rte_Send_p_e
Com_SendSignal
E_OK
RTE_E_OK
Rte_COMCallback_Signal_e
(1) The receiver's COM invokes the callback function provided by RTE.
Com_ReceiveSignal
E_OK
(2) RTE receives the event item e from COM and puts it into the RTE - queue for event e.
Activate an OSEK Task
Inter-ECU communication
Port name = p
Data element name = e
DataElementPrototype attribute isQueued = TRUE (Event distribution)
The sender DataElementPrototype is referenced by a DataSendPoint
The receiver DataElementPrototype is referenced by a DataReceivedEvent which in turn references the receiver RunnableEntity.
(3) The OSEK task that will execute the receiver's runnable is started.
ReceiversRunnable
(4) RTE fetches an event from the event e queue and calls the receiver's runnable.
(5) The task is completed

Figure 4.17: Sender Receiver communication with isQueued true and activation of runnable entity as reception mechanism

4.3.2 Client-Server

4.3.2.1 Introduction

Client-server communication involves two entities, the client which is the requirer (or user) of a service and the server that provides the service.

The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server, in the form of the RTE, waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client's request. So, the direction of initiation is used to categorize whether a AUTOSAR software-component is a client or a server.

A single component can be both a client and a server depending on the software realization.

The invocation of a server is performed by the RTE itself when a request is made by a client. The invocation occurs synchronously with respect to the RTE (typically via a function call) however the client's invocation can be either synchronous (wait for server to complete) or asynchronous with respect to the server.

[rte_sws_6019] The only mechanism through which a server can be invoked is through a client-server invocation request from a client [RTE00029].

The above requirement means that *direct invocation* of the function implementing the server outside the scope of the RTE is not permitted.

A server has a dedicated provide port and a client has a dedicated require port. To be able to connect a client and a server, both ports must be categorized by the same interface.

The client can be blocked (synchronous communication) respectively non-blocked (asynchronous communication) after the service request is initiated until the response of the server is received.

A server implemented by a RunnableEntity with attribute canBeInvokedConcurrently set to FALSE is not allowed to be invoked concurrently and since a server can have one or more clients the server may have to handle concurrent service calls (n:1 communication) the RTE must ensure that concurrent calls do not interfere.

[rte_sws_4515] It is the responsibility of the RTE to ensure that serialization[8] of the operation is enforced when the server runnable attribute canBeInvokedConcurrently is FALSE. Note that the same server may be called using both synchronous and asynchronous communication [RTE00033].

[8]Serialization ensures at most one thread of control is executing an instance of a runnable entity at any one time. An AUTOSAR software-component can have multiple instances (and therefore a runnable entity can also have multiple instances). Each instance represents a different server and can be executed in parallel by different threads of control thus serialization only applies to an individual instance of a runnable entity – multiple runnable entities within the same component instance may also be executed in parallel.

[rte_sws_4516] The RTE's implementation of the client-server communication has to ensure that a service result is dispatched to the correct client if more than one client uses a service [RTE00080].

The result of the client/server operation can be collected using "wake up of wait point", "explicit data read access" or "activation of runnable entity".

If the client and server are executing in the same ECU, i.e. intra ECU Client-Server communication, the RTE API call for client-server communication (see Sect. 5.6.9) can be optimized to a direct function call of the client without any interaction with the RTE or the communication service. Since the communication occurs conceptually via the RTE (it is initiated via an RTE API call) the optimization does not violate the requirement that servers are only invoked via client-server requests.

4.3.2.2 Multiplicity

Client-server interfaces contain two dimensions of multiplicity; multiple clients invoking a single server and multiple operations within a client-server interface.

4.3.2.2.1 Multiple Clients Single Server

Client-server communication involves an AUTOSAR software-component invoking a defined "server" operation in another AUTOSAR software-component which may or may not return a reply.

[rte_sws_4519] The RTE shall support multiple clients invoking the same server operation ('n:1' communication where $n \geqslant 1$). [RTE00029]

4.3.2.2.2 Multiple operations

A client-server interface contains one or more operations. A port of a AUTOSAR software-component that *requires* an AUTOSAR client-server interface to the component can independently invoke any of the operations defined in the interface [RTE00089].

[rte_sws_4517] The RTE API shall support independent access to operations in a client-server interface [RTE00029].

Example 4.4

Consider a client-server interface that has two operations, `op1` and `op2` and that an AUTOSAR software-component definition requires a port typed by the interface. As a result, the RTE generator will create two API calls; one to invoke `op1` and another to invoke `op2`. The calls can invoke the server operations either synchronously or asynchronously depending on the configuration.

Recall that each data element in a sender-receiver interface is transmitted independently (see Section 4.3.1.3) and that the coherent transmission of multiple data items is achieved through combining multiple items into a single complex data type. The transmission of the parameters of an operation in a client-server interface is similar to a record since the RTE guarantees that all parameters are handled atomically [RTE00073].

[rte_sws_4518] The RTE shall treat the parameters (and results) of a client-server operation atomically [RTE00033].

However, unlike a sender-receiver interface, there is no facility to combine multiple client-server operations so that they are invoked as a group.

4.3.2.2.3 Single Client Multiple Server

The RTE is *not* required to support multiple server operations invoked by a single client component request ('1:n' communication where n $>$ 1).

4.3.2.2.4 Serialization

Each client can invoke the server simultaneously and therefore the RTE is required to support multiple requests of servers. If the server requires serialization, the RTE has to ensure it.

[rte_sws_4520] The RTE shall support simultaneous invocation requests of a server operation. [RTE00080]

[rte_sws_4522] The RTE shall ensure that the runnable entity implementing a server operation has completed the processing of a request before it begins processing the next request, if serialization is required by the server operation, i.e canBeInvokedConcurrently attribute set to FALSE [RTE00033].

When this requirement is met the operation is said to be “serialized”. A serialized server only accepts and processes requests atomically and thus avoids the potential for conflicting concurrent access.

Client requests that cannot be serviced immediately due to a server operation being “busy” are required to be queued pending processing. The presence and depth of the queue is configurable.

If the runnable entity implementing the server operation is reentrant , i.e. canBeInvokedConcurrently attribute set to TRUE, no serialization is necessary. This allows to implement invocations of reentrant server operations as direct function calls without involving the RTE.

4.3.2.3 Communication Time-out

The ServerCallPoint allows to specify a timeout so that the client can be notified that the server is not responding and can react accordingly. If the client invokes the server synchronously, the RTE API call to invoke the server reports the timeout. If the client invokes the server asynchronously, the timeout notification is passed to the client by the RTE as a return value of the API call that collects the result of the server operation.

[rte_sws_3763] The RTE shall ensure that timeout monitoring is performed for client-server communication, regardless of the receive mode for the result.

If the server is invoked asynchronously and a WaitPoint is specified to collect the result, two timeout values have to be specified, one for the ServerCallPoint and one for the WaitPoint.

[rte_sws_3764] If different timeout values are specified for the AsynchronousServer-CallPoint and for the WaitPoint associated with the AsynchronousServerCallReturn-sEvent for this AsynchronousServerCallPoint, the configuration shall be rejected by the RTE generator.

In asynchronous client-server communication the AsynchronousServerCallReturnsEv-ent associated with the AsynchronousServerCallPoint for an OperationPrototype shall indicate that the server communication is finished or that a timeout occurred. The status information about the success of the server operation shall be available as the return value of the RTE API call generated to collect the result.

[rte_sws_3765] For each asynchronous invocation of an operation prototype only one AsynchronousServerCallReturnsEvent shall be passed to the client component by the RTE. The AsynchronousServerCallReturnsEvent shall indicate either that the trans-mission was successful or that the transmission was not successfull.

[rte_sws_3766] The status information about the success or failure of the asynchro-nous server invocation shall be available as the return value of the RTE API call to retrieve the result.

After a timeout was detected, no result shall be passed to the client.

[rte_sws_3770] If a timeout was detected by the RTE, no result shall be passed back to the client.

Since an asynchronous client can have only one outstanding server invocation at a time, the RTE has to monitor when the server can be safely invoked again. In normal operation, the server can be invoked again when the result of the previous invocation was collected by the client.

[rte_sws_3773] If a server is invoked asynchronously and no timeout occurred, the RTE shall ensure that the server can be invoked again by the same client, after the result was successfully passed to the client.

In intra-ECU client-server communication, the RTE can determine whether the server runnable is still running or not.

[rte_sws_3771] If a timeout was detected in asynchronous intra-ECU client-server communication, the RTE shall ensure that the server is not invoked again by the same client until the server runnable has terminated.

In inter-ECU communication, the client RTE has no knowledge about the actual status of the server. The response of the server could have been lost because of a communication error or because the server itself did not respond. Since the client-side RTE cannot distinguish the two cases, the client must be able to invoke the server again after a timeout expired.

[rte_sws_3772] If a timeout was detected in asynchronous inter-ECU client-server communication, the RTE shall ensure that the server can be invoked again by the same client after the timeout notification was passed to the client.

Note that this might lead to client and server running out of sync, i.e. the response of the server belongs to the previous, timed-out invocation of the client. The application has to handle the synchronization of client and server after a timeout occurred.

[rte_sws_3767] If the timeout value of the ServerCallPoint is 0, no timeout monitoring shall be performed.

If the canBeInvokedConcurrently attribute of the server runnable is set to TRUE, no timeout monitoring has to be performed to allow the optimization of the RTE API call to invoke the server to a direct function call.

[rte_sws_3768] If the canBeInvokedConcurrently attribute of the server runnable is set to TRUE, no timeout monitoring shall be performed if the RTE API call to invoke the server is implemented as a direct function call.

4.3.2.4 Port-Defined argument values

Port-defined argument values exist in order to support interaction between Application Software Components and Basic Software Modules.

Several Basic Software Modules use an integer identifier to represent an object that should be acted upon. For instance, the NVRAM Manager uses an integer identifier to represent the NVRAM block to access. This identifier is not known to the client, as the client must be location independent, and the NVRAM block to access for a given application software component cannot be identified until components have been mapped onto ECUs.

There is therefore a mismatch between the information available to the client and that required by the server. Port-defined argument values bridge that gap.

The required port-defined arguments (the fact that they are required, their data type and their values) are specified within the input to the RTE generator. (See requirements rte_sws_in_1361 and rte_sws_in_1362.)

[rte_sws_1360] When invoking the runnable entity specified for an OperationInvokedEvent, the RTE must include the port-defined argument values between the instance handle (if it is included) and the operation-specific parameters, in the order they are given in the template.

Requirement rte_sws_1360 means that a client will make a request for an operation on a require (Client-Server) port including only its instance handle (if required) and the explicit operation parameters, yet the server will be passed the implicit parameters as it requires.

Note that the values of implicit parameters are constant for a particular server runnable entity; it is therefore expected that using port-defined argument values imposes no RAM overhead (beyond any extra stack required to store the additional parameters).

Since the instance-specific values for the port-defined argument values are not known for the contract phase, the RTE has to use standardized symbolic names for the port-defined arguments for the contract phase API mapping. The port-defined argument values are then included in the RTE generation phase by defining constants with the symbolic name of the contract phase and the value available for the RTE generation phase. See Sections 5.3.3.3.2 and 5.3.7.3.6 for the details.

4.3.2.5 Buffering

Client-Server-Communication is a two-way-communication. A request is sent from the client to the server and a response is sent back.

The RTE shall store or buffer the communication on the corresponding receiving sides, requests on server side and responses on client side, respectively:

- **[rte_sws_2527]** The RTE shall buffer a request on the server side in a first-in-first-out queue as described in chapter 4.3.1.10.2 for ‘events’.
- **[rte_sws_2528]** The RTE shall also keep the response on the client side with event-semantics, but always with queue length 1.

For the server side, the attribute queueLength of the ServerComSpec specifies the length of the queue.

[rte_sws_2529] The RTE shall reject a queue of length $\leqslant 0$.

[rte_sws_2530] The RTE shall use the queue of requests to serialise access to a server.

A buffer overflow of the server is not reported to the client. The client will receive a time out.

4.3.2.6 Operation

4.3.2.6.1 Inter-ECU Mapping

[rte_sws_6028] The arguments of an operation shall be mapped to two dedicated composite data items; one for the arguments and one for the return values (the application error value, "OUT" and "INOUT" arguments).

[rte_sws_6029] If (and only if) the interface lists any application errors, the composite data item for the return values shall contain an additional `Rte_StatusType` as the first element.

[rte_sws_6027] The order of the arguments within the composite data item shall follow the order of arguments in the configuration.

The basic mapping of the composite data items is performed in the same way as sender-receiver signals are mapped to COM signals and signal groups, see Section 4.3.1.11.1. However, there is one important difference; the return values form a return path from server to client and therefore the RTE on the server side ECU must be able to identify from which client a request originates. In case of multiple clients on one ECU with a server on another ECU, this requires additional rules for the identification.

Limitation because of missing Client-Server communication protocol
For Inter-ECU Client-Server communication the Client-Server call needs at least one in and one out parameter. This restriction is stated in Appendix A.8

This means Client-Server calls with less than two parameters between two ECUs are not possible yet.
This limitation is **not** valid for Intra-ECU communication!

4.3.2.6.2 Client Identity

The RTE uses the following mechanism to implement client identity:

[rte_sws_2579] In case of a server on one ECU with multiple clients on other ECUs, the client server communication shall use different unique COM signals and signal groups for each client to allow the identification of the client associated with each system signal.

With this mechanism, the server-side RTE must handle the fan-in. This is done in the same way as for sender-receiver communication.

[rte_sws_3769] If multiple clients have access to one server, the RTE on the server side has to queue all incoming server invocations while ensuring data consistency.

4.3.2.6.3 Atomicity

The requirements for atomicity from Section 4.3.1.11.2 also apply for the composit data types described in Section 4.3.2.6.1.

4.3.2.6.4 Fault detection and reporting

Client Server communication may encounter interruption like:

- Buffer overflow at the server side.
- Communication interruption.
- Server might be inaccessible for some reason.

The client specifies a timeout that will expire in case the server or communication fails to complete within the specified time. The reporting method of an expired timeout depends on the communication attributes:

- If the C/S communication is synchronous the RTE returns `RTE_E_TIMEOUT` on the `Rte_Call` function (see chapter 5.6.9).
- If the C/S communication is asynchronous the RTE returns `RTE_E_TIMEOUT` on the `Rte_Result` function (see chapter 5.6.10).

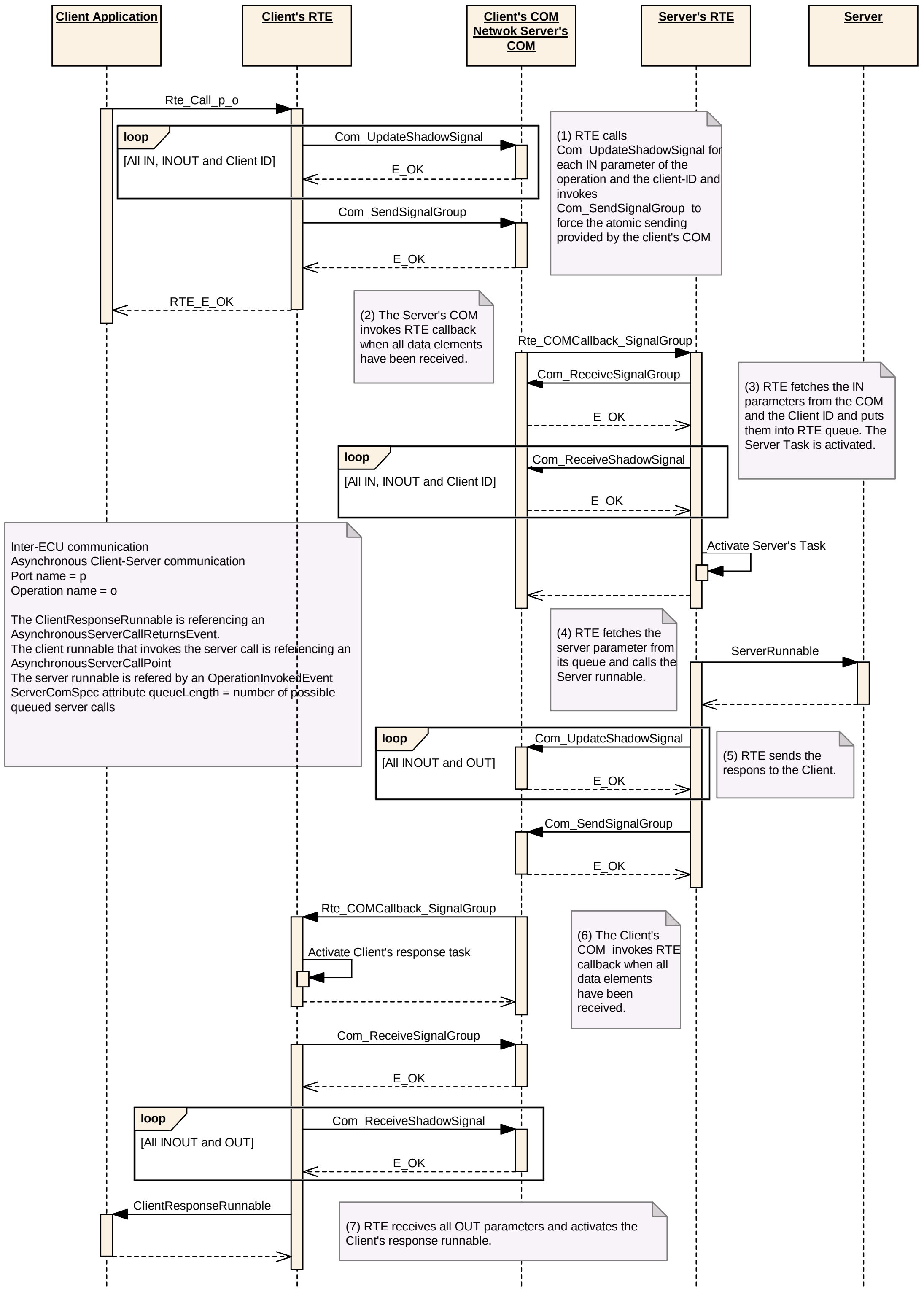
In the case that RTE detects a communication fault when forwarding signals to COM, the RTE returns `RTE_E_COMMS_ERROR` on the `Rte_Call` (see chapter 5.6.9).

If the client still has an outstanding server invocation when the server is invoked again, the `RTE\_E\_TIMEOUT` on the `Rte_Call` (see chapter 5.6.9).

In the absence of structural errors, application errors will be reported if present.

4.3.2.6.5 Asynchronous Client Server communication

Figure 4.18 shows a sequence diagram of how asynchronous client server communication may be implemented by RTE.



(1) RTE calls Com_UpdateShadowSignal for each IN parameter of the operation and the client-ID and invokes Com_SendSignalGroup to force the atomic sending provided by the client's COM

(2) The Server's COM invokes RTE callback when all data elements have been received.

(3) RTE fetches the IN parameters from the COM and the Client ID and puts them into RTE queue. The Server Task is activated.

Inter-ECU communication
Asynchronous Client-Server communication
Port name = p
Operation name = o

The ClientResponseRunnable is referencing an AsynchronousServerCallReturnsEvent.
The client runnable that invokes the server call is referencing an AsynchronousServerCallPoint
The server runnable is refered by an OperationInvokedEvent
ServerComSpec attribute queueLength = number of possible queued server calls

(4) RTE fetches the server parameter from its queue and calls the Server runnable.

(5) RTE sends the respons to the Client.

(6) The Client's COM invokes RTE callback when all data elements have been received.

(7) RTE receives all OUT parameters and activates the Client's response runnable.

Figure 4.18: Client Server asynchronous

4.3.2.6.6 Synchronous Client Server communication

Figure 4.19 shows a sequence diagram of how synchronous client server communication may be implemented by RTE.

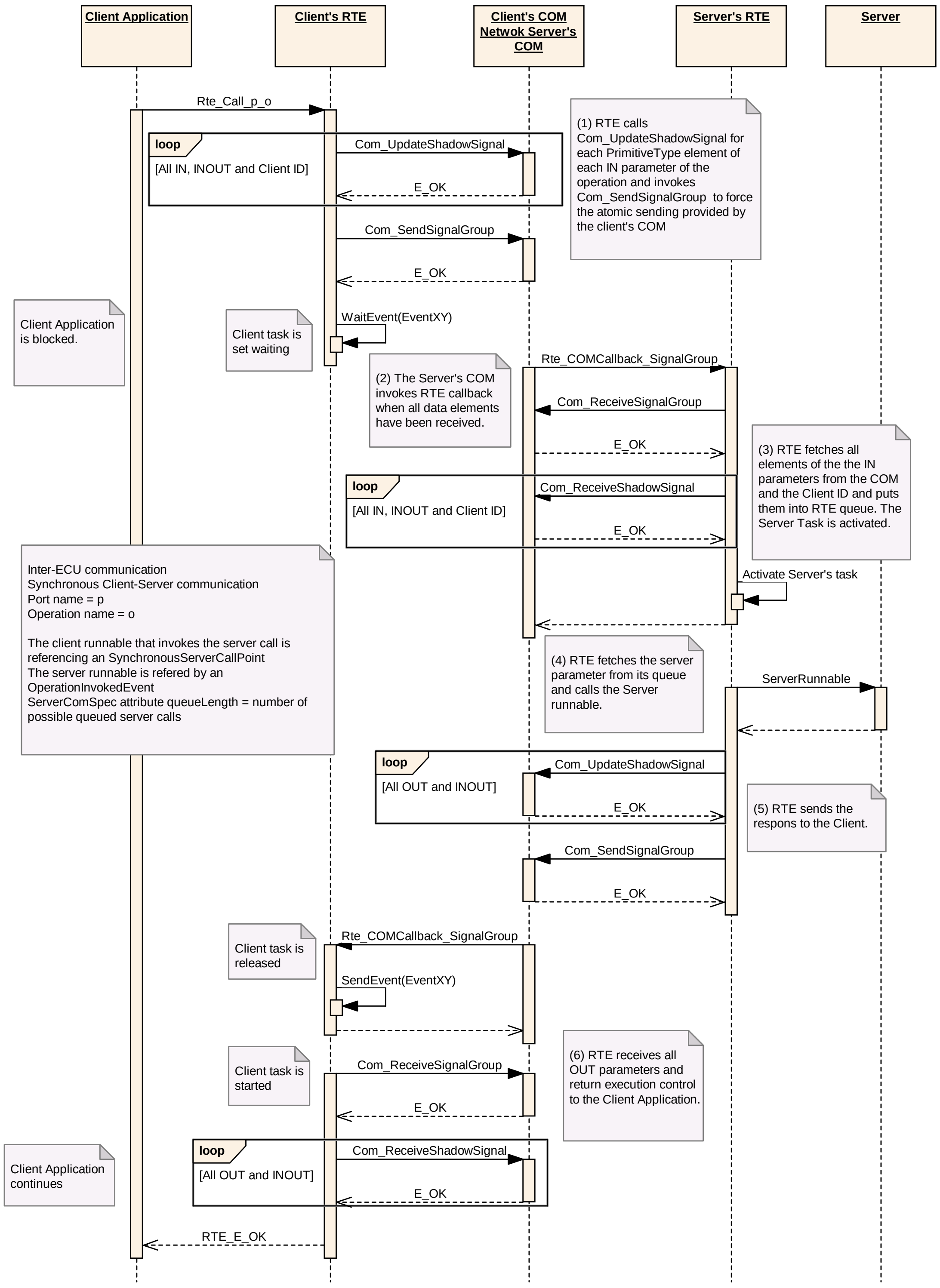

Client Application
Client's RTE
Client's COM Netwok Server's COM
Server's RTE
Server
Rte_Call_p_o
loop
[All IN, INOUT and Client ID]
Com_UpdateShadowSignal
E_OK
(1) RTE calls Com_UpdateShadowSignal for each PrimitiveType element of each IN parameter of the operation and invokes Com_SendSignalGroup to force the atomic sending provided by the client's COM
Com_SendSignalGroup
E_OK
Client Application is blocked.
Client task is set waiting
WaitEvent(EventXY)
Rte_COMCallback_SignalGroup
(2) The Server's COM invokes RTE callback when all data elements have been received.
Com_ReceiveSignalGroup
E_OK
(3) RTE fetches all elements of the the IN parameters from the COM and the Client ID and puts them into RTE queue. The Server Task is activated.
loop
[All IN, INOUT and Client ID]
Com_ReceiveShadowSignal
E_OK
Activate Server's task
Inter-ECU communication
Synchronous Client-Server communication
Port name = p
Operation name = o
The client runnable that invokes the server call is referencing an SynchronousServerCallPoint
The server runnable is refered by an OperationInvokedEvent
ServerComSpec attribute queueLength = number of possible queued server calls
(4) RTE fetches the server parameter from its queue and calls the Server runnable.
ServerRunnable
loop
[All OUT and INOUT]
Com_UpdateShadowSignal
E_OK
(5) RTE sends the respons to the Client.
Com_SendSignalGroup
E_OK
Client task is released
Rte_COMCallback_SignalGroup
SendEvent(EventXY)
Client task is started
Com_ReceiveSignalGroup
E_OK
(6) RTE receives all OUT parameters and return execution control to the Client Application.
Client Application continues
loop
[All OUT and INOUT]
Com_ReceiveShadowSignal
E_OK
RTE_E_OK

Figure 4.19: Client Server synchronous

4.3.3 SWC internal communication

4.3.3.1 InterRunnableVariables

Sender/Receiver and Client/Server communication through AUTOSAR ports are the model for communication between AUTOSAR SW-Cs.

For communication between Runnables inside of an AUTOSAR SW-C the AUTOSAR SW-C Template [18] establishes a separate mechanism. Non-composite AUTOSAR SW-C can reserve InterRunnableVariables which can only be accessed by the Runnables of this one AUTOSAR SW-C. The Runnables might be running in the same or in different task contexts. Read and write accesses are possible.

[rte_sws_3589] The RTE has to support InterRunnableVariables for single and multiple instances of AUTOSAR SW-Cs.

InterRunnableVariables have a behavior corresponding to Sender/Receiver communication *between* AUTOSAR SW-Cs (or rather between Runnables of different AUTOSAR SW-Cs).

But why not use Sender/Receiver communication directly instead? Purpose is data encapsulation / data hiding. Access to InterRunnableVariables of an AUTOSAR SW-C from other AUTOSAR SWCs is not possible and not supported by RTE. InterRunnableVariable content stays SW-C internal and so no other SW-C can use. Especially not misuse it without understanding how the data behaves.

Like in Sender/Receiver (S/R) communication between AUTOSAR SW-Cs two different behaviors exist:

1. InterRunnableVariables with **implicit** behavior
   This behavior corresponds with *DataReadAccess / DataWriteAccess* of Sender/Receiver communication and is supported by *implicit S/R API* in this specification.

   For more details see section 4.2.4.6.1.
   For APIs see sections 5.6.17 and 5.6.18.

2. InterRunnableVariables with **explicit** behavior
   This behavior corresponds with *DataSendPoint / DataReceivePoint* of Sender/Receiver communication and is supported by *explicit S/R API* in this specification.

   For more details see section 4.2.4.6.2
   For APIs see sections 5.6.19 and 5.6.20.

4.4 Modes

The purpose of modes is to to trigger runnables on the transition between modes and to disable (/enable) specified triggers of runnables in certain modes. Here, we use the specification of modes from the Software Component Template Specification [18].

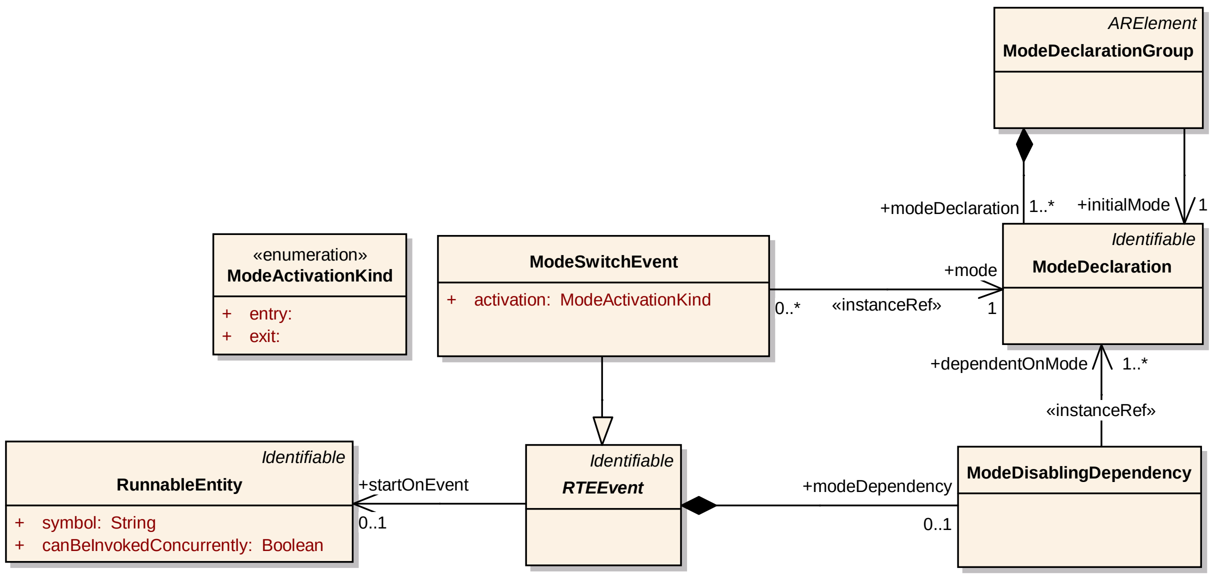


Figure 4.20: Summary of the use of ModeDeclarations by an AUTOSAR Software-Component instance as defined in the Software Component Template Specification [18].

The first subsection 4.4.1 describes how modes can be used by an AUTOSAR software-component. How ModeDeclarations are connected to a state machine is described in subsection 4.4.2. The behaviour of the RTE regarding mode switches is detailed in subsection 4.4.3.

One usecase of modes is described in section 4.5.2 for the initialization and finalization of AUTOSAR Software-Components. Other examples are given in subsection 4.4.4. Modes can be used for handling of communication states as well as for specific application purposes. The general definition of modes and their use is not in the scope of this document.

The status of the modes will be notified to the AUTOSAR software-component using sender/receiver communication as described in the subsection 4.4.5. The port for receiving (or sending) a mode switch notification will be called mode port .

An AUTOSAR SW-C that depends on modes, by ModeDisablingDependency, ModeSwitchEvent, or simply by reading the currend state of a mode is called a mode user in the following.

Entering and leaving modes is initiated by a `mode manager`. A `mode manager` might for example be the communication manager (COMM), the ECU state manager, or an `application mode manager`, e.g., representing the clamp state. An “application mode manager” is an AUTOSAR Software-Component that provides the service of switching modes. The modes of an `application mode manager` are not standardized.

4.4.1 Use of modes by an AUTOSAR Software-Component

To use modes, an AUTOSAR software-component has to reference a ModeDeclarationGroup in a mode port, see section 4.4.5. The ModeDeclarationGroup contains the required modes.

The ModeDeclarations can be used in two ways within the AUTOSAR SW-C (see also figure 4.20):

1. Modes can be used to trigger runnables: The InternalBehavior of the AUTOSAR SW-C can define a ModeSwitchEvent referencing the required ModeDeclaration. This ModeSwitchEvent can then be used as trigger for runnables.

   RTE does not support a wait point for a ModeSwitchEvent (see rte_sws_1358).

2. An RTEEvent that starts a runnable can contain a ModeDisabelingDependency which references a ModeDeclaration.

   [rte_sws_2503] If a runnable entity *r* is referenced with startOnEvent by an RTEEvent *e* that has a ModeDisablingDependency on a mode *m*, then the RTE shall not start *r* on any occurence of *e* while *m* is active.

   The existence of a ModeDisabelingDependency shall prevent the RTE to start a runnable due to the owning RTEEvent during the corresponding mode.

   ModeDisablingDependencies override any activation of a runnable by the owning RTEEvents including the ModeSwitchEvent.

   A runnable can not be "enabled explicitly". A runnable is only "enabled" by the absence of any active ModeDisablingDependency.

   [rte_sws_2504] The existence of a ModeDisabelingDependency shall not instruct the RTE to kill or preempt a running runnable at a mode switch.

   The RTE might switch schedule tables to implement mode disabling dependencies for cyclic triggers of runnables.

   - To do this, the RTE generator needs to know mutual exclusivity and coverage of modes, see rte_sws_2542.
   - **[rte_sws_ext_2559]** The RTE configurator shall have access to the schedule table configuration (see also rte_sws_4014).

4.4.2 Refinement of the semantics of ModeDeclarations and ModeDeclarationGroups

To allow the RTE, e.g., to switch between schedule tables to realize mode dependencies, the RTE needs some basic information about the available modes. For this reason, RTE will make the following additional assumptions about the modes of one ModeDeclarationGroup:

1. **[rte_sws_ext_2542]** Whenever any runnable entity is running, there shall always be exactly one mode active of each ModeDeclarationGroup.

2. **[rte_sws_2544]** After initialization, RTE shall assume the initial mode of each ModeDeclarationGroup to be active.

   RTE will enforce the ModeDisablingDependencies of the initial modes after initialization.

3. **[rte_sws_2626]** If ModeSwitchEvents for entering an initial mode are defined, they shall be triggered by RTE directly after initialization.

   The runnables for entering the initial mode are already executed in the context of the initial mode, see rte_sws_2544 and rte_sws_2564.

4. rte_sws_in_2543 RTE needs to know the initial mode of each ModeDeclarationGroup as input information.

In other words, RTE assumes, that the modes of one ModeDeclarationGroup belong to exactly one state machine without nested states. The state machines cover the whole lifetime of the RTE.

4.4.3 Order of actions taken by the RTE upon interception of a mode switch notification

1. **[rte_sws_2562]** The RTE shall activate the runnables triggered by a ModeSwitchEvent for leaving a mode immediately upon the interception of a notification of the mode switch from the `mode manager`.

   Since a mode switch has to be executed within finite execution time, only category 1 runnables may be connected to ModeSwitchEvents. The RTE generator shall reject configurations with category 2 or 3 runnables connected to ModeSwitchEvents (see rte_sws_2500).

2. **[rte_sws_2563]** The RTE shall execute the switch of mode disablings after the following conditions are met:

   - the runnables that have to be triggered for leaving the previous mode (see rte_sws_2562) have terminated and
   - no runnables are active that are started by RTEEvents with ModeDisabelingDependencies on the previous or next mode.

   Implementation hint: This can be achieved

   (a) actively by counting the active runnables with dependencies on the modes

   (b) or passive by performing the mode switch in a task with lowest priority, e.g., the idle task.

3. **[rte_sws_2564]** Immediately after the switch of mode disablings of RTEEvents (see rte_sws_2563), the RTE shall trigger the ModeSwitchEvents for entering the next mode.

RTE will activate the runnables with startOnEvent dependencies on ModeSwitchEvents for entering the next mode.

4. **[rte_sws_2587]** Immediately after triggering the ModeSwitchEvents for entering the next mode, the RTE shall trigger the DataSendCompletedEvents that are connected to the mode manager's send operation.[9]

   This will result in an acknowledgement on the sender side wich allows the mode manager to wait for completion of the mode switch. Note, that the runnables for entering the next mode may not have terminated when the acknowledgement occurs.

If any of the described steps can not be executed, since no corresponding ModeSwitchDependency or ModeDisablingDependency is defined, RTE shall still execute the subsequent steps.

An implementation of the RTE will have to keep track of the active modes of each ModeDeclarationGroup, to disable the start of runnables as indicated by the ModeDisabelingDependencies, see rte_sws_2546.

[rte_sws_2630] RTE shall execute all steps of a mode switch (see rte_sws_2562, rte_sws_2563, rte_sws_2564, and rte_sws_2587) synchronously for all `mode ports` connected to the same `mode port` of the `mode manager`.

[9]The release candidate 4 of the Software-Component Template Specification [18], does not yet define a SenderComSpec (and AcknowledgementRequest) for a ModeDeclarationGroupPrototype.

4.4.4 Examples

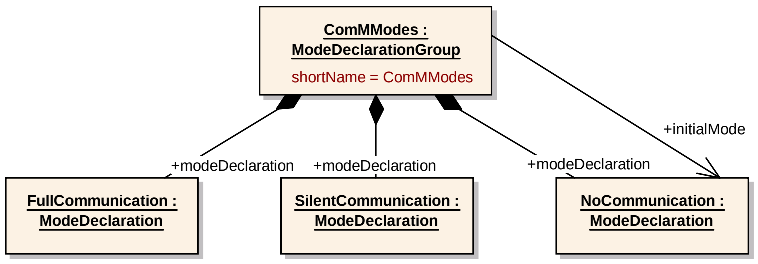

ComMModes : ModeDeclarationGroup
shortName = ComMModes
+modeDeclaration
+modeDeclaration
+modeDeclaration
+initialMode
FullCommunication : ModeDeclaration
SilentCommunication : ModeDeclaration
NoCommunication : ModeDeclaration

Figure 4.21: This is an example of a ModeDeclarationGroup that could be used with the COM manager. (this is no binding specification of ComM!)

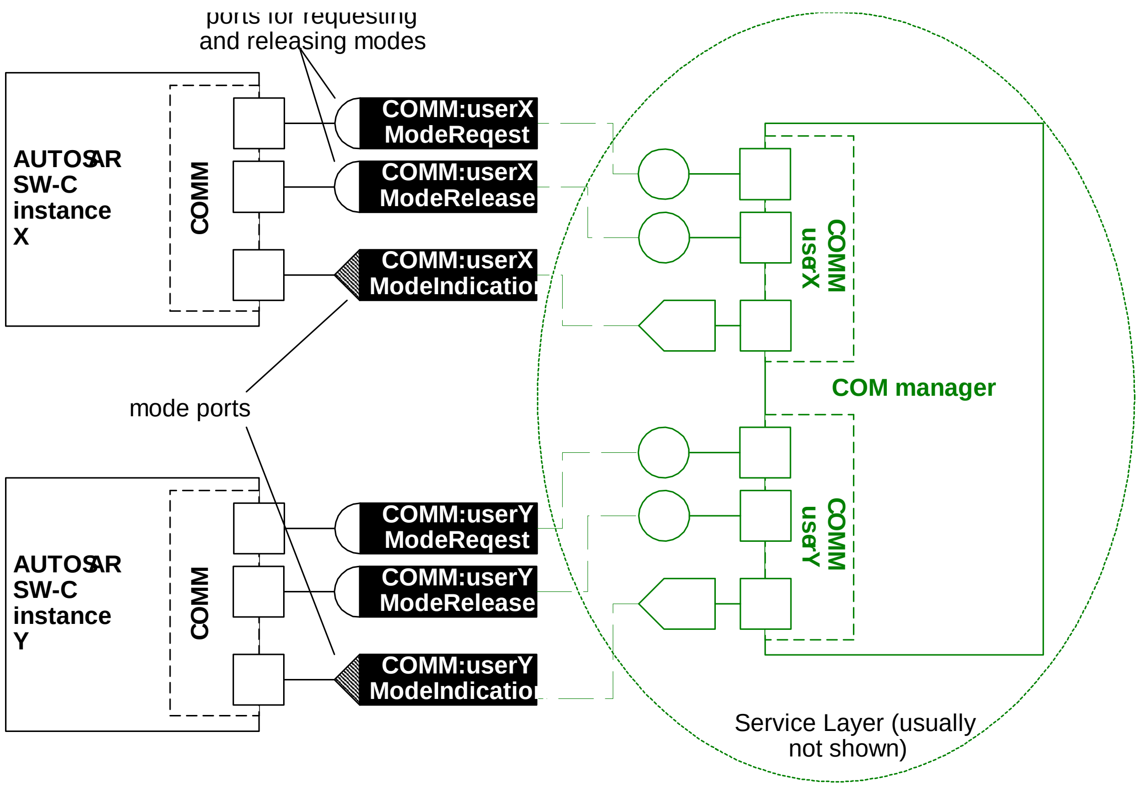


Figure 4.22: This is an example of how a SW-C could be connected to a COM manager (this is no binding specification of ComM!)

This section shows two possible implementations of mode managers. One might be suitable for a Communication Manager (COMM). The COMM is a basic software module that manages a state machine of the communication modes for inter ECU communication. The other example might be suitable for a Vehicle Mode Manager (VMM). In this example, the VMM is responsible for some global vehicle modes like the clamp status. *Please note that this section is purely illustrative. It is in no way a specification or binding for the implementation of COMM or a VMM.*

The modes, that are provided by the COMM are declared in ComMModes (see Fig. 4.21). ComMModes is an instance of a ModeDeclarationGroup. Even though each SW-C (user) uses the same kind of communication modes, described in ComMModes, each

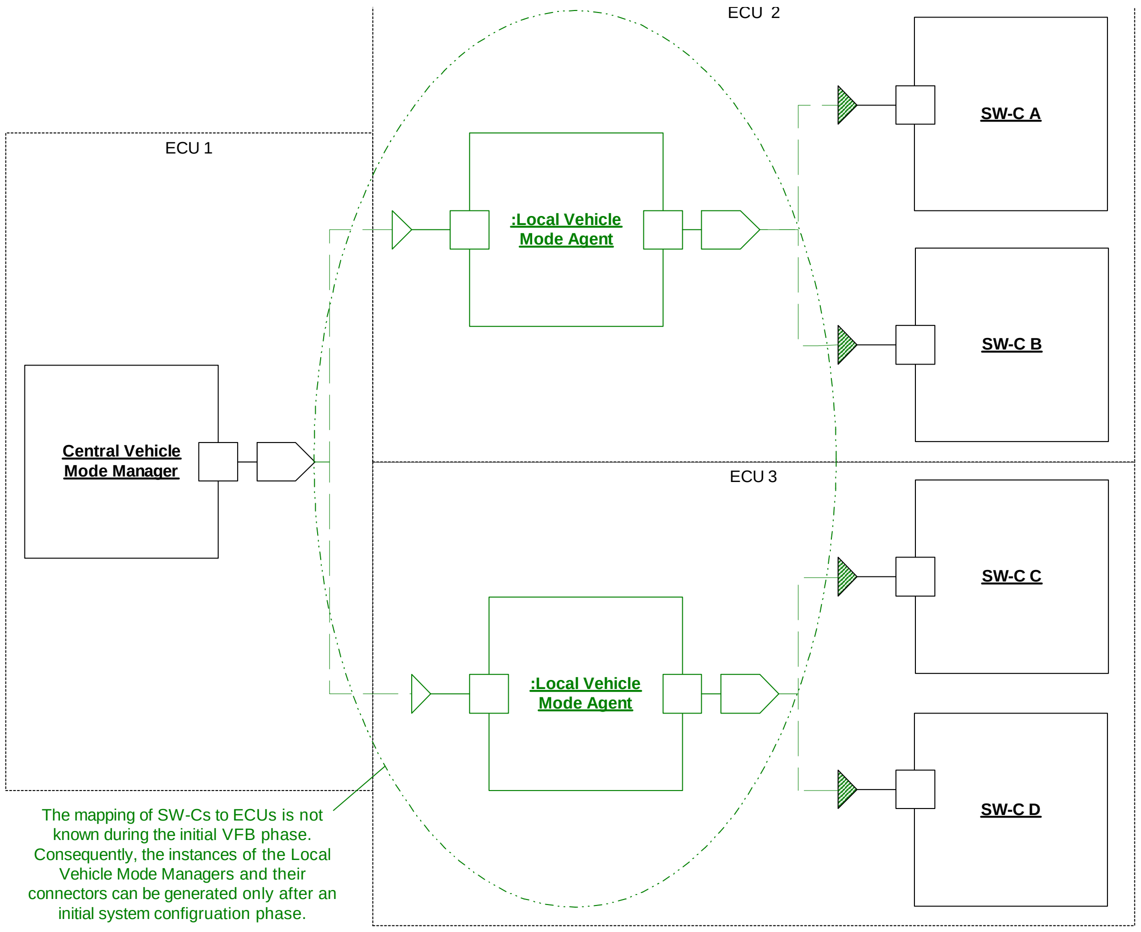

ECU 2
ECU 1
ECU 3
:Local Vehicle Mode Agent
:Local Vehicle Mode Agent
Central Vehicle Mode Manager
SW-C A
SW-C B
SW-C C
SW-C D
The mapping of SW-Cs to ECUs is not known during the initial VFB phase. Consequently, the instances of the Local Vehicle Mode Managers and their connectors can be generated only after an initial system configruation phase.

Figure 4.23: This is an example of how SW-Cs could interact with a central vehicle mode manager using local agents.

SW-C instance might use different physical bus systems. So, their view of the currently active mode is different. The COMM supports this by different mode machines for each user. Consequently, the COMM has a separate `mode port` to provide the mode information for each user. This is shown in Fig. 4.22.

In addition to the `mode port`, the user has additional client/server ports to request and release modes of the COMM. The request and release communication is not integrated in the `mode port`.

Fig. 4.23 gives an implementation example for the VMM. The VMM is not part of the basic software, but of AUTOSAR Software-Components. By definition, the VMM deals with vehicle global modes. To ensure a consistent behavior, when the connection to the central VMM is lost, local agents of the VMM have to be provided on each ECU. The local agents are `application mode manager`s. The local VMM agents only need one port to distribute the mode switch information to all SW-Cs on that ECU.

It has to be noted that during VFB phase, the mapping of SW-Cs to ECUs is not known. Consequently, the instances of the Local VMM Agents and their connectors can be generated only after an initial system configruation phase.

4.4.5 Notification of mode switches

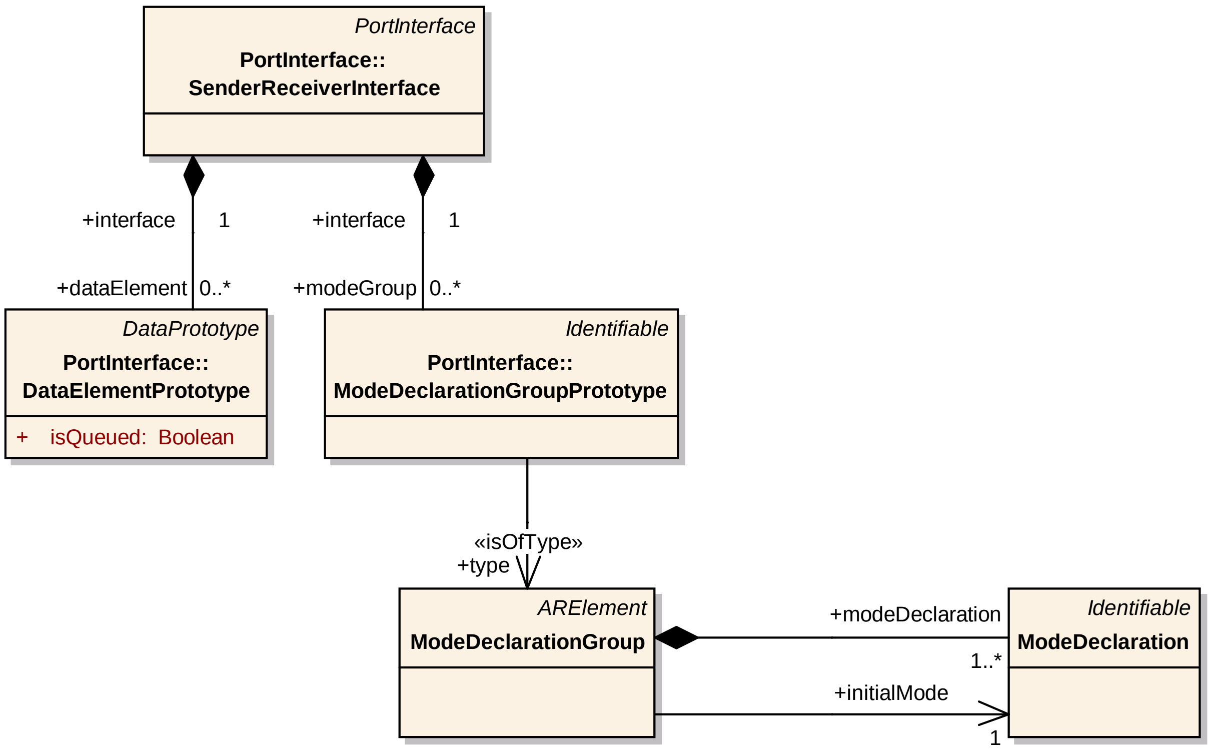


Figure 4.24: Definition of a mode switch interface.

- **[rte_sws_2549]** Mode switches shall be communicated by ModeDeclarationGroup-Prototypes of a SenderReceiverInterface as defined in [18], see Fig. 4.24.

  A SenderReceiverInterface with a ModeDeclarationGroupPrototype of a ModeDeclarationGroup will be called mode switch interface for the ModeDeclarationGroup in the following. The `mode ports` of the `mode manager` and the `mode user` are of the type of a `mode switch interface`.

- RTE only requires the notification of switches between modes. A `mode manager` may also provide an interface for requesting the currently active mode. But, this is not in the scope of this specification.

- **[rte_sws_ext_2507]** The mode switch shall be notified to the mode user (and RTE) locally on each ECU.

  [rte_sws_2586]The RTE generator shall reject a configuration with a nonlocal connection of a ModeDeclarationGroupPrototype.

  Rationale: Even without communication to other ECUs, each state machine has to be in a well defined state/mode. This Requirement rte_sws_ext_2507 does not prevent distributed mode management. But, for distributed mode management, a local agent is required on each ECU.

  This implies that the connector between an `application mode manager` instance and the `mode user` instance can only be created after mapping of the

SW-C to an ECU, because the `application mode manager` instance needs to be a specific agent, bound to one ECU.

- **[rte_sws_2508]** A mode switch shall be notified asynchronously as indicated by the use of a SenderReceiverInterface.

  Rationale: This simplifies the communication. Due to rte_sws_ext_2507 the communication is local and no handshake is required to guarantee reliable transmission.

  RTE offers the api `Rte_Switch` to the mode manager for this notification, see 5.6.4.

- The `mode manager` might still require a feedback to keep it's internal state machine synchronized with the RTE view of active modes.

  The RTE generator shall support an AcknowledgementRequest from the `mode port` of a `mode manager`, see rte_sws_2587, to notify the `mode manager` of the completion of a mode switch.

- **[rte_sws_2566]** A `mode switch interface` shall support 1:n communication.

  Rationale: This simplifies the configuration and the communication. One mode switch can be notified to all receivers simultaneously.

- **[rte_sws_2624]** A mode switch shall be notified with event semantics, i.e., the mode switch notifications shall be buffered by RTE like data elements with isQueued = TRUE.

  The queueing of mode switches (and ModeSwitchEvents) depends like that of DataReceivedEvents on the settings for the receiving port, see section 4.3.1.10.2.[10]

- **[rte_sws_2567]** A `mode switch interface` shall only indicate the next mode of the transition.

  RTE will use identifiers of the modes as defined in rte_sws_2568.

- **[rte_sws_2546]** The RTE shall keep track of the active modes of a ModeDeclarationGroup.

  Rationale: This allows the RTE to guarantee consistency between the timing for fireing of ModeSwitchEvents and disabling the start of runnables by ModeDisabelingDependency without blowing the interface to a mode manager with fine grained substates on the transitions.

- The RTE offers an API `Rte_Mode` to the SW-C to get the active mode, see section 5.6.23.

- In addition to the `mode ports`, the `mode manager` may offer an AUTOSAR interface for requesting and releasing modes as a means to keep modes alive like for COMM and ECU State Manager.

[10]The release candidate 4 of the Software-Component Template Specification [18], does not yet define a ReceiverComSpec (and queueLength) for a ModeDeclarationGroupPrototype. By default, a queueLength of 1 should be assumed.

4.4.6 Examples continued

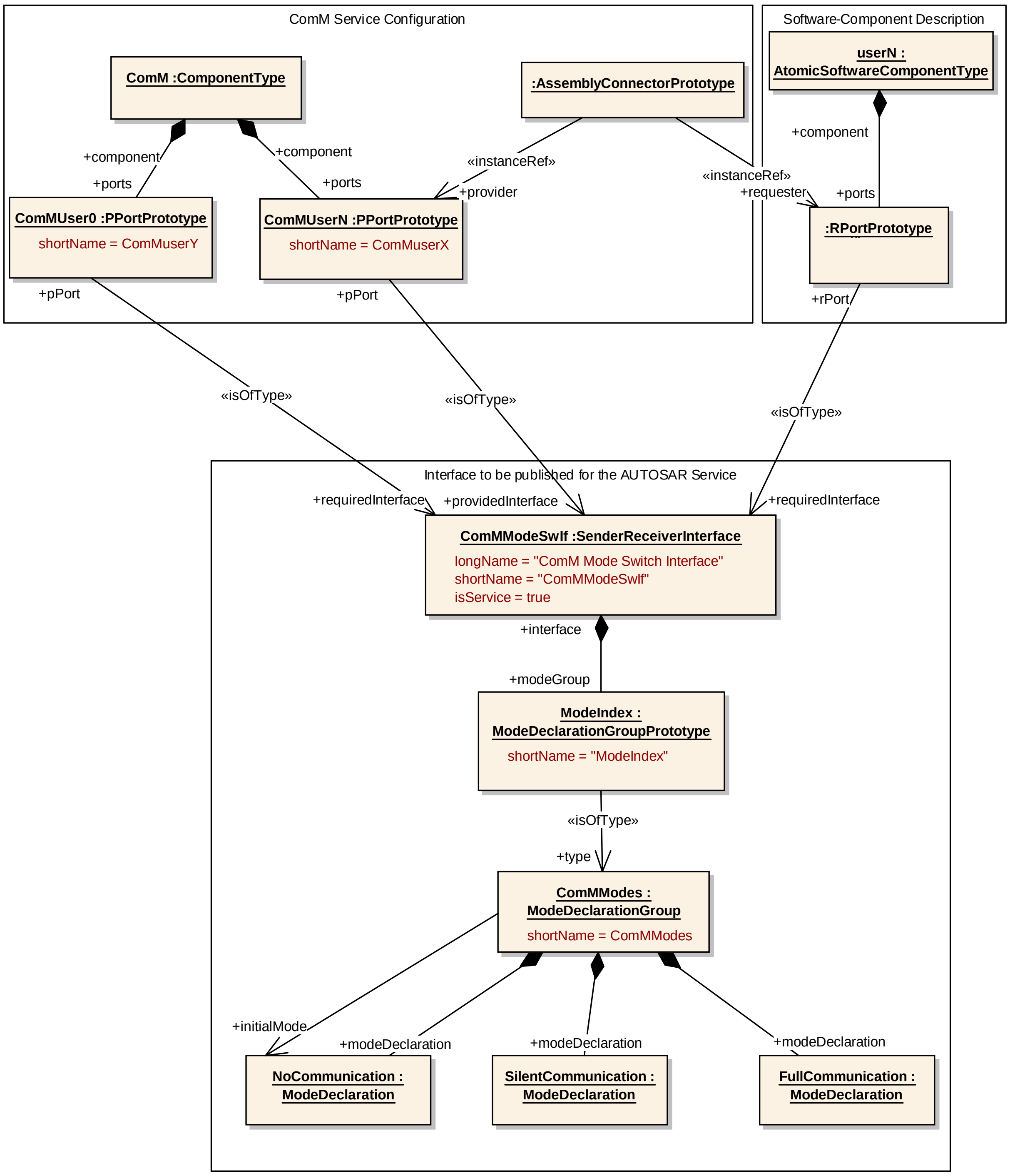

ComM Service Configuration
Software-Component Description
ComM :ComponentType
:AssemblyConnectorPrototype
userN : AtomicSoftwareComponentType
+component
+component
+component
«instanceRef»
«instanceRef»
+ports
+ports
+provider
+requester
+ports
ComMUser0 :PPortPrototype
shortName = ComMuserY
ComMUserN :PPortPrototype
shortName = ComMuserX
:RPortPrototype
+pPort
+pPort
+rPort
«isOfType»
«isOfType»
«isOfType»
Interface to be published for the AUTOSAR Service
+requiredInterface
+providedInterface
+requiredInterface
ComMModeSwIf :SenderReceiverInterface
longName = "ComM Mode Switch Interface"
shortName = "ComMModeSwIf"
isService = true
+interface
+modeGroup
ModeIndex : ModeDeclarationGroupPrototype
shortName = "ModeIndex"
«isOfType»
+type
ComMModes : ModeDeclarationGroup
shortName = ComMModes
+initialMode
+modeDeclaration
+modeDeclaration
+modeDeclaration
NoCommunication : ModeDeclaration
SilentCommunication : ModeDeclaration
FullCommunication : ModeDeclaration

Figure 4.25: This is an example, how the definition of ports of a mode manager and the connection of a user N to the mode manager might look like. (this is no specification of ComM!)

This section shows how the `mode ports` for the COMM example from section 4.4.4 might be formally defined. The box 'Interface ...´ of Fig. 4.25 shows a possible definition of the interface of COMM. The other boxes show the definition of the `mode ports` of the mode manager and a user. The ports are connected by an AssemblyConnectorPrototype. Please note that this connector will not be specified during VFB phase. Additional ports for requesting and releasing modes are not shown.

4.5 Initialization and Finalization

4.5.1 Initialization and Finalization of the RTE

The ECU state manager calls the startup routine `Rte_Start` of the RTE at the end of startup phase II when the OS is available and all basic software modules are initialized.

[rte_sws_2513] The initialization routine of the RTE shall return within finite execution time.

Before the RTE is initialized completely, there is only a limited capability of RTE to handle incoming data from COM:

[rte_sws_2535] RTE shall ingore incoming client server communication requests, before RTE is initialized completely.

[rte_sws_2536] Incoming data and events from sender receiver communication shall be ignored, before RTE is initialized completely.

The finalization routine `RTE_Stop` of the RTE is called by the ECU state manager at the beginning of shutdown phase I when the OS is still available. (For details of the ECU state manager, see [8]. For details of `Rte_Start` and `Rte_Stop` see section 5.8.)

[rte_sws_2538] Finalization of the RTE shall imply that all SW-C are stopped and not executed any more.

4.5.2 Initialization and Finalization of AUTOSAR Software-Components

For the initialization and finalization of AUTOSAR software components, RTE provides the mechanism of mode switches. A ModeSwitchEvent of an appropriate ModeDeclaration can be used to trigger a corresponding initialization or finalization runnable (see rte_sws_2562). Runnables that shall not run during initialization or finalization can be disabled in the corresponding modes with a ModeDisabelingDependency (see rte_sws_2503).

Since category 2 runnables have no predictable execution time and can not be terminated using ModeDisablingDependencies, it is the responsibility of the implementer to set meaningful termination criteria for the cat 2 runnables. These criteria could include mode information. At latest, all runnables will be terminated by RTE during the shutdown of RTE, see rte_sws_2538.

It is appropriate to use user defined modes that will be handled in a proprietary `application mode manager`.

All runnables that are triggered by entering an initial mode, are activated immediately after the initialization of RTE. They can be used for initialization. In many cases it it might be prefereable to have a multi step initialization supported by a sequence of different initialization modes.

4.6 RTE Functionality Levels

There is a single RTE functionality level. So RTE is compliant AUTOSAR Functionality Conformance Class 1 (FCC1)

5 RTE Reference

“Everything should be as simple as possible, but no simpler.”
– *Albert Einstein*

5.1 Scope

This chapter presents the RTE API from the perspective of AUTOSAR applications and basic software – the same API applies to all software whether they are AUTOSAR software-components or basic software.

Section 5.2 presents basic principles of the API including naming conventions and supported programming languages. Section 5.3 describes the header files used by the RTE and the files created by an RTE generator. The data types used by the API are described in Section 5.5 and Sections 5.6 and 5.7 provide a reference to the RTE API itself including the definition of runnable entities. Section 5.10 defines the events that can be monitored during VFB tracing.

5.1.1 Programming Languages

The RTE is required to support components written using the C and C++ programming languages [RTE00126] as well as legacy software modules [RTE_IN016]. The ability for multiple languages to use the same generated RTE is an important step in reducing the complexity of RTE generation and therefore the scope for errors.

[rte_sws_1167] The RTE shall be generated in C.

[rte_sws_1168] All RTE code, whether generated or not, shall conform to the HIS subset of the MISRA C standard. In technically reasonable, exceptional cases MISRA violations are permissable. Such violations shall be clearly identified and documented.

Specified MISRA violations are defined in Appendix D.

The RTE API presented in Section 5.6 is described using C. The API is also directly accessible from an AUTOSAR software-component written using C++ provided all API functions and instances of data structures are imported with C linkage.

[rte_sws_1011] The RTE generator shall ensure that, for a component written in C++, all imported RTE symbols are declared using C linkage.

For the RTE API for C and C++ components the import of symbols occurs within the application header file (Section 5.3.3).

5.1.2 Generator Principles

5.1.2.1 Operating Modes

An object-code component is compiled against an application header file that is created during the first “RTE Contract” phase of RTE generation. The object code is then linked against an RTE created during the second “RTE Generation” phase. To ensure that the object-code component and the RTE code are compatible the RTE generator supports *compatibility mode* that uses well-defined data structures and types for the component data structure. In addition, an RTE generator may support a *vendor* operating mode that removes compatibility between RTE generators from different vendors but permits implementation specific, and hence potentially more efficient, data structures and types.

[rte_sws_1195] All RTE operating modes shall be source-code compatible at the SW-C level.

Requirement rte_sws_1195 ensures that a SW-C can be used in any operating mode as long as the source is available. The converse is not true – for example, an object-code SW-C compiled after the “RTE Contract” phase must be linked against an RTE created by an RTE generator operating in the same operating mode. If the vendor mode is used in the “RTE Contract” phase, an RTE generator from the same vendor (or one compatible to the vendor-mode features of the RTE generator used in the “RTE Contract” phase) has to be used for the “RTE Generation” phase.

5.1.2.1.1 Compatibility Mode

Compatibility mode is the default operating mode for an RTE generator and guarantees compatibility even between RTE generators from different vendors through the use of well-defined, “standardized”, data structures. The data structures that are used by the generated RTE in the compatibility mode are defined in Section 5.4.

Support for compatibility mode is required and therefore is guaranteed to be implemented by all RTE generators.

[rte_sws_1151] The *compatibility mode* shall be the default operating mode and shall be supported by all RTE generators, whether they are for the “RTE Contract” or “RTE Generation” phases.

The compatibility mode uses custom (generated) functions with standardized names and data structures that are defined during the “RTE Contract” phase and used when compiling object-code components.

[rte_sws_1216] SW-Cs that are compiled against an “RTE Contract” phase application header file (i.e. object-code SW-Cs) generated in compatibility mode shall be compatible with an RTE that was generated in compatibility mode.

The use of well-defined data structures imposes tight constraints on the RTE implementation and therefore restricts the freedom of RTE vendors to optimize the solution of object-code components but have the advantage that RTE generators from different vendors can be used to compile a binary-component and to generate the RTE.

Note that even when an RTE generator is operating in compatibility mode the data structures used for *source-code* components are not defined thus permiting vendor-specific optimizations to be applied.

5.1.2.1.2 Vendor Mode

Vendor mode is an optional operating mode where the data structures defined in the "RTE Contract" phase and used in the "RTE Generation" phase are implementation specific rather than "standardized".

[rte_sws_1152] An RTE generator may optionally support *vendor mode*.

The data structures defined and declared when an RTE generator operates in vendor mode are implementation specific and therefore *not* described in this document. This omission is deliberate and permits vendor-specific optimizations to be implemented for object-code components. It also means that RTE generators from different vendors are unlikely to be compatible when run in the vendor mode.

The operating mode for a component is assumed to be "compatibility" mode unless the component is explicitly marked as requiring "vendor" mode.

[rte_sws_1234] An AUTOSAR software-component shall be assumed to be operating in "compatibility" mode unless "vendor mode" is explicitly requested.

[rte_sws_in_3750] An AUTOSAR software-component shall indicate its required operating mode.

The potential for more efficient implementations of object-code components offered by the vendor mode comes at the expense of requiring high cohesion between object-code components (compiled after the "RTE Contract" phase) and the generated RTE. However, this is not as restrictive as it may seem at first sight since the tight coupling is also reflected in many other aspects or the AUTOSAR methodology, not least of which is the requirement that the same compiler (and compatible options) is used when compiling both the object-code component and the RTE.

5.2 API Principles

[rte_sws_1316] The RTE shall be configured and/or generated for each ECU [RTE00021].

Part of the process is the customization (i.e. configuration or generation) of the RTE API for each AUTOSAR software-component on the ECU. The customization of the API implementation for each AUTOSAR software-component, whether by generation anew or configuration of library code, permits improved run-time efficiency and reduces memory overheads.

The design of the RTE API has been guided by the following core principles:

- The API should be orthogonal – there should be only one way of performing a task.
- **[rte_sws_1314]** The API shall be compiler independent.
- **[rte_sws_3787]** The RTE implementation shall use the compiler abstraction.
- **[rte_sws_1315]** The API shall support components where the source-code is available [RTE00024] and where only object-code is available [RTE00140].
- The API shall support the multiple instantiation of AUTOSAR software-components [RTE00011] that share code [RTE00012].

Two forms of the RTE API are available to software-components; direct and indirect. The direct API has been designed with regard to efficient invocation and includes an API mapping that can be used by an RTE generator to optimize a component's API, for example, to permit the direct invocation of the generated API functions or even eliding the generated RTE completely. The indirect API cannot be optimized using the API mapping but has the advantage that the handle used to access the API can be stored in memory and accessed, via an iterator, to apply the same API to multiple ports.

5.2.1 RTE Namespace

All RTE symbols (e.g. function names, global variables, etc.) visible within the global namespace are required to use the "Rte" prefix.

[rte_sws_1171] All externally visible symbols created by the RTE generator shall use the prefix `Rte_`.

In order to maintain control over the RTE namespace the creation of symbols in the global namespace using the prefix `Rte_` is reserved for the RTE generator.

The generated RTE is required to work with components written in several source languages and therefore should not use language specific features, such as C++ namespaces, to ensure symbol name uniqueness.

5.2.2 Direct API

The direct invocation form is the form used to present the RTE API in Section 5.6. The RTE direct API mapping is designed to be optimizable so that the instance handle is elided (and therefore imposes zero run-time overhead) when the RTE generator can determine that exactly one instance of a component is mapped to an ECU.

[rte_sws_1048] The RTE shall support direct invocation of generated API functions where the instance handle is passed to the API as the first formal parameter.

All runnable entities for a AUTOSAR software-component type are passed the same instance handle type (as the first formal parameter) and can therefore use the same type definition from the component's application header file.

The direct API can also be further optimized for source code components via the API mapping.

5.2.3 Indirect API

The indirect API is an alternative form of API invocation that uses indirection through a port handle to invoke RTE API functions rather than direct invocation. This form is less efficient (the indirection cannot be optimized away) but supports a different programming style that may be more convenient. For example, when using the indirect API, an array of port handles of the same interface and provide/require direction is provided by RTE and the same RTE API can be invoked for multiple ports by iterating over the array.

Both direct and indirect forms of API call are equivalent and result in the same generated RTE function being invoked.

The semantics of the port handle must be the same in both the “RTE Contract” and “RTE Generation” phases since the port handle accesses the standardized data structures of the RTE.

It is possible to mix the indirect and direct APIs within the same SW-C.

The indirect API uses port handles during the invocation of RTE API calls. The type of the port handle is determined by the port interface that types the port which means that if a component declares multiple ports typed by the same port interface the port handle points to an array of port data structures and the same API invoked for each element.

The port handle type is defined in Section 5.4.2.6.

5.2.3.1 Accessing Port Handles

An AUTOSAR SW-C needs to obtain port handles using the instance handle before the indirect API can be used. The definition of the instance handle in Section 5.4.2 defines the “Port API” section of the component data structure and these entries can be used to access the port handles in either object-code or source-code components.

The API `Rte_Ports` and `Rte_NPorts` provides port data handles of a given interface. Example 5.1 shows how the indirect API can be used to apply the same operation to multiple ports in a component within a loop.

Example 5.1

The port handle points to an array that can be used within a loop to apply the same operation to each port. The following example sends the same data to each receiver:

```
void TT1(Rte_Instance self)
{
  Rte_PortHandle_interface1_P my_array;
  my_array=Rte_Ports_interface1_P(self);
  int s;
  for(s = 0; s < Rte_NPorts_interface1_P(self); s++) {
    my_array[s].Send_a(23);
  }
}
```

5.2.4 DataReadAccess and DataWriteAccess

The RTE is required to support DataReadAccess and DataWriteAccess semantics for data elements. The required semantics are subject to two constraints:

- For DataReadAccess, the data accessed by a runnable entity must not change during the lifetime of the runnable entity.
- For DataWriteAccess, the data written by a runnable entity is only visible to other runnable entities after the accessing runnable entity has terminated.

The generated RTE satisfies both requirements through data copies that are created when the RTE is generated based on the known task and runnable mapping.

Example 5.2

Consider a data element, a, of port p which is accessed using DataReadAccess semantics by runnable `re1` and DataWriteAccess by runnable `re2`. Furthermore, consider that `re1` and `re2` are mapped to different tasks and that execution of `re1` can pre-empt `re2`.

In this example, the RTE will create two different copies to contain a to prevent updates from `re2` ‘corrupting’ the value access by `re1` since the latter must remain unchanged during the lifetime of `re1`.

Rather than providing direct access to the data copies, the RTE API includes two API calls to support DataReadAccess and DataWriteAccess for a software-component; `Rte_IRead` (see Section 5.6.13) and `Rte_IWrite` (see Section 5.6.14). These API calls access the data copies (for read and write access respectively). The use of an API call enables the definition to be changed based on the task and runnable mapping without affecting the software-component code.

Example 5.3

Consider a data element, a, of port p which is declared as being accessed using DataWriteAccess semantics by runnables `re1` and `re2` within component c. The RTE API for component c will then contain two API functions to write the data element;

```
void Rte_IWrite_re1_p_a(Rte_Instance self, <type> val);
void Rte_IWrite_re2_p_a(Rte_Instance self, <type> val);
```

The two defined API calls are used by `re1` and `re2` as required. The definitions of the API depend on where the data copies are defined. If both re1 and re2 are mapped to the same task then each can access the same copy. However, if re1 and re2 are mapped to different (pre-emptable) tasks then the RTE will ensure that each API access a different copy.

The Rte_IRead and Rte_IWrite use the "data handles" defined in the component data structure (see Section 5.4.2).

5.2.5 PerInstanceMemory

The RTE is required to support PerInstanceMemory [RTE00013].

The component's instance handle defines a particular instance of a component and is therefore used when accessing the PerInstanceMemory using the Rte_Pim API.

Note that the PerInstanceMemory concept is not applicable to access NVRAM blocks of the BSW an AUTOSAR SW-C instance might need.

The Rte_Pim API does not impose the RTE to apply a data consistency mechanism for the access to PerInstanceMemory. An application is responsible for consistency of accessed data by itself. This design decision permits efficient (zero overhead) access when required. If a component possesses multiple runnable entities that require concurrent access to the same PerInstanceMemory, an exclusive area can be used to ensure data consistency, either through explicit Rte_Enter and Rte_Exit API calls or by declaring that, implicitly, the runnable entities run inside an exclusive area.

Thus, the PerInstanceMemory is exclusively used by a particular software-component instance and needs to be declared and allocated (statically).

[rte_sws_2303] The generated RTE shall declare PerInstanceMemory in accordance to the attribute *type* of a particular *PerInstanceMemory*.

In addition, the attribute *type* needs to be defined in the corresponding software-component header. Therefore, the attribute *typeDefinition* of the *PerInstanceMemory* contains its definition as plain text string. It is assumed that this text is valid 'C' syntax, because it will be included verbatim in the application header file.

[rte_sws_2304] The generated RTE shall define the type of a PerInstanceMemory by interpreting the text string of the attribute *typeDefinition* of a particular *PerInstance-Memory* as the 'C' definiton.

Note that the type is specified within the scope of a software component and therefore not necessarily unique within the scope of the ECU. Therefore the RTE needs to define a unique type within the RTE Types header file while providing the component-specific type via the application header file to the software component.

[rte_sws_3789] The RTE types header file shall contain the typedefinition
`typedef <typedefinition> Rte_PimType_<c>_<type>;`
where ¡typedefinition¿ is the content of the <typeDefinition> string of the PerInstanceMemory, <type> the name of the type and <c> the name of the component type to which the PerInstanceMemory belongs.

[rte_sws_3782] The type of a PerInstanceMemory shall be defined in the application header file as
`typedef Rte_PimType_<c>_<type> <type>;`
where <c> is the component type name and `<type>` the type name of the PerInstanceMemory.

[rte_sws_2305] The generated RTE shall instantiate (or allocate) declared PerInstanceMemory.

However, the mechanism used to allocate memory is outside of the scope of this specification. Note that the memory allocated for a PerInstanceMemory is not initialized by the generated RTE, but by the corresponding software-component instances.

Example 5.4

A software-component c contains a particular *PerInstanceMemory* `mem` with the attributes *type* = `MyMemType` and *typeDefinition* = `struct {uint16 val1; uint8 * val2; };`. This description shall result in the following code:

In the RTE Types header file:

```
/* typedef to ensure unique typename */
/* according to the attributes */
/* 'type' and 'typeDefinition' */
typedef struct{
                uint16 val1;
                 uint8 * val2;
} Rte_PimType_c_MyMemType;
```

In the respective application header file:

```
/* typedef visible within the scope */
/* of the component according to the attributes */
/* 'type' and 'typeDefinition' */
typedef Rte_PimType_c_MyMemType MyMemType;
```

In Rte.c:

```
/* declare and instantiate mem */
Rte_PimType_c_MyMemType mem1;
```

Note that the name used for the definition (in example 5.4 `mem1`) does not appear outside of the generated RTE and is therefore vendor specific.

5.2.6 API Mapping

The RTE API is implemented by macros and generated API functions that are created (or configured, depending on the implementation) by the RTE generator during the “RTE Generation” phase. Typically one customized macro or function is created for each “end” of a communication though the RTE generator may elide or combine custom functions to improve run-time efficiency or memory overheads.

[rte_sws_1274] The API mapping shall be implemented in the application header file.

The RTE generator is required to provide a mapping from the RTE API name to the generated function [RTE00051]. The API mapping provides a level of indirection necessary to support binary components and multiple component instances. The indirection is necessary for two reasons. Firstly, some information may not be known when the component is created, for example, the component's instance name, but are necessary to ensure that the names of the generated functions are unique. Secondly, the names of the generated API functions should be unique (so that the ECU image can link correctly) and the steps taken to ensure this may make the names not "user-friendly". Therefore, the primary rationale for the API mapping is to provide the required abstraction that means that a component does not need to concern itself with the preceding problems.

The requirements on the API mapping depend on the phase in which an RTE generator is operating. The requirements on the API mapping are only binding for RTE generators operating in compatibility mode.

5.2.6.1 "RTE Contract" Phase

Within the "RTE Contract" phase the API mapping is required to convert from the source API call (as defined in Section 5.6) to the runnable entity provided by a software-component or the implementation of the API function created by the RTE generator.

When compiled against a "RTE Contract" phase header file a software-component that can be multiply instantiated is required to use a general API mapping that uses the instance handle to access the function table defined in the component data structure.

[rte_sws_3706] If a software-component supports multiple instantiation rte_sws_in_0004, the "RTE Contract" phase API mapping shall access the generated RTE functions using the instance handle to indirect through the generated function table in the component data structure.

> **Example 5.5**
>
> For a required client-server port 'p1' with operation 'a' with a single argument, the general form of the API mapping would be:
>
> ```
> #define Rte_Call_p1_a(s,v) ((s)->p1.Call_a(v))
> ```
>
> Where s is the instance handle.

[rte_sws_3707] If a software-component does not support multiple instantiation rte_sws_in_0004, the "RTE Contract" phase API mapping shall access the generated RTE functions directly.

When accessed directly, the names of the generated functions are formed according to the following rules:

- **[rte_sws_1143]** The function generated for API calls of the form `<name>_<p>_<o>` shall be `<name>_<c>_<p>_<o>` where `<name>` is the API root (e.g. Call), `<p>`

the port name, <o> the data element or operation name and <c> the component type name.

- **[rte_sws_1348]** The function generated for API calls of the form `<name>_<re>_<p>_<o>` shall be `<name>_<c>_<re>_<p>_<o>` where `<name>` is the API root (e.g. IrvRead), <p> the port name, `<re>` the runnable entity name and <o> the data element or operation name and <c> the component type name.
- **[rte_sws_1155]** The function generated for API calls of the form `<name>_<e>` shall be `<name>_<c>_<e>` where `<name>` is the API root (e.g. Enter), <e> the API name (e.g. an exclusive area name) and <c> is the component type name.
- **[rte_sws_1156]** The macro generated for the `Rte_Pim` and `Rte_CData` API calls shall map to the relevant fields of the component data structure.

The functions generated that are the destination of the API mapping, which is created during the “RTE Contract” phase, are created by the RTE generator during the second “RTE Generation” phase.

[rte_sws_1153] The generated function (or runnable) shall take the same parameters, in the same order, as the API mapping.

Example 5.6

For a required client-server port ‘p1’ with operation ‘a’ with a single argument for component type ‘c1’ for which multiple instantiation is forbidden, the following mapping would be generated:

```
#define Rte_Call_p1_a Rte_Call_c1_p1_a
```

5.2.6.2 “RTE Generation” Phase

There are no requirements on the *form* that the API mapping created during the “RTE Generation” phase should take. This is because the application header files defined during this phase are used by source-code components and therefore compatibility between the generated RTE and source-code components is automatic.

The RTE generator is required to produce the component data structure instances required by object-code components and multiply instantiated source-code components.

If multiple instantiation of a software-component is forbidden, then the API mapping specified for the “RTE Contract” phase (Section 5.2.6.1) defines the names of the generated functions. If multiple instantiation is possible, there are no corresponding requirements that define the name of the generated function since all accesses to the generated functions are performed via the component data structure which contains well-defined entries (Sections 5.4.2.6 and 5.4.2.6).

5.2.6.3 Function Elidation

Using the “RTE Generation” phase API mapping, it is possible for the RTE generator to elide the use of generated RTE functions.

[rte_sws_1146] If the API mapping elides an RTE function the “RTE Generation” phase API mapping mechanism shall ensure that the invoking component still receives a “return value” so that no changes to the AUTOSAR software-component are necessary.

In C, the elidation of API calls can be achieved using a comma expression[1]

Example 5.7

As an example, consider the following component code:

```
Std_ReturnType s;
s = Rte_Send_p1_a(self,23);
```

Furthermore, assume that the communication attributes are specified such that the sender-receiver communication can be performed as a direct assignment and therefore no RTE API call needs to be generated. However, the component source cannot be modified and expects to receive an `Std_ReturnType` as the return. The “RTE Generation” phase API mapping could then be rewritten as:

```
#define Rte_Send_p1_a(s,a) (<var> = (a), RTE_E_OK)
```

Where `<var>` is the implementation dependent name for an RTE created cache between sender and receiver.

5.2.6.4 API Naming Conventions

An AUTOSAR software-component communicates with other components (including basic software) through ports and therefore the names that constitute the RTE API are formed from the combination of the API call’s functionality (e.g. Call, Send) that defines the API root name and the access point through which the API operates.

For any API that operates through a port, the API’s access point includes the port name.

A SenderReceiverInterface can support multiple data items and a ClientServerInterface can support multiple operations, any of which can be invoked through the requiring port by a client. The RTE API therefore needs a mechanism to indicate which data item/operation on the port to access and this is implemented by including the data item/operation name in the API’s access point.

As described above, the RTE API mapping is responsible for mapping the RTE API name to the correct generated RTE function. The API mapping permits an RTE gener-

[1]This is contrary to MISRA Rule 42 “*comma expression shall not be used except in the control expression of a for loop*”. However, a comma expression is valid, legal, C and the elidation cannot be achieved without a comma expression and therefore the rule must be relaxed.

ator to include targeted optimization as well as removing the need to implement functions that act as routing functions from generic API calls to particular functions within the generated RTE.

For C and C++ the RTE API names introduce symbols into global scope and therefore the names are required to be prefixed with `Rte_` rte_sws_1171.

5.2.6.5 API Parameters

All API parameters fall into one of two classes; parameters that are strictly read-only ("In" parameters) and parameters whose value may be modified by the API function ("In/Out" and "Out" parameters).

The type of these parameters is taken from the data element prototype or operation prototype in the interface that characterizes the port for which the API is being generated.

- **"In" Parameters**

  [rte_sws_1017] All input parameters that are a primitive data types (with the exception of a string) shall be passed by value.

  [rte_sws_1018] An input parameter that is a complex data type (i.e. a record or an array) or is a string shall be passed by reference.

  Note that AUTOSAR defines a string as a primitive data type yet due to its inherent size it would be inefficient to pass by value and is therefore treated the same as a complex data type.

- **"Out" Parameters**

  [rte_sws_1019] All output parameters shall be passed by reference, irrespective of their type.

- **"In/Out" Parameters**

  [rte_sws_1020] All bi-directional parameters (i.e. both input and output) shall be passed in by reference irrespective of their type.

5.2.6.6 Error Handling

In RTE, error and status information is defined with the data type `Std_ReturnType`, see Section 5.5.1.

It is possible to distinguish between infrastructure errors and application errors. Infrastructure errors are caused by a resource failure or an invalid input parameter. Infrastructure errors usually occur in the basic software or hardware along the communication path of a data element. Application errors are reported by a SW-C or by AUTOSAR services. RTE has the capability to treat application errors that are forwarded

- by return value in client server communication or
- by signal invalidation in sender receiver communication with isQueued set to false.

Errors that are detected during an RTE API call are notified to the caller using the API's return value.

[rte_sws_1034] Error states (including 'no error') shall only be passed as return value of the RTE API to the AUTOSAR SW-C.

Requirement rte_sws_1034 ensures that, irrespective of whether the API is blocking or non-blocking, the error is collected at the same time the data is made available to the caller thus ensuring that both items are accessed consistently.

Certain RTE API calls operate asynchronously from the underlying communication mechanism. In this case, the return value from the API indicates only errors detected during that API call. Errors detected after the API has terminated are returned using a different mechanism rte_sws_1111. RTE also provides an 'implicit' API for direct access to virtually shared memory. This API does not return any errors. The underlying communication is decoupled. Instead, an API is provided to pick up the current status of the corresponding data element.

5.2.6.7 Success Feedback

The RTE supports the notification of results of transmission attempts to an AUTOSAR software-component.

The `Rte_Feedback` API rte_sws_1083 can be configured to return the transmission result as either a blocking or non-blocking API or via activation of a runnable entity.

5.2.7 Unconnected Ports

[rte_sws_1329] The RTE shall handle both require and provide ports that are not connected.

The API calls for unconnected ports are specified to behave as if the port was connected but the remote communication point took no action.

5.2.7.1 Data Elements

5.2.7.1.1 Explicit Communication

[rte_sws_1330] A non-blocking `Rte_Read` API for an unconnected require port typed by a SenderReceiverInterface shall return RTE_E_OK code as if a sender was connected but did not transmit anything.

Requirement rte_sws_1330 applies to elements with ‘data’ semantics (isQueued = false) and therefore “last is best” semantics. This means that the initial value will be returned.

[rte_sws_1331] A blocking `Rte_Receive` API for an unconnected require port typed by a SenderReceiverInterface shall return RTE_E_TIMEOUT immediately without waiting for expiry of the timeout.

[rte_sws_1336] A non-blocking `Rte_Receive` API for an unconnected require port typed by a SenderReceiverInterface shall return RTE_E_NO_DATA immediately.

The existence of blocking and non-blocking `Rte_Read` and `Rte_Receive` API calls is controlled by the presence of DataReceivePoints, DataReceiveEvents and WaitPoints within the SW-C description rte_sws_1288, rte_sws_1289 and rte_sws_1290

[rte_sws_1344] A blocking `Rte_Feedback` API for a DataElementPrototype of an unconnected provide port shall return RTE_E_OK immediately.

[rte_sws_1345] A non-blocking `Rte_Feedback` API for a DataElementPrototype of an unconnected provide port shall return RTE_E_OK immediately.

The existence of blocking and non-blocking `Rte_Feedback` API is controlled by the presence of DataSendPoints, DataSendCompletedEvents and WaitPoints within the SW-C description for a DataElementPrototype with acknowledgement enabled, see rte_sws_1283, rte_sws_1284, rte_sws_1285 and rte_sws_1286.

[rte_sws_1332] The `Rte_Send` or `Rte_Write` API for an unconnected provide port typed by a SenderReceiverInterface shall discard the input parameters and return RTE_E_OK.

The existence of `Rte_Send` or `Rte_Write` is controlled by the presence of DataSend-Points within the SW/C description rte_sws_1280 and rte_sws_1281.

[rte_sws_3783] The `Rte_Invalidate` API for an unconnected provide port typed by a SenderReceiverInterface shall return RTE_E_OK.

The existence of `Rte_Invalidate` is controlled by the presence of DataSendPoints within the SW/C description for a DataElementPrototype which is marked as invalidatable and has canInvalidate enabled rte_sws_1282.

5.2.7.1.2 Implicit Communication

[rte_sws_1346] An `Rte_IRead` API for an unconnected require port typed by a Sender-ReceiverInterface shall return the initial value.

The existence of `Rte_IRead` is controlled by the presence of DataReadAccess in the SW-C description rte_sws_1301.

[rte_sws_1347] An `Rte_IWrite` API for an unconnected provide port typed by a Sender-ReceiverInterface shall discard the written data.

The existence of `Rte_IWrite` is controlled by the presence of DataWriteAccess in the SW-C description rte_sws_1302.

[rte_sws_3784] An `Rte_IInvalidate` API for an unconnected provide port typed by a SenderReceiverInterface shall perform no action.

The existence of `Rte_IInvalidate` is controlled by the presence of DataWriteAccess in the SW-C description for a DataElementPrototype which is marked as invalidatable and has canInvalidate enabled rte_sws_3801.

[rte_sws_3785] An `Rte_IStatus` API for an unconnected require port typed by a SenderReceiverInterface shall return RTE_E_OK.

The existence of `Rte_IStatus` is controlled by the presence of DataReadAccess in the SW-C description for a DataElementPrototype with `data element outdated` notification or `data element invalidation` rte_sws_2600.

5.2.7.2 Mode Ports

For the mode user an unconnected mode port behaves as if it was connected to a mode manager that never sends a mode switch notification.

[rte_sws_2638] A `Rte_Mode` API for an unconnected mode port of a mode user shall return the initial state.

[rte_sws_2639] Regarding the modes of an unconnected mode port of a mode user, the mode disabling dependencies on the initial mode shall be permanently active and the mode disabling dependencies on all other modes shall be inactive.

[rte_sws_2640] Regarding the modes of an unconnected mode port of a mode user, RTE will only generate a ModeSwitchEvent for entering the initial mode which occurs directly after startup.

[rte_sws_2641] The `Rte_Switch` API for an unconnected mode port of the mode manager shall discard the input parameters and return RTE_E_OK.

[rte_sws_2642] A blocking or non blocking `Rte_Feedback` API for an unconnected mode port of the mode manager shall return `RTE_E_OK` immediately.

5.2.7.3 Client-Server

[rte_sws_1333] The `Rte_Result` API for an unconnected asynchronous require port typed by a ClientServerInterface with a WaitPoint for the AsynchronousServerCallReturnsEvent shall return RTE_E_TIMEOUT immediately without waiting for expiry of the timeout.

[rte_sws_1337] The `Rte_Result` API for an unconnected asynchronous require port typed by a ClientServerInterface without a WaitPoint for the AsynchronousServerCallReturnsEvent shall return RTE_E_NO_DATA immediately.

[rte_sws_1334] An asynchronous `Rte_Call` API for an unconnected require port typed by a ClientServerInterface shall return RTE_E_OK immediately.

[rte_sws_1338] A synchronous `Rte_Call` API for an unconnected require port typed by a ClientServerInterface shall return RTE_E_TIMEOUT immediately without waiting for expiry of the timeout.

There is no analogous requirement for handling unconnected servers since the RTE need take no action to ensure that it is not invoked.

5.2.8 Non-identical ports

Two ports are permitted to be connected provided that they are characterized by compatible, but not necessarily identical, interfaces. For the full definition of whether two interfaces are compatible, see the System Template.

[rte_sws_1368] The RTE generator must report an error if two connected ports are connected by incompatible interfaces.

A significant issue in determining whether two interfaces are compatible is that the interface characterizing the require port may be a strict subset of the interface characterizing the provide port. This means that there may be provided data elements or operations for which there is no corresponding element in the require port. This can be imagined as a multi-strand wire between the two ports (the assembly connector) where each strand represents the connection between two data elements or operations, and where some of the strands from the ‘provide’ end are not connected to anything at the ‘require’ end.

Define, for the purposes of this section, an “unconnected element” as a data element or operation that occurs in the provide interface, but for which no corresponding data element or operation occurs in a particular R-Port’s interface.

[rte_sws_1369] For each data element or operation within the provide interface, every connected requirer with an “unconnected element” must be treated as if it were not connected.

Note that requirement rte_sws_1369 means that in the case of a 1:n Sender-Receiver the `Rte_Write` call may transmit to some but not all receivers. Similarly, there may be some clients that cannot write into a server’s queue.

The extreme is if all connected requirers have an “unconnected element”:

[rte_sws_1370] For a data element or operation in a provide interface for which *all* connected R-Ports have an “unconnected element”, the generated Send or Write API must act as if the port were unconnected.

See Section 5.2.7 for the required behaviour in this case.

5.3 RTE Modules

Figure 5.1 defines the relationship between header files and how those files are included by modules implementing AUTOSAR software-components and by general, non-component, code.

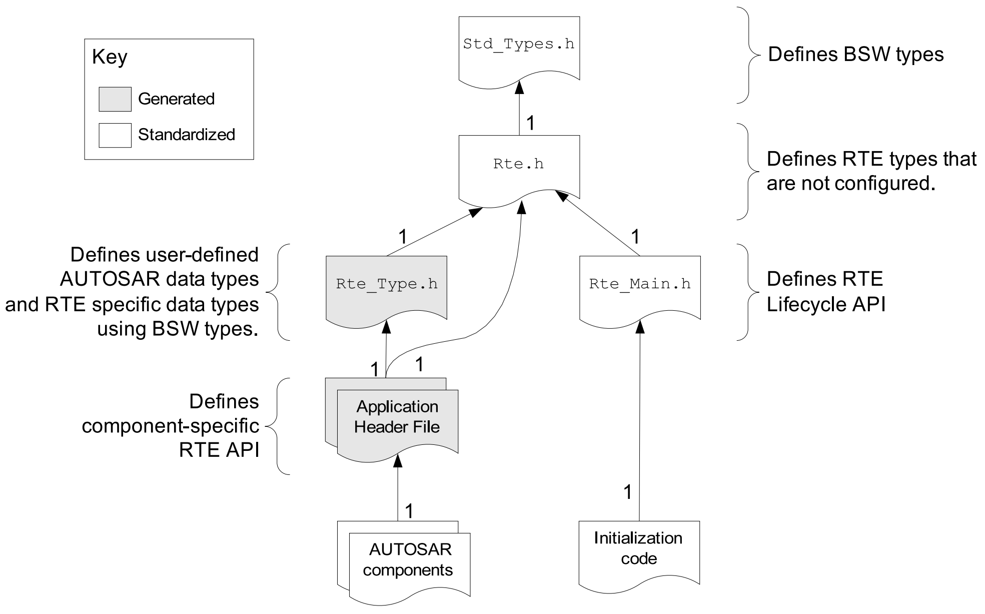

Key
Generated
Standardized
Std_Types.h
Defines BSW types
1
Rte.h
Defines RTE types that are not configured.
1
1
Defines user-defined AUTOSAR data types and RTE specific data types using BSW types.
Rte_Type.h
Rte_Main.h
Defines RTE Lifecycle API
1
1
Defines component-specific RTE API
Application Header File
1
1
AUTOSAR components
Initialization code

Figure 5.1: Relationships between RTE Header Files

The output of an RTE generator can consist of both generated code and configuration for “library” code that may be supplied as either object code or source code. Both configured and generated code reference standard definitions that are defined in one of two standardized header files; the *RTE Header File* and the *Lifecycle Header File*.

The relationship between the RTE header file, application header files, the lifecycle header file and AUTOSAR software-components is illustrated in Figure 5.1.

5.3.1 RTE Header File

The RTE header file defines fixed elements of the RTE that do not need to be generated or configured for each ECU.

[rte_sws_1157] For C/C++ AUTOSAR software-components, the name of the RTE header file shall be `Rte.h`.

Typically the contents of the standardized header file are fixed for any particular implementation and therefore it is not created by the RTE generator. However, customization for each generated RTE is not forbidden.

[rte_sws_1164] The RTE header file shall include the file `Std_Types.h`.

The file `Std_Types.h` is the standard AUTOSAR file [13] that defines basic data types including platform specific definitions of unsigned and signed integers and provides access to the compiler abstraction.

[rte_sws_3788] The RTE header file shall include the file `MemMap.h`.

The contents of the RTE header file are not restricted to standardized elements that are defined within this document – it can also contain definitions specific to a particular implementation.

5.3.2 Lifecycle Header File

The Lifecycle header file defines the two RTE Lifecycle API calls `Rte_Start` and `Rte_Stop` (see Section 5.8).

[rte_sws_1158] For C/C++ AUTOSAR software-components, the name of the lifecycle header file shall be `Rte_Main.h`.

[rte_sws_1159] The lifecycle header file shall include the *RTE header file*.

5.3.3 Application Header File

The application header file [RTE00087] is central to the definition of the RTE API. An application header file defines the RTE API and any associated data structures that are required by the RTE implementation. But the application header file is not allowed to create objects in memory.

[rte_sws_1000] The RTE generator shall create an application header file for each component type defined in the input.

[rte_sws_3786] The application header file shall not contain code that creates objects in memory.

Due to the restriction rte_sws_5034 it is only allowed to have exactly one InternalBehavior for each component type.

RTE generation consists of two phases; an initial “RTE Contract” phase and a second “RTE Generation” phase (see Section 2.3). Object-code components are compiled after the first phase of RTE generation and therefore the application header file should conform to the form of definitions defined in Sections 5.4.1 and 5.5.2. In contrast, source-code components are compiled after the second phase of RTE generation and therefore the RTE generator produces an optimized application header file based on knowledge of component instantiation and deployment.

5.3.3.1 File Name

[rte_sws_1003] The name of the application header file shall be formed by prefixing the AUTOSAR software-component type name with `Rte_` and appending the result with `.h`.

Example 5.8

The following declaration in the input XML:

```
<ATOMIC-SOFTWARE-COMPONENT-TYPE>
    <SHORT-NAME>Source</SHORT-NAME>
</ATOMIC-SOFTWARE-COMPONENT-TYPE>
```

should result in the application header file `Rte_Source.h` being generated.

The component type name is used rather than the component instance name for two reasons; firstly the same component code is used for all component instances and, secondly, the component instance name is an internal identifier, and should not appear outside of generated code.

5.3.3.2 Scope

[rte_sws_1004] The application header file for a component shall contain only information relevant to that component.

Requirement rte_sws_1004 means that compile time checks ensure that a component that uses the application header file only accesses the generated data structures and functions to which it has been configured. Any other access, e.g. to fields not defined in the customized data structures or RTE API, will fail with a compiler error [RTE00017].

[rte_sws_1005] The application header file shall be valid for both C and C++ source.

Requirement rte_sws_1005 is met by ensuring that all definitions within the application header file are defined using C linkage if a C++ compiler is used.

[rte_sws_3709] All definitions within in the application header file shall be preceded by the following fragment;

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

[rte_sws_3710] All definitions within the application header file shall be suffixed by the following fragment;

```
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
```

The definitions of the RTE API contained in the application header file can be optimized during the “RTE Generation” phase when the mapping of software-components

```
#include <Rte_c1.h>

void
runnable_entry(Rte_Instance self)
{
  /* ... server code ... */
}
```

Figure 5.2: Skeleton server runnable entity

to ECUs and the communication matrix is known. Consequently multiple application header files must not be included in the same source module to avoid conflicting definitions of the RTE API definitions that the files contains.

[rte_sws_1001] A source module that implements an AUTOSAR software-component (whether completely or partially) shall include exactly one application header file.

Figure 5.2 illustrates the code structure for the declaration of the entry point of a runnable entity that provides the implementation for a ServerPort in component c1. The RTE generator is responsible for creating the API and tasks used to execute the server and the symbol name of the entry point is extracted from the attribute symbol of the runnable entitiy. The example shows that the first parameter of the entry point function is the software-component's instance handle rte_sws_1016.

Figure 5.2 includes the component-specific application header file `Rte_c1.h` created by the RTE generator. The RTE generator will also create the supporting data structures and the task body to which the runnable is mapped.

The RTE is also responsible for preventing conflicting concurrent accesses when the runnable entity implementing the server operation is triggered as a result of a request from a client received via the communication service or directly via inter-task communication.

5.3.3.3 File Contents

Multiple application header file must not be included in the same module rte_sws_1001 and therefore the file contents should contain a mechanism to enforce this requirement.

[rte_sws_1006] An application header file shall include the following mechanism before any other definitions.

```
#ifdef RTE_APPLICATION_HEADER_FILE
#error Multiple application header files included.
#endif /* RTE_APPLICATION_HEADER_FILE */
#define RTE_APPLICATION_HEADER_FILE
```

The RTE uses an instance handle to identify different instances of the same component type. The definition of the instance handle type rte_sws_1148 is unique to each component type and therefore should be included in the application header file.

[rte_sws_1007] The application header file shall define the type of the instance handle for the component.

All runnable entities for a component are passed the same instance handle type (as the first formal parameter rte_sws_1016) and can therefore use the same type definition from the component's application header file.

[rte_sws_1263] The application header file shall include the *AUTOSAR Types Header File*.

The name of the AUTOSAR Types Header File is defined in Section 5.3.4.

The application header file also includes a prototype for each runnable entity entry point (rte_sws_1132) and the API mapping (rte_sws_1274).

5.3.3.3.1 RTE-Component Interface

The application header file defines the "interface" between a component and the RTE. The interface consists of the RTE API for the component and the prototypes for runnable entities. The definition of the RTE API requires that both relevant data structures and API calls are defined.

The data structures required to support the API are defined in the RTE Types header file rte_sws_3713. This enables the definitions to be available to multiple modules to support direct function invocation.

The data structure types are declared in the RTE Types file whereas the instances are defined in the generated RTE. The necessary data structures for object-code software-components are defined 5.5.2.

[rte_sws_1009] The application header file shall define the mapping from the RTE API to the generated API functions that are generated/configured for the component.

The RTE generator is required rte_sws_1004 to limit the contents of the application header file to only that information that is relevant to that component type. This requirement includes the definition of the API mapping.

[rte_sws_1276] Only RTE API calls that are valid for the particular software-component type shall be defined within the component's application header file.

Requirement rte_sws_1276 ensures that attempts to invoke invalid API calls will be rejected as a compile-time error [RTE00017].

5.3.3.3.2 Port-Defined Argument Values

The input to the RTE generator can include lists of port-defined argument values for runnable entities to be invoked by an OperationInvokedEvent. In order to allow the implementation of a server invocation as a direct function call for object-code components even if port-defined argument values are specified, the RTE generator shall declare standardized symbolic names for the port-defined argument values of each client that are then used in the API mapping of the contract phase.

[rte_sws_3780] The symbolic name of a port-defined argument value shall be `Rte_-PDAV_<c>_<p>_<o>_<n>` where `<c>` is the name of the component type, `<p>` the name of the require port, `<o>` the operation prototype name and `<n>` the index of the port-defined argument value in the ordered port argument list, starting with 1.

[rte_sws_3779] The application header file shall declare the symbolic names of the port-defined argument values encountered in the input using the appropriate AUTOSAR data-type.

For C (or C++) an `extern` declaration must be used for to make visible the symbolic names of the port-defined argument values. The AUTOSAR data-types are defined in the AUTOSAR types header file – see Section 5.3.4.

5.3.4 AUTOSAR Types Header File

The AUTOSAR types header file defines RTE specific types derived either from the input configuration or from the RTE implementation.

The generated RTE can include zero or more AUTOSAR data types created from the definitions of AUTOSAR meta-model classes within the RTE generator's input. The available meta-model classes are defined by the AUTOSAR software-component template and include classes for defining integers, floats as well as "complex" data types such as records.

[rte_sws_1160] The RTE generator shall create the AUTOSAR Types header file defining the AUTOSAR data types and RTE implementation types.

The AUTOSAR data types header file should be output for "RTE Contract" and "RTE Generation" phases. RTE implementation types include the Component Data Structure (Section 5.4.2).

5.3.4.1 File Contents

[rte_sws_2648] The AUTOSAR Types header file shall include the definitions of all AUTOSAR data types irrespective of their use by the generated RTE.

This requirement ensures the availability of AUTOSAR data types for the internal use in AUTOSAR software components.

The types header file may need to define types in terms of BSW types (from the file `Std_Types.h`) or from the implementation specific RTE header file. However, since the RTE header file includes the file `Std_Types.h` already so only the RTE header file needs direct inclusion within the types header file.

[rte_sws_1163] The AUTOSAR Types header file shall include the *RTE header file*.

5.3.4.2 Primitive AUTOSAR Data Types

The AUTOSAR types file defines the mapping from primitive AUTOSAR data-types (defined in the XML) to programming language specific type definitions. The mapping from primitive AUTOSAR data-types to BSW standard types (as defined in `Std_Types.h` is defined in Table 5.1).

Requirement	Meta-type	Range	Base Type
[rte_sws_1175]	`CHAR-TYPE`	All encodings other than 'UTF-16'	`uint8`
[rte_sws_1215]	`CHAR-TYPE`	Encoding 'UTF-16'	`uint16`
[rte_sws_1176]	`STRING-TYPE`	Declaration, n is defined maximum length including zero terminator	`uint8[n]`
[rte_sws_1212]	`STRING-TYPE`	Function parameter	`uint8*`
[rte_sws_1177]	`INTEGER-TYPE`	[-128,127]	`sint8`
[rte_sws_1178]	`INTEGER-TYPE`	[-32768,32767]	`sint16`
[rte_sws_1179]	`INTEGER-TYPE`	[-2147483648,2147483647]	`sint32`
[rte_sws_1180]	`INTEGER-TYPE`	[0,255]	`uint8`
[rte_sws_1181]	`INTEGER-TYPE`	[0,65535]	`uint16`
[rte_sws_1182]	`INTEGER-TYPE`	[0,4294967295]	`uint32`
[rte_sws_1183]	`OPAQUE-TYPE`	Bit length 1..8	`uint8`
[rte_sws_1184]	`OPAQUE-TYPE`	Bit length 9..16	`uint16`
[rte_sws_1185]	`OPAQUE-TYPE`	Bit length 17..32	`uint32`
[rte_sws_1186]	`REAL-TYPE`	Encoding `single`	`float32`
[rte_sws_1187]	`REAL-TYPE`	Encoding `double`	`float64`
[rte_sws_1188]	`BOOLEAN-TYPE`	N/A	`boolean`

Table 5.1: C/C++ mapping from primitive AUTOSAR data-types

An integer type is defined using either an *open* or *closed* interval – a closed interval includes its endpoints whereas an open interval does not. For simplicity, Table 5.1 defines mappings for integer types using *closed* intervals.

[rte_sws_1265] Where the range expressed in a type definition is not exactly the same as a range defined in Table 5.1, the RTE generator shall select the smallest suitable base type.

Example 5.9 describes the definition of an 11-bit unsigned integer type in terms of a 16-bit base type.

Example 5.9

The following declaration of the user-defined type `UInt11` in the input XML:

```
<INTEGER-TYPE>
  <SHORT-NAME>UInt11</SHORT-NAME>
  <LOWER-LIMIT>
    <INTERVAL-TYPE>CLOSED</INTERVAL-TYPE>
    <VALUE>0</VALUE>
  </LOWER-LIMIT>
  <UPPER-LIMIT>
    <INTERVAL-TYPE>OPEN</INTERVAL-TYPE>
    <VALUE>2048</VALUE>
  </UPPER-LIMIT>
</INTEGER-TYPE>
```

Should result in a mapping to the base type `uint16` and the following type definition;

```
typedef uint16 UInt11;
```

[rte_sws_1214] An attempt to declare a type with a range which cannot be represented by a base type from Table 5.1 shall be rejected by the RTE generator.

Table 5.1 applies to the standard AUTOSAR types as well as user-defined types and primitive data-types with semantics. Using the requirements defined in Table 5.1 the standard AUTOSAR primitive types are mapped as follows:

AUTOSAR Type	**BSW Type**
`UInt4`	`uint8`
`SInt4`	`sint8`
`UInt8`	`uint8`
`SInt8`	`sint8`
`UInt16`	`uint16`
`SInt16`	`sint16`
`UInt32`	`uint32`
`SInt32`	`sint32`
`Float_with_NaN`	`float32`
`Float`	`float32`
`Double_with_NaN`	`float64`
`Double`	`float64`
`Boolean`	`boolean`
`Char8`	`uint8`
`Char16`	`uint16`

Table 5.2: C/C++ mapping for standard AUTOSAR data-types

5.3.4.3 Complex AUTOSAR Data Types

In addition to the primitive data-types defined in the previous section, it is also necessary for the RTE generator to define complex data-types; arrays and records.

An array definition needs three pieces of information; the array base type, the array name and the number of elements.

[rte_sws_1189] An `ARRAY-TYPE` data-type shall be declared as `typedef <type> <name>[n]` where `<type>` is the base type, `<name>` the data-type name and n the number of elements.

Example 5.10

The array data-type declaration;

```
<ARRAY-TYPE>
  <SHORT-NAME>array</SHORT-NAME>
  <DESC>array of myInt values</DESC>
  <INTEGER-TYPE-REF>myInt</INTEGER-TYPE-REF>
  <NUMBER-OF-ELEMENTS>2</NUMBER-OF-ELEMENTS>
</ARRAY-TYPE>
```

Produces the following type definition;

```
typedef myInt array[2];
```

ANSI C does not allow a type declaration to have zero elements and therefore we require that the “number of elements” to be a positive integer.

[rte_sws_1190] The number of elements of an `ARRAY_TYPE` data type shall be an integer that is $\geqslant 1$.

A record definition contains references to one or more data elements with a base type for each element. A record definition is recursive; a data element can include a type reference that is itself another record definition.

[rte_sws_1191] A `RECORD-TYPE` data-type shall be declared as `typedef struct { <elements> } <name>` where `<elements>` is the record element specification and `<name>` the data-type name.

ANSI C does not allow a `struct` to have zero elements and therefore we require that a record include at least one element.

[rte_sws_1192] A record shall include at least one element.

Example 5.11

The record data-type declaration;

```
<RECORD-TYPE>
  <SHORT-NAME>R2</SHORT-NAME>
  <DATA-PROTOTYPES>
    <DATA-ELEMENT-PROTOTYPE>
      <SHORT-NAME>Abc</SHORT-NAME>
      <BOOLEAN-TYPE-TREF>myBool</BOOLEAN-TYPE-TREF>
    </DATA-ELEMENT-PROTOTYPE>
    <DATA-ELEMENT-PROTOTYPE>
      <SHORT-NAME>Def</SHORT-NAME>
      <INTEGER-TYPE-TREF>myInt</INTEGER-TYPE-TREF>
    </DATA-ELEMENT-PROTOTYPE>
  </DATA-PROTOTYPES>
</RECORD-TYPE>
```

Produces the following type definition;

```
typedef struct {
   myBool  Abc;
   myInt   Def;
} R2;
```

5.3.4.4 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1161] The name of the AUTOSAR types header file shall be `Rte_Type.h`.

[rte_sws_1162] Within the AUTOSAR types header file, each data type shall be defined using `typedef`.

A `typedef` is used when defining a new data type instead of a `#define` even though C only provides weak type checking since other static analysis tools can then be used to overlay strong type checking onto the C before it is compiled and thus detect type errors before the module is even compiled.

5.3.5 VFB Tracing Header File

The VFB Tracing Header File defines the configured VFB Trace events.

[rte_sws_1319] The VFB Tracing Header File shall be created by the RTE Generator during "RTE Generation" phase only.

The VFB Tracing Header file is included by the generated RTE and by the user in the module(s) that define the configured hook functions. The header file includes prototypes for the configured functions to ensure consistency between the invocation by the RTE and the definition by the user.

5.3.5.1 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1250] The name of the VFB Tracing Header File shall be `Rte_Hook.h`.

5.3.5.2 File Contents

[rte_sws_1251] The VFB Tracing header file shall include the *RTE Configuration file* (Section 5.3.6).

[rte_sws_1357] The VFB Tracing header file shall include the *AUTOSAR Types Header file* (Section 5.3.4).

[rte_sws_1320] The VFB Tracing header file shall contain the following code immediately after the include of the RTE Configuration file.

```
#ifndef RTE_VFB_TRACE
#define RTE_VFB_TRACE (0)
#endif /* RTE_VFB_TRACE */
```

Requirement rte_sws_1320 enables VFB tracing to be globally enabled/disabled within the RTE Configuration file and ensures that it defaults to 'disabled'.

[rte_sws_1236] For each trace event hook function defined in Section 5.10.2, the RTE generator shall define the following code sequence in the VFB Tracing header file:

```
#if defined(<trace event>) && (RTE_VFB_TRACE == 0)
#undef <trace event>
#endif
#if defined(<trace event>)
#undef <trace event>
extern void <trace event>(<params>);
#else
#define <trace event>(<params>) ((void)(0))
#endif /* <trace event> */
```

In the example above, `<trace event>` is the name of trace event hook function and `<params>` is the formal parameter list of the trace event hook function prototype as defined in Section 5.10.2.

The code fragment within rte_sws_1236 benefits from a brief analysis of its structure. The first `#if` block ensures that an individually configured trace event in the RTE Configuration file rte_sws_1324 is disabled if tracing is globally disabled rte_sws_1323. The second `#if` block emits the prototype for the hook function only if enabled in the RTE Configuration file and thus ensures that only configured trace events are prototyped. The `#undef` is required to ensure that the trace event function is invoked as a function by the generated RTE. The `#else` block comes into effect if the trace event is disabled, either individually rte_sws_1325 or globally, and ensures that it has no run-time effect rte_sws_1235. Within the `#else` block the definition to `((void)(0))` enables the hook function to be used within the API Mapping in a comma-expression.

An individual trace event defined in Section 5.10.2 actually defines a class of hook functions. A member of the class is created for each RTE object created (e.g. for each API function, for each task) and therefore an individual trace event may give rise to many hook function definitions in the VFB Tracing header file.

Example 5.12

Consider an API call `Rte_Write_p1_a` for an instance of SW-C `c`. This will result in two trace event hook functions being created by the RTE generator:

```
Rte_WriteHook_c_p1_a_Start
```

and

```
Rte_WriteHook_c_p1_a_Return
```

5.3.6 RTE Configuration Header File

The RTE Configuration Header file contains user definitions that affect the behaviour of the generated RTE.

The directory containing the required RTE Configuration header file should be included in the compiler's include path when using the VFB tracing header file.

5.3.6.1 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1321] The name of the RTE Configuration Header File shall be `Rte_Cfg.h`.

5.3.6.2 File Contents

[rte_sws_1322] The RTE generator shall globally enable VFB tracing when `RTE_VFB_TRACE` is defined in the RTE configuration header file as a non-zero integer.

Note that, as observed in Section 5.10, VFB tracing enables debugging of software components, not the RTE itself. For this reason, `RTE_VFB_TRACE` is used in preference to `RTE_DEV_ERROR_DETECT`.

[rte_sws_1323] The RTE generator shall globally disable VFB tracing when `RTE_VFB_TRACE` is defined in the RTE configuration header file as `0`.

As well as globally enabling or disabling VFB tracing, the RTE Configuration header file also configures those individual VFB tracing events that are *enabled*.

[rte_sws_1324] The RTE generator shall enable VFB tracing for a given hook function when there is a `#define` in the RTE configuration header file for the hook function name and tracing is globally enabled.

Note that the particular value assigned by the `#define`, if any, is not significant.

[rte_sws_1325] The RTE generator shall disable VFB tracing for a given hook function when there is no `#define` in the RTE configuration header file for the hook function name even if tracing is globally enabled.

Example 5.13

Consider the trace events from Example 5.12. The trace event for API start is enabled by the following definition;

```
#define Rte_WriteHook_i1_p1_a_Start
```

And the trace event for API termination is enabled by the following definition;

```
#define Rte_WriteHook_i1_p1_a_Return
```

5.3.7 Generated RTE

Figure 5.1 defines the relationship between generated and standardized header files. It is **not** necessary to standardize the relationship between the C module, `Rte.c`, and the header files since when the RTE is generated the application header files are created anew along with the RTE. This means that details of which header files are included by `Rte.c` can be left as an implementation detail.

5.3.7.1 Header File Usage

[rte_sws_1257] In compatibility mode, the Generated RTE module shall include `os.h`.

[rte_sws_3794] In compatibility mode, the generated RTE module shall include `Com.h`.

[rte_sws_1279] In compatibility mode, the Generated RTE module shall include `Rte.h`.

[rte_sws_1326] In compatibility mode, the Generated RTE module shall include the VFB Tracing header file.

Figure 5.3 provides an example of how the RTE header and generated header files could be used by a generated RTE.

In the example in Figure 5.3, the generated RTE C module requires access to the data structures created for each AUTOSAR software-component and therefore includes each application header file[2]. In the example, the generated RTE also includes the RTE header file and the lifecycle header file in order to obtain access to RTE and lifecycle related definitions.

5.3.7.2 C/C++

The following requirements apply to RTEs generated for C and C++.

[rte_sws_1169] The name of the C module containing the generated RTE shall be `Rte.c`.

[2]The requirement that a software module include at most one application header file applies only to modules that actually implement a software-component and therefore does not apply to the generated RTE.

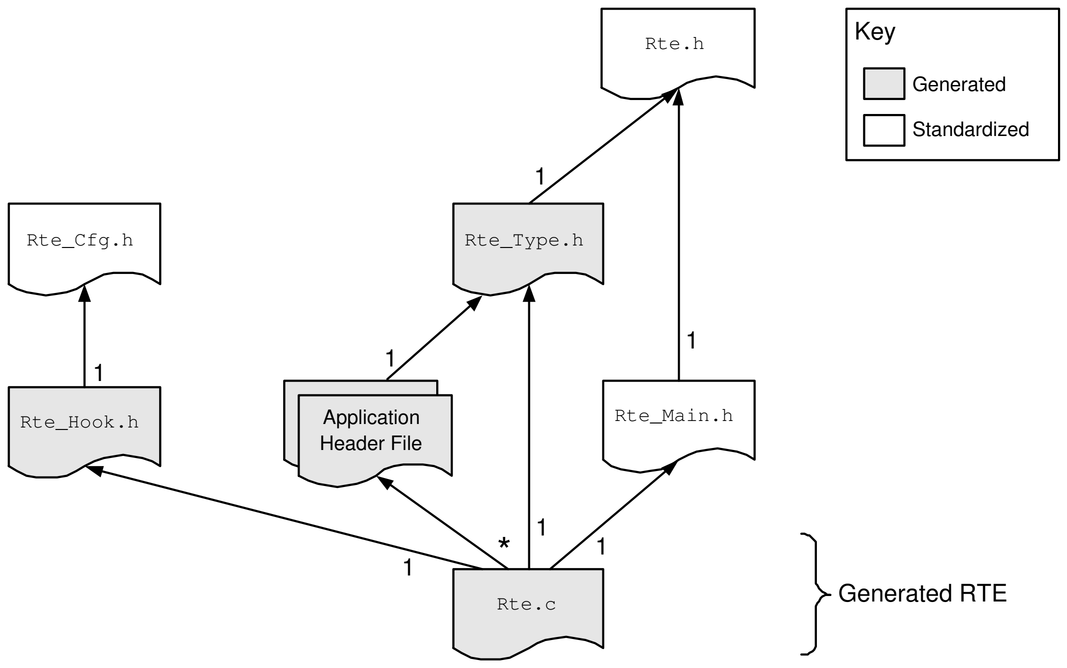


Figure 5.3: Example of header file use by the generated RTE.

An RTE that includes configured code from an object-code or source-code library may use additional modules.

5.3.7.3 File Contents

By its very nature the contents of the generated RTE is largely vendor specific. It is therefore only possible to define those common aspects that are visible to the “outside world” such as the names of generated APIs and the definition of component data structures that apply any operating mode.

5.3.7.3.1 Component Data Structures

The *Component Data Structure* (Section 5.4.2) is a per-component data type used to define instance specific information required by the generated RTE.

[rte_sws_3711] The generated RTE shall contain an instance of the relevant Component Data Structure for each software-component instance on the ECU for which the RTE is generated.

[rte_sws_3712] The name of a Component Data Structure instantiated by the RTE generator shall be `Rte_Instance_<name>` where `<name>` is an automatically generated name, created in some manner such that all instance data structure names are unique.

The software component instance name referred to in rte_sws_3712 is never made visible to the users of the generated RTE. There is therefore no need to specify the precise form that the unique name takes. The `Rte_Instance_` prefix is mandated in order to ensure that no name clashes occur and also to ensure that the structures are readily identifiable in map files, debuggers, etc.

5.3.7.3.2 Generated API

[rte_sws_1266] The RTE module shall define the generated functions that will be invoked when an AUTOSAR software-component makes an RTE API call.

The semantics of the generated functions are not defined (since these will obviously vary depending on the RTE API call that it is implementing) nor are the implementation details (which are vendor specific). However, the names of the generated functions defined in Section 5.2.6.1.

The signature of a generated function is the same as the signature of the relevant RTE API call (see Section 5.6) with the exception that the instance handle can be omitted since the generated function is applicable to a specific software-component instance.

5.3.7.3.3 Callbacks

In addition to the generated functions for the RTE API, the RTE module includes callbacks invoked by COM when signal events (receptions, transmission acknowledgement, etc.) occur.

[rte_sws_1264] The RTE module shall define COM callbacks for relevant signals.

The required callbacks are defined in Section 5.9.2.

[rte_sws_3795] The RTE generator shall generate a separate header file containing the prototypes of the COM callback functions.

[rte_sws_3796] The name of the header file containing the COM callback prototypes shall be `Rte_Cbk.h` in a C/C++environment.

5.3.7.3.4 Task bodies

The RTE module define task bodies for tasks created by the RTE generator only in compatibility mode.

[rte_sws_1277] In compatibility mode rte_sws_1257, the RTE module shall define all task bodies created by the RTE generator.

Note that in vendor mode it is assumed that greater knowledge of the OS is available and therefore the above requirement does *not* apply so that specific optimizations, such as creating each task in a separate module, can be applied.

5.3.7.3.5 Lifecycle API

[rte_sws_1197] The RTE module shall define the RTE lifecycle API.

The RTE lifecycle API is defined in Section 5.8.

5.3.7.3.6 Port-Defined Argument Values

The RTE generator is required to declare symbolic names for the port-defined argument values encountered in the input in the application header file rte_sws_3780. Additionally, these constant values must be defined for those symbolic names in the RTE module.

[rte_sws_3781] The generated RTE shall define the constant values for the port-defined argument values encountered in the input using the appropriate AUTOSAR data-type.

The mapping of the port-defined argument values to specific ROM regions is specified in [11]. The generated RTE has to be compliant with the mechanisms defined there.

5.3.7.4 Configuration Data

When constructing the initializers for the component data structure, the RTE generator needs to import a label for each configuration data section. The label is extracted from the input information:

[rte_sws_1335] The RTE shall extract from the ECU configuration the location of the characteristic data for each configuration data section for each software component instance, and ensure that this value is returned for the correct `Rte_Ctc_xxx` API for the particular software component instance.

The source of the configuration data needs to be an input requirement (rather than using a constructed label) to permit multiple software component instances to share a configuration data set, and because it would be very error-prone for an integrator to attempt to identify a particular software component instance based upon a constructed name.

5.3.7.5 Reentrancy

[rte_sws_1172] All code generated by an RTE generator, or invoked by generated code, that can be subject to concurrent execution shall be reentrant.

The requirement for reentrancy can be overridden if the generated code is not subject to concurrent execution, for example, if protected by a data consistency mechanism to ensure that access to critical regions is serialized.

5.4 RTE Data Structures

Object-code software components are compiled against an application header file created during the “RTE Contract” phase but are linked against an RTE (and application header file) created during the “RTE Generation” phase. When generated in compatibility mode, an RTE has to work for object-code components compiled against an application header file created in compatibility mode, even if the application header file was created by a different RTE generator. It is thus necessary to define the data structures and naming conventions for the compatibility mode to ensure that the object-code is compatible with the generated RTE. An RTE generated in vendor mode only has to work for those object-code components that were compiled against application header files created in vendor mode by a compatible RTE generator (which in general would mean an RTE generator supplied by the same vendor).

The use of standardized data structures imposes tight constraints on the RTE implementation and therefore restricts the freedom of RTE vendors to optimize the solution of object-code components but has the advantage that RTE generators from different vendors can be used to compile an object-code software-component and to generate the RTE. No such restrictions apply for the vendor mode. If an RTE generator operating in vendor mode is used for an object-code component in both phases, vendor-specific optimizations can be used.

Note that with the exception of data structures required for support object-code software components in compatibility mode, the data structures used for “RTE Generation” phase are not defined. This permits vendor specific API mappings and data structures to be used for a generated RTE without loss of portability.

The following definitions only apply to RTE generators operating in compatibility mode – in this mode the instance handle and the component data structure have to be defined even for those (object-code) software components for which multiple instantiation is forbidden to ensure compatibility.

5.4.1 Instance Handle

The RTE is required to support object-code components as well as multiple instances of the same AUTOSAR software-component mapped to an ECU [RTE00011]. To minimise memory overhead all instances of a component on an ECU share code [RTE00012] and therefore both the RTE and the component instances require a means to distinguish different instances.

Support for both object-code components and multiple instances requires a level of indirection so that the correct generated RTE custom function is invoked in response to a component action. The indirection is supplied by the instance handle in combination with the API mapping defined in Section 5.2.6.

[rte_sws_1012] The component instance handle shall identify particular instances of a component.

The instance handle is passed to each runnable entity in a component when it is activated by the RTE as the first parameter of the function implementing the runnable entity rte_sws_1016. The instance handle is then passed back by the runnable entity to the RTE, as the first parameter of each direct RTE API call, so that the RTE can identify the correct component instance making the call. This scheme permits multiple instances of a component on the same ECU to share code.

The instance handle indirection permits the name of the RTE API call that is used within the component to be unique within the scope of a component as well as independent of the component's instance name. It thus enables object-code AUTOSAR software-components to be compiled before the final "RTE Generation" phase when the instance name is fixed.

[rte_sws_1013] For the RTE C/C++ API, any call that can operate on different instances of a component that supports multiple instantiation rte_sws_in_0004 shall have an instance handle as the first formal parameter.

[rte_sws_3806] If a component does not support multiple instantiation, the instance handle parameter shall be omitted in the RTE C/C++ API calls.

If the component does not support multiple instantiation, the name of the instance handle must be specified, since it is not passed to the API calls and runnable entities as parameters.

[rte_sws_3793] If a software component does not support multiple instantiation, the name of the instance handle shall be `Rte_Inst_<c>`, where <c> is the component type name.

The data type of the instance handle is defined in Section 5.5.2.

5.4.2 Component Data Structure

Different component instances share many common features - not least of which is support for shared code. However, each instance is required to invoke different RTE API functions and therefore the instance handle is used to access the component data structure that defines all instance specific data.

It is necessary to define the component data structure to ensure compatibility between the two RTE phases when operating in compatibility mode – for example, a "clever" compiler and linker may encode type information into a pointer type to ensure type-safety. In addition, the structure definition cannot be empty since this is an error in ANSI C.

[rte_sws_3713] The component data structure type shall be defined in the AUTOSAR Types Header file.

[rte_sws_3714] The type name of the component data structure shall be `Rte_CDS_<c>` where <c> is the component type name.

The members of the component data structure include function pointers. It is important that such members are not subject to run-time modification and therefore the component data structure is required to be placed in read-only memory.

[rte_sws_3715] All instances of the component data structure shall be defined as “const” (i.e. placed in read-only memory).

The elements of the component data structure are sorted into sections, each of which defines a logically related section. The sections defined within the component data structure are:

- **[rte_sws_3718]** Data Handles section.
- **[rte_sws_3719]** Per-instance Memory Handles section.
- **[rte_sws_1349]** Inter-runnable Variable Handles section.
- **[rte_sws_3720]** (per-instance) Configuration Data Handles section.
- **[rte_sws_3721]** Exclusive-area Handles section.
- **[rte_sws_3716]** Port API section.
- **[rte_sws_3717]** Inter Runnable Variable API section.
- **[rte_sws_3722]** Vendor specific section.

The order of elements within each section of the component data structure is defined as follows;

[rte_sws_3723] Section entries shall be sorted alphabetically (case sensitive, English rules) unless stated otherwise.

The sorting of entries is applied to each section in turn.

Note that there is *no* prefix associated with the name of each entry within a section; the component data structure as a whole has the prefix and therefore there is no need for each member to have the same prefix.

ANSI C does not permit empty structure definitions yet an instance handle is required for the RTE to function. Therefore if there are no API calls then a single dummy entry is defined for the RTE.

[rte_sws_3724] If all sections of the Component Data Structure are empty the Component Data Structure shall contain a `uint8` with name `_dummy`.

5.4.2.1 Data Handles Section

The data handles section is required to support the `Rte_IRead` and `Rte_IWrite` calls (see Section 5.2.4).

[rte_sws_3733] Data Handles shall be named `<re>`_<p>_<d> where `<re>` is the runnable entity name that reads/writes the data item, <p> the port name, <d> the data element or operation name.

[rte_sws_2608] The Data Handle shall be a pointer to a `Data Element with Status` if and only if the runnable has read access and either

- `data element outdated` notification or
- `data element invalidation`

is activated for this `data element`.

[rte_sws_2588] Otherwise, the data type for a Data Handle shall be a pointer to either a `Data Element without Status`.

See below for the definitions of these terms.

5.4.2.1.1 Data Element without Status

[rte_sws_1363] The data type for a "Data Element without Status" shall be named `Rte_DE_<dt>` where `<dt>` is the data element type.

[rte_sws_1364] A `Data Element without Status` shall be a structure containing a single member named `value`.

[rte_sws_2607] The `value` member of a `Data Element without Status` shall have the same data type as the corresponding DataElement.

Note that requirements rte_sws_1364 and rte_sws_2607 together imply that creating a variable of data type `Rte_DE_<dt>` allocates enough memory to store the data copy.

5.4.2.1.2 Data Element with Status

[rte_sws_1365] The data type for a "Data Element with Status" shall be named `Rte_DES_<dt>` where `<dt>` is the data element type.

[rte_sws_1366] A `Data Element with Status` shall be a structure containing two members.

[rte_sws_3734] The first member of each `Data Element with Status` shall be named 'value'

[rte_sws_2607] The value member of a `Data Element with Status` shall have the type of the corresponding DataElement.

[rte_sws_2589] The second member of each `Data Element with Status` shall be named 'status'.

[rte_sws_2590] The status member of a `Data Element with Status` shall be of the `Std_ReturnType` type.

[rte_sws_2609] The status member of a `Data Element with Status` shall contain the error status corresponding to the value member.

5.4.2.1.3 Usage

[rte_sws_1367] A definition for every required `Data Element with Status` and every `Data Element without Status` must be emitted in the *AUTOSAR Types Header File*.

The AUTOSAR Types Header File is defined in Section 5.3.4).

Example 5.14

Consider a `uint8` data element, a, of port `p` which is accessed using DataWriteAccess semantics by runnables `re1` and `re2` and DataReadAccess semantics by runnable `re2` within component c. `data element outdated` is defined for this DataElementPrototype.

The required data types within the *AUTOSAR Types Header File* would be:

```
typedef struct {
  uint8 value;
} Rte_DE_uint8;

typedef struct {
  uint8 value;
  Std_ReturnType status;
} Rte_DES_uint8;
```

The component data structure for c would also include:

```
Rte_DE_uint8*  re1_p_a;
Rte_DES_uint8* re2_p_a;
```

A software-component that is supplied as object-code or is multiply instantiated requires “general purpose” definitions of `Rte_IRead`, `Rte_IWrite`, and `Rte_IStatus` that use the data handles to access the data copies created within the generated RTE. For example:

```
#define Rte_IWrite_re1_p_a(s,v) ((s)->re1_p_a->value = (v))
#define Rte_IWrite_re2_p_a(s,v) ((s)->re2_p_a->value = (v))
#define Rte_IRead_re2_p_a(s,v)  ((s)->re2_p_a->value)
#define Rte_IStatus_re2_p_a(s)  ((s)->re2_p_a->status)
```

The definitions of `Rte_IRead`, `Rte_IWrite`, and `Rte_IStatus` are type-safe since an attempt to assign an incorrect type will be detected by the compiler.

For source code component that does **not** use multiple instantiation the definitions of `Rte_IRead`, `Rte_IWrite`, and `Rte_IStatus` can remain as above or vendor specific optimizations can be applied without loss of portability.

The values assigned to data handles within *instances* of the component data structure created within the generated RTE depend on the mapping of tasks and runnables – See Section 5.2.4.

5.4.2.2 Per-instance Memory Handles Section

The Per-instance Memory Section Handles section enables to access instance specific memory (sections).

[rte_sws_2301] The CDS shall contain a handle for each Per-instance Memory. This handle member shall be named `Pim_<name>` where `<name>` is the per-instance memory name.

The Per-instance Memory Handles are typed;

[rte_sws_2302] The data type of each Per-instance Memory Handle shall be a pointer to the type of the per instance memory that is defined in the RTE Types header file.

The RTE supports the access to the per-instance memories by the `Rte_Pim` API.

Example 5.15

Referring to the specification items rte_sws_2301 and rte_sws_2302 Example 5.4 can be extended –

with respect to the software-component header:

```
struct Rte_CDS_c {
  ...
  /* per-instance memory handle section */
  Rte_PimType_c_MyMemType *Pim_mem;

  ...
};

#define Rte_Pim_mem(s) ((s)->Pim_mem)
```

and in Rte.c:

```
Rte_PimType_c_MyMemType mem1;

const struct Rte_CDS_c Rte_Instance_c1 = {
  ...
  /* per-instance memory handle section */
  /* Rte_PimType_c_MyMemType Pim_mem */
  &mem1
  ...
};
```

5.4.2.3 Inter Runnable Variable Handles Section

Each runnable may require separate handling for the inter runnable variables that it accesses. The indirection required for explicit access to inter runnable variables is described in section 5.4.2.7. The inter runnable variable handles section within the component data structure contains pointers to the (shadow) memory of inter runnable variables that can be directly accessed with the implicit API macros. The inter runnable variable handles section does not contain pointers for memory to handle inter runnable variables that are accessed with explicit API only.

[rte_sws_2636] For each runnable and each inter runnable variable that is accessed implicitly by the runnable, there shall be exactly one inter runnable handle member within the component data structure and this inter runnable variable handle shall point to the (shadow) memory of the inter runnable variable for the runnable.

[rte_sws_1350] The name of each inter runnable variable handle member within the component data structure shall be `Irv_<re>_<name>` where `<name>` is the Inter-Runnable Variable short name and `<re>` is short name of the runnable name.

[rte_sws_1351] The data type of each inter runnable variable handle member shall be a pointer to the type of the inter runnable variable.

5.4.2.4 Configuration Data Handles Section

A software-component's instance handle defines a particular instance of a component and is therefore necessary to access the configuration data. The instance handle is passed as a parameter to the `Rte_CData` API. Configuration data is read-only and therefore no data consistency mechanism is required.

[rte_sws_3737] The name of each Configuration Data Handle entry shall be `Ctc_<name>` where `<name>` is the configuration data name.

The configuration data handles are typed;

[rte_sws_3738] The data type of each Configuration Data Handle entry shall be a pointer to the (per-instance) configuration data.

The RTE supports per-instance configuration data (a.k.a. *Characteristics*) through the `Rte_CData` API.

5.4.2.5 Exclusive-area handles Section

The Exclusive-area handles section includes exclusive areas that are accessed explicitly, using the RTE API, by the software-component.

[rte_sws_3739] The name of each Exclusive-area Handle entry shall be `<name>` where `<name>` is the Exclusive-area name.

[rte_sws_3740] The data type of each Exclusive-area Handle entry shall be a `void` pointer.

The generated RTE can use the `void` pointer in any way it requires since the semantics of the handle are defined by the generated RTE itself. For example, it could be cast to a resource handle suitable for passing to the operating system.

5.4.2.6 Port API Section

Port API section comprises zero or more *function references* within the component data structure type that defines all API functions that access a port and can be invoked by the software-component (instance).

[rte_sws_2616] The function table entries for port access shall be grouped by the port names into port data structures.

Each entry in the port API section of the component data structure is a “port data structure”.

[rte_sws_2617] The name of each port data structure in the component data structure shall be <p> where <p> is the port short-name.

[rte_sws_3731] The data type name for a port data structure shall be `struct Rte_PDS_<c>_<i>_<P/R>` where `<c>` is the component type name, `<i>` is the port interface name and ‘P’ or ‘R’ are literals to indicate provide or require ports respectively.

[rte_sws_3732] The port data structure type(s) shall be defined in the AUTOSAR types header file.

A port data structure type is defined for each port interface that types a port. Thus different ports typed by the same port interface structure share the same port data structure type.

[rte_sws_3730] A port data structure shall contain a function table entry for each API function associated with the port as referenced in table 5.3. Pure API macros, like `Rte_IRead` and other implicit API functions, do not have a function table entry.

API function	reference
`Rte_Send_<p>_<d>`	5.6.4
`Rte_Switch_<p>_<m>`	5.6.4
`Rte_Write_<p>_<d>`	5.6.4
`Rte_Invalidate_<p>_<d>`	5.6.5
`Rte_Feedback_<p>_<d>`	5.6.6
`Rte_Read_<p>_<d>`	5.6.7
`Rte_Receive_<p>_<d>`	5.6.8
`Rte_Call_<p>_<o>`	5.6.9
`Rte_Result_<p>_<o>`	5.6.10
`Rte_Mode_<p>_<o>`	5.6.23

Table 5.3: Table of API functions that are referenced in the port API section.

[rte_sws_2620] An API function shall only be included in a port data structure, if it is required at least by one port.

[rte_sws_2621] If a function table entry is available in a port data structure, the corresponding function shall be implemented for all ports that use this port data structure type. API functions that are not required by the AUTOSAR configuration shall behave like those for an unconnected port.

APIs may be required only for some ports of a software component instance due to differences in for example the need for transmission acknowledgement. rte_sws_2621 is necessary for the concept of the indirect API. It allows iteration over ports.

[rte_sws_1055] The name of each function table entry in a port data structure shall be `<name>_<d/o>` where `<name>` is the API root (e.g. Call, Write) and `<d/o>` the data element or operation name.

Requirement rte_sws_1055 does *not* include the port name in the function table entry name since the port is implicit when using a port handle.

[rte_sws_3726] The data type of each function table entry in a port data structure shall be a function pointer that points to the generated RTE function.

The signature of a generated function, and hence the definition of the function pointer type, is the same as the signature of the relevant RTE API call (see Section 5.6) with the exception that the instance handle is omitted.

Example 5.16

This example shows a port data structure for the provide ports of the interface type `i2` in an AUTOSAR SW-C `c`.

`i2` is a SenderReceiverInterface which contains a data element prototype of type `uint8` with isQueued set to false.

If one of the provide ports of `c` for the interface `i2` has a transmission acknowledgement defined and `i2` is not used with `data element invalidation`, the AUTOSAR types header file would include a port data structure type like this:

```
struct Rte_PDS_c_i2_P {
  Std_ReturnType (*Feedback_a)(uint8);
  Std_ReturnType (*Write_a)(uint8);
}
```

If the provide port `p1` of the AUTOSAR SW-C `c` is of interface `i2`, the generated component header file would include the following macros to provide the direct API functions `Rte_Feedback_p1_a` and `Rte_Write_p1_a`:

```
/*direct API*/
#define Rte_Feedback_p1_a(inst,data) ((inst)->p1.Feedback_a)(data)
#define Rte_Write_p1_a(inst,data) ((inst)->p1.Write_a)(data)
```

[rte_sws_2618] The port data structures within a component data structure shall first be sorted on the port data structure type name and then on the short name of the port.

The requirements rte_sws_3731 and rte_sws_2618 guarantee, that all port data structures within the component data structure are grouped by their interface type and require/provide-direction.

Example 5.17

This example shows the grouping of port data structures within the component data structure.

The AUTOSAR types header file for an AUTOSAR SW-C c with three provide ports p1, p2, and p3 of interface i2 would include a block of port data structures like this in the generated AUTOSAR Types Header file:

```
struct Rte_CDS_c {
  ...
  struct Rte_PDS_c_i1_R z;

/* component data structures          *
 * for provide ports of interface i2  */
  struct Rte_PDS_c_i2_P p1;
  struct Rte_PDS_c_i2_P p2;
  struct Rte_PDS_c_i2_P p3;

/*further component data structures*/
  struct Rte_PDS_c_i2_R c;
  ...
}

```

If `inst` is a pointer to a component data structure, and ph is defined by

```
struct Rte_PDS_c_i2_P *ph = &(inst->p1);
```

ph points to the port data structure `p1` of the instance handle `inst`. Since the three provide port data structures `p1`, `p2`, and `p3` of interface `i2` are ordered squentially in the component data structure, `ph` can also be interpreted as an array of port data structures. E.g., `ph[2]` is equal to `inst->p3`.

In the following, ph will be called a port handle.

[rte_sws_1343] RTE shall create port handle types for each port data structure using `typedef` to a pointer to the appropriate port data structure.

[rte_sws_1342] The port handle type name shall be `Rte_PortHandle_<i>_<P/R>` where `<i>` is the port interface name and ‘P’ or ‘R’ are literals to indicate provide or receive ports respectively.

[rte_sws_1053] The port handle types shall be written to the application header file.

The port handle types cannot be included in the AUTOSAR types header file due to potential name clashes between components.

RTE provides port handles for access to the arrays of port data structures of the same interface type and provide/receive direction by the macro `Rte_Ports`, see section 5.6.1, and to the number of similar ports by the macro `Rte_NPorts`, see 5.6.1.

Example 5.18

For the provide port `i2` of AUTOSAR SW-C `c` from example 5.16, the following port handle type will be defined in the component header file:

```
typedef struct Rte_PDS_c_i2_P *Rte_PortHandle_i2_P;
```

The macros to access the port handles for the indirect API might look like this in the generated component header file:

```
/*indirect (port oriented) API*/
#define Rte_Ports_i2_P(inst) &((inst)->p1)
#define Rte_NPorts_i2_P(inst) 3
```

So, the port handle ph of the previous example 5.17 could be defined by a user as:

```
Rte_PortHandle_i2_P ph = Rte_Ports_i2_P(inst);
```

To write '49' on all ports `p1` to `p3`, the indirect API can be used within the software component as follows:

```
uint8 p;
Rte_PortHandle_i2_P ph = Rte_Ports_i2_P(inst);
for(p=0;p<Rte_NPorts_i_P(inst);p++) {
  ph[p].Write_a(49);
}
```

Software components may also want to set up their own port handle arrays to iterate over a smaller sub group than all ports with the same interface and direction. `Rte_Port` can be used to pick the port handle for one specific port, see 5.6.3.

5.4.2.7 Inter Runnable Variable API Section

The Inter Runnable Variable API section comprises zero or more *function table entries* within the component data structure type that defines all explicit API functions to access an inter runnable variable by the software-component (instance). The API for implicit access of inter runnable variables does not have any *function table entries*, since the implicit API uses macro's to access the inter runnable variables or their shadow memory directly, see section 5.4.2.3.

Since the entries of this section are only required to access the explicit InterRunnableVariable API if a software component supports multiple instantiation, it shall be omitted for software components which do not support multiple instantiation.

[rte_sws_3725] If the component supports multiple instantiation, the member name of each function table entry within the component data structure shall be `<name>_<re>_<d>`

where <name> is the API root (e.g. IrvRead), <re> the runnable name, and <d> the inter runnable variable name.

[rte_sws_3752] The data type of each function table entry shall be a function pointer that points to the generated RTE function.

The signature of a generated function, and hence the definition of the function pointer type, is the same as the signature of the relevant RTE API call (see Section 5.6) with the exception that the instance handle is omitted.

[rte_sws_2623] If the component supports multiple instantiation, the inter runnable variable API section shall contain pointers to the following API functions:

API function	reference
`Rte_IrvRead_<re>_<d>`	5.6.19
`Rte_IrvWrite_<re>_<d>`	5.6.20

Table 5.4: Table of API functions that are referenced in the inter runnable variable API section

[rte_sws_3791] If the software component does not support multiple instantiation, the inter runnable variable API section shall be empty.

5.4.2.8 Vendor Specific Section

The vendor specific section is used to contain any vendor specific data required to be supported for each instances. By definition the contents of this section are outside the scope of this chapter and only available for use by the RTE generator responsible for the “RTE Generation” phase.

5.5 API Data Types

Besides the API functions for accessing RTE services, the API also contains RTE-specific data types.

5.5.1 Std_ReturnType

The specification in [13] specifies a standard API return type `Std_ReturnType`. The `Std_ReturnType` defines the "‘status"’ and "‘error values"’ returned by API functions. It is defined as a `uint8` type. The value “0” is reserved for “No error occurred”.

Figure 5.4 shows the general layout of `Std_ReturnType`.

The two most significant bits of the `Std_ReturnType` are reserved flags:

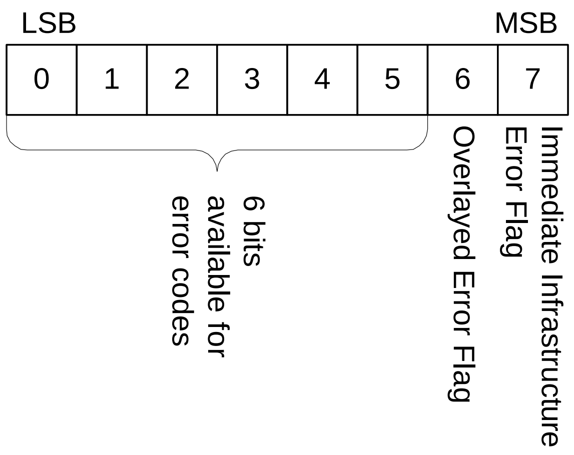

LSB
MSB
0
1
2
3
4
5
6
7
6 bits available for error codes
Overlayed Error Flag
Immediate Infrastructure Error Flag

Figure 5.4: Bit-Layout of the Std_ReturnType

- The most significant bit 7 of `Std_ReturnType` is the “Immediate Infrastructure Error Flag” with the following values
  - “1” the error code indicates an immediate infrastructure error.
  - “0” the error code indicates no immediate infrastructure error.
- The second most significant bit 6 of `Std_ReturnType` is the Overlayed Error Flag. The use of this flag depends on the context and will be explained in table 5.6.

5.5.1.1 Infrastructure Errors

Infrastructure errors are split into two groups:

- "Immediate Infrastructure Errors" can be associated with the currently available data set. These `Immediate Infrastructure Errors` are mutually exclusive. Only one of these errors can be notified to a SW-C with one API call.

  [rte_sws_2593] `Immediate Infrastructure Errors` shall override any application level error.

  `Immediate Infrastructure Error` codes are used on the receiver side for errors that result in no reception of application data and application errors.

  An `Immediate Infrastructure Error` is indicated in the `Std_ReturnType` by the `Immediate Infrastructure Error Flag` being set. rte_sws_2591

- "Overlayed Errors" are associated with communication events that happened after the reception of the currently available data set, e.g., `data element outdated` notification, or loss of data elements due to queue overflow.

  [rte_sws_1318] `Overlayed Error Flags` shall be reported using the unique bit of the `Overlayed Error Flag` within the `Std_ReturnType` type.

  An `Overlayed Error` can be combined with any other application or infrastructure error code.

5.5.1.2 Application Errors

[rte_sws_2573] An application error shall be coded in the least significant 6 bits of `Std_ReturnType` with the `Immediate Infrastructure Error Flag` set to "0". The application error code may not use the `Overlayed Error Flag`.

This results in the following value range for application errors:

range	minimum value	maximum value
application errors	1	63

Table 5.5: application error value range

In client server communication, the server may return any value within the application error range. The client will then receive one of the following:

- An `Immediate Infrastructure Error` to indicate that the communication was not successful or
- The server return code or
- The server return code might be overlayed by the `Overlayed Error Flag` in a future release of RTE. In this release, there is no overlayed error defined for client server communication.

The client can filter the return value, e.g., by using the following code:

```
Std_ReturnType status;
status = Rte_Call_<p>_<d>(<instance>, <parameters>);
if (status & 64) {
    /* handle overlayed error flag                       *
     * in this release of the RTE, the flag is reserved *
     * but not used for client server communication     */
}
status &= (Std_ReturnType)(~64);
if(status & 128) {
    /* handle infrastructure error                       */
}
else {
    /* handle application error with error code status  */
}
```

5.5.1.3 Predefined Error Codes

[rte_sws_in_2622] For client server communication, application error values are defined per client server interface and shall be passed to the RTE with the interface configuration.

The following standard error and status identifiers are defined:

Symbolic name	Value	Comments
[rte_sws_1058] `RTE_E_OK`	0	No error occurred.

Standard Application Error Values:		
[rte_sws_2594] `RTE_E_INVALID`	1	Generic application error indicated by signal invalidation in sender receiver communication with isQueued = false on the receiver side.
To be defined by the corresponding AUTOSAR Service	1	Returned by AUTOSAR Services to indicate a generic application error.

`Immediate Infrastructure Error` codes

Symbolic name	Value	Comments
[rte_sws_1060] `RTE_E_COMMS_ERROR`	128	A communications error occurred. No value is available. This error has the following semantics: • The OUT buffers of a client or of explicit read APIs are not modified • no runnable with startOnEvent on a DataReceivedEvent for this dataElement-Prototype is triggered. • the buffers for implicit read access will keep the previous value.
[rte_sws_1064] `RTE_E_TIMEOUT`	129	A blocking API call returned due to expiry of a local timeout rather than the intended result. OUT buffers are not modified. The interpretation of this being an error depends on the application.
[rte_sws_1317] `RTE_E_LIMIT`	130	A internal RTE limit has been exceeded. Request could not be handled. OUT buffers are not modified.
[rte_sws_1061] `RTE_E_NO_DATA`	131	An explicit read API call returned no data. (This is no error.)
[rte_sws_1065] `RTE_E_TRANSMIT_ACK`	132	Transmission acknowledgement received.

`Overlayed Errors` These errors do not refer to the data returned with the API. They can be overlayed with other Application- or Immediate Infrastructure Errors.		
[rte_sws_2571] `RTE_E_LOST_DATA`	64	An API call for reading received data of isQueued = true indicates that some incoming data has been lost due to an overflow of the receive queue or due to an error of the underlying communication stack.
[rte_sws_2525] `RTE_E_MAX_AGE_EXCEEDED`	64	An API call for reading received data of isQueued = false indicates that the available data has exceeded the aliveTimeout limit. A COM signal outdated callback will result in this error.

Table 5.6: RTE Error and Status values

The underlying type for `Std_ReturnType` is defined as a `uint8` for reasons of compatibility – it avoids RTEs from different vendors assuming a different size if an `enum` was the underlying type. Consequently, `#define` is used to declare the error values:

```
typedef uint8 Std_ReturnType;

#define RTE_E_OK ((Std_ReturnType) 0)
```

[rte_sws_1269] The standard errors as defined in table 5.6 including `RTE_E_OK` shall be defined in the RTE Header File.

[rte_sws_2575] Application Error Identifiers with exception of `RTE_E_INVALID` shall be defined in the Application Header File.

[rte_sws_2576] The application errors shall have a symbolic name defined as follows:

```
#define RTE_E_<interface>_<error> <error value>
```

where `<interface>` rte_sws_in_1352 and `<error>` rte_sws_in_2574 are the interface and error names from the configuration.

An `Std_ReturnType` value can be directly compared (for equality) with the above pre-defined error identifiers.

5.5.2 Rte_Instance

The `Rte_Instance` data type defines the handle used to access instance specific information from the component data structure.

[rte_sws_1148] The underlying data type for an instance handle shall be a pointer to a *Component Data Structure*.

The component data structure (see Section 5.4.2) is uniquely defined for a component type and therefore the data type for the instance handle is automatically unique for each component type.

The instance handle type is defined in the application header file rte_sws_1007.

To avoid long and complex type names within SW-C code the following requirement imposes a fixed name on the instance handle data type.

[rte_sws_1150] The name of the instance handle type shall be defined, using `typedef` as `Rte_Instance`.

5.5.3 RTE Modes

An `Rte_ModeType` is used to hold the identifiers for the ModeDeclarations of a ModeDeclarationGroup.

[rte_sws_2627] For each ModeDeclarationGroup, the AUTOSAR Types HeaderFile shall contain a type definition

```
typedef <type> Rte_ModeType_<ModeDeclarationGroup> <type>;
```

where `<ModeDeclarationGroup>` is the short name of the ModeDeclarationGroup and `<type>` is `uint8` for ModeDeclarationGroups with 258 or less ModeDeclarations and `uint16` for ModeDeclarationGroups with more than 258 ModeDeclarations.

[rte_sws_2568] For each mode of a mode declaration group, the AUTOSAR Types Header file shall contain a definition

```
#define RTE_MODE_<ModeDeclaration> <index>
```

where `<ModeDeclaration>` is the short name of a ModeDeclaration and `<index>` is the index of the ModeDeclarations in alphabetic ordering of the short names within the ModeDeclarationGroup. The lowest index shall be '0'.

5.6 API Reference

The functions described in this section are organized by the RTE API mapping name used by C and C++ AUTOSAR software-components to access the API. The API mapping hides from the AUTOSAR software-component programmer any need to be aware of the steps taken by the RTE generator to ensure that the generated API functions have unique names.

The instance handle as the first parameter of the API calls is marked as an optional parameter in this section. If an AUTOSAR software-component supports multiple instantiation, the instance handle shall be passed rte_sws_1013.

Note that rte_sws_3806 requires that the instance handle parameter does not exist if the AUTOSAR software-component does not support multiple instantiation.

5.6.1 Rte_Ports

Purpose:	Provide an array of the ports of a given interface type and a given provide / require usage.

Signature:	**[rte_sws_2619]**
	`Rte_PortHandle_<i>_<R/P>`
	`Rte_Ports_<i>_<R/P>([IN Rte_Instance])`
	
	Where here `<i>` is the port interface name and 'P' or 'R' are literals to indicate provide or require ports respectively.

Existence:	**[rte_sws_2613]** An `Rte_Ports` API shall be created for each interface type and usage by a port of the AUTOSAR SW-C.
Description:	The `Rte_Ports` API provides access to an array of ports for the port oriented API.
Return Value:	Array of port data structures of the corresponding interface type and usage.

Notes: None.

5.6.2 Rte_NPorts

Purpose: Provide the number of ports of a given interface type and provide / require usage.

Signature: **[rte_sws_2614]**

```
uint8
Rte_NPorts_<i>_<R/P>([IN Rte_Instance])
```

Where here <i> is the port interface name and 'P' or 'R' are literals to indicate provide or require ports respectively.

Existence: **[rte_sws_2615]** An Rte_NPorts API shall be created for each interface type and usage by a port of the AUTOSAR SW-C.

Description: The Rte_NPorts API supports access to an array of ports for the port oriented API.

Return Value: Number of port data structures of the corresponding interface type and usage.

Notes: None.

5.6.3 Rte_Port

Purpose: Provide access to the port data structure for a single port of a particular software component instance. This allows a software component to extract a sub-group of ports characterized by the same interface in order to iterate over this sub-group.

Signature: **[rte_sws_1354]**

```
Rte_PortHandle_<i>_<R/P>
Rte_Port_<p>([IN Rte_Instance])
```

where <i> is the port interface name and <p> is the name of the port.

Existence: **[rte_sws_1355]** An Rte_Port API shall be created for each port of the AUTOSAR SW-C.

Description: The Rte_Port API provides a pointer to a single port data structure, in order to support the indirect API.

Return Value: Pointer to port data structure for the appropriate port.

Notes: None.

5.6.4 Rte_Send/Rte_Write/Rte_Switch

Purpose: Initiate an "explicit" sender-receiver transmission. The `Rte_Write` API call is used for "data" (isQueued = false), the `Rte_Send` API call used for "events" (isQueued = true), and the `Rte_Switch` API call is used for mode switches.

Signature: **[rte_sws_1071]**

```
Std_ReturnType
Rte_Write_<p>_<o>([IN Rte_Instance <instance>],
                  IN <data>)
```

[rte_sws_1072]

```
Std_ReturnType
Rte_Send_<p>_<o>([IN Rte_Instance <instance>],
                 IN <data>)
```

[rte_sws_2631]

```
Std_ReturnType
Rte_Switch_<p>_<o>([IN Rte_Instance <instance>],
                   IN Rte_ModeType_<M> <mode>)
```

Where <p> is the port name and <o> the DataElementPrototype or ModeDeclarationGroupPrototype within the sender-receiver interface categorizing the port.

Existence: **[rte_sws_1280]** The presence of a DataSendPoint for a provided DataElementPrototype with isQueued = false shall result in the generation of an `Rte_Write` API for the provided DataElementPrototype.

[rte_sws_1281] The presence of a DataSendPoint for a provided DataElementPrototype with isQueued = true shall result in the generation of an `Rte_Send` API for the provided DataElementPrototype.

[rte_sws_2632] An `Rte_Switch` API shall be generated for each provided ModeDeclarationGroupPrototype.

A DataSendPoint should be configured for each provided ModeDeclarationGroupPrototype. [3]

Description: The `Rte_Send` `Rte_Write` and `Rte_Switch` API calls initiate a sender-receiver communication where the transmission occurs at the point the API call is made (cf. explicit transmission).

The `Rte_Send` and `Rte_Write` API calls include exactly one IN parameter for the data element – this will be passed by value for primitive data types and by reference for all other types.

If the IN parameter is passed by reference, the pointer must remain valid until the API call returns.

[3]The release candidate 4 of the Software-Component Template Specification [18], does not yet allow a reference of a ModeDeclarationGroupPrototype by a SendPoint.

The Rte_Switch API calls includes exactly one IN parameter for the next mode <mode> of type Rte_ModeType_<M> where <M> is the ModeDeclarationGroup short name.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the Rte_Write Rte_Send or Rte_Switch call.

- **[rte_sws_1073]** RTE_E_OK – data passed to communication service successfully.
- **[rte_sws_1074]** RTE_E_COMMS_ERROR – a communications error was detected by the RTE (inter ECU communication only).
- **[rte_sws_2634]** RTE_E_LIMIT – an 'event' or a mode switch has been discarded due to a full queue. (intra ECU communication only).

Notes: The Rte_Write Rte_Send and Rte_Switch calls are closely related – Rte_Write is used to transmit "data" (isQueued = false), Rte_Send to transmit "events" (isQueued = true) and Rte_Switch to transmit mode switch indications.

Rte_Switch is restricted to ECU local communication. It behaves like Rte_Send in the case of local communication.

[rte_sws_1077] In case of inter ECU communication, the Rte_Write and Rte_Send shall cause an immediate transmission request.

Note that depending on the configuration a transmission request may not result in an actual transmission, for example transmission may be rate limited (time-based filtering) and thus dependent on other factors than API calls.

[rte_sws_1081] In case of inter ECU communication, the Rte_Write or Rte_Send API shall return 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 at the point the API call returns.

[rte_sws_2633] In case of intra ECU communication, the Rte_Switch API call and the Rte_Send API call shall return after an attempt to enqueue the mode switch.

[rte_sws_2635] In case of intra ECU communication, the Rte_Write API call shall return after copying the data.

Note that the mode switch might be discarded when the queue is full, see rte_sws_2634. This is different from the communication of "events" where the loss is silently discarded.

[rte_sws_1080] If the transmission acknowledgement is enabled, the RTE shall notify component when the transmission is acknowledged (mode switch completed) or a transmission error occurs.

[rte_sws_1082] If a provide port typed by a sender-receiver interface has multiple require ports connected (i.e. it has multiple receivers), then the RTE shall ensure that writes to all receivers are independent.

Requirement rte_sws_1082 ensures that an error detected by the RTE when writing to one receiver, e.g. an overflow in one component's queue, does not prevent the transmission of this message to other components.

5.6.5 Rte_Invalidate

Purpose: Invalidate a data element for an "explicit" sender-receiver transmission.

Signature: **[rte_sws_1206]**

```
Std_ReturnType
Rte_Invalidate_<p>_<o>([IN Rte_Instance <instance>])
```

Where <p> is the port name and <o> the data element within the sender-receiver interface categorizing the port.

Existence: **[rte_sws_1282]** An `Rte_Invalidate` API shall be created for any DataSendPoint that references a provided DataElementPrototype with isQueued = false that is marked as invalidatable and canInvalidate is enabled.

Description: The `Rte_Invalidate` API takes no parameters other than the instance handle – the return value is used to indicate the success, or otherwise, of the API call to the caller.

[rte_sws_1231] When COM is used for communication the API shall invoke the COM API functions `Com_InvalidateSignal`.

The behaviour required when COM is not used for communication is described in Section 4.3.1.8.

Return Value: The return value is used to indicate the "OK" status or errors detected by the RTE during execution of the `Rte_Invalidate` call.

- **[rte_sws_1207]** `RTE_E_OK` – No error occurred.
- **[rte_sws_1339]** `RTE_E_COMMS_ERROR` – a communications error was detected by the RTE.

Notes: The API name includes an identifier <p>_<o> that is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

The communication service configuration determines whether the signal receiver(s) receive an “invalid signal” notification or whether the invalidated signal is silently replaced by the signal’s initial value.

5.6.6 Rte_Feedback

Purpose: Provide access to acknowledgement notifications for explicit sender-receiver communication and to pass error notification to senders.

Signature: **[rte_sws_1083]**

```
Std_ReturnType
Rte_Feedback_<p>_<o>([IN Rte_Instance <instance>])
```

Where <p> is the port name and <o> the DataElementPrototype or ModeDeclarationGroupPrototype within the sender-receiver interface categorizing the port.

Existence: **[rte_sws_1283]** Acknowledgement is enabled for a provided DataElementPrototype or ModeDeclarationGroupPrototype by the presence of an AcknowledgementRequest.

[rte_sws_1284] A blocking `Rte_Feedback` API shall be generated for a provided DataElementPrototype if acknowledgement is enabled and a WaitPoint references a DataSendCompletedEvent that in turn references the DataElementPrototype or ModeDeclarationGroupPrototype.

[rte_sws_1285] A non-blocking `Rte_Feedback` API shall be generated for a provided DataElementPrototype if acknowledgement is enabled and a DataSendPoint references the DataElementPrototype but no DataSendCompletedEvent references the DataElementPrototype or ModeDeclarationGroupPrototype.

[rte_sws_1286] If acknowledgement is enabled for a provided DataElementPrototype/ModeDeclarationGroupPrototype and a DataSendCompletedEvent references a runnable entity as well as the DataElementPrototype/ModeDeclarationGroupPrototype, the runnable entity shall be activated when the transmission acknowledgement occurs or when a timeout was detected by the RTE. rte_sws_1137.

Requirement rte_sws_1286 merely affects when the runnable is activated – an API call should still be created, according to requirement rte_sws_1285 to actually read the data.

[rte_sws_1287] A DataSendCompletedEvent that references a runnable entity and is referenced by a WaitPoint shall be an invalid configuration.

Description: The `Rte_Feedback` API takes no parameters other than the instance handle – the return value is used to indicate the acknowledgement status to the caller.

The `Rte_Feedback` API applies only to explicit sender-receiver communication.

Return Value: The return value is used to indicate the "status" status and errors detected by the RTE during execution of the `Rte_Feedback` call.

- **[rte_sws_1084]** `RTE_E_NO_DATA` – (non-blocking read) no data returned and no other error occurred when the feedback read was attempted.
- **[rte_sws_1085]** `RTE_E_TIMEOUT` – (intra-ECU only) no data returned within the specified timeout and no other error occurred when the feedback read was attempted.
- **[rte_sws_3774]** `RTE_E_COMMS_ERROR` – (inter-ECU only) no data was returned within the specified timeout or another COM error occurred within the API call.
- **[rte_sws_1086]** `RTE_E_TRANSMIT_ACK` – A "transmission acknowledgment" has been received from the communication service.

  For intra ECU communication of mode switches, this indicates, that the old mode is left, the mode disabling dependencies are switched to the next mode, and the runnables for entering the new mode are triggered (see rte_sws_2587).

The RTE_E_TRANSMIT_ACK return value is not considered to be an error but rather indicates correct operation of the API call.

When `RTE_E_NO_DATA` occurs, a component is free to reinvoke `Rte_Feedback` and thus repeat the attempt to read the feedback status.

Notes: The API name includes an identifier <p>_<o> that indicates the read access point name and is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

If multiple transmissions on the same port/element are outstanding it is not possible to determine which is acknowledged first. If this is important, transmissions should be serialized with the next occuring only when the previous transmission has been acknowledged or has timed out.

5.6.7 Rte_Read

Purpose: Performs an "explicit" read on a sender-receiver communication data element with "data" semantics (isQueued = false).

Signature: **[rte_sws_1091]**

```
Std_ReturnType
Rte_Read_<p>_<o>([IN Rte_Instance <instance>],
                 OUT <data>)
```

Where <p> is the port name and <o> the data element within the sender-receiver interface categorizing the port.

Existence: **[rte_sws_1289]** A non-blocking `Rte_Read` API shall be generated if a DataReceivePoint references a required DataElementPrototype with ‘data’ semantics (isQueued = false).

[rte_sws_1291] A WaitPoint that references a DataReceivedEvent that in turn references a required DataElementPrototype with ‘data’ semantics (isQueued = false) shall be considered an invalid configuration.

[rte_sws_1292] When a DataReceivedEvent references a RunnableEntity and a required DataElementPrototype and no WaitPoint references the DataReceivedEvent, the runnable entity shall be activated when the data is received. rte_sws_1135.

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

[rte_sws_1313] A DataReceivedEvent that references a runnable entity and is referenced by a WaitPoint shall be an invalid configuration.

Description: The `Rte_Read` API call includes exactly one OUT parameter to pass back the received data. The pointer to the OUT parameter must remain valid until the API call returns.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the `Rte_Read` or `Rte_Receive` API call or errors detected by the communication system.

- **[rte_sws_1093]** `RTE_E_OK` – data read successfully.
- **[rte_sws_2626]** `RTE_E_INVALID` – `data element invalid`.
- **[rte_sws_2627]** `RTE_E_MAX_AGE_EXCEEDED` – `data element outdated`. This `Overlayed Error` can be combined with any of the above error codes.

Notes: The API name includes an identifier <p>_<o> that indicates the read access point name and is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

5.6.8 Rte_Receive

Purpose: Performs an “explicit” read on a sender-receiver communication data element with “event” semantics (isQueued = true).

[rte_sws_1092]
```
Std_ReturnType
Rte_Receive_<p>_<o>([IN Rte_Instance <instance>],
                    OUT <data>)
```

Where <p> is the port name and <o> the data element within the sender-receiver interface categorizing the port.

Existence: **[rte_sws_1288]** A non-blocking `Rte_Receive` API shall be generated if a DataReceivePoint references a required DataElementPrototype with ‘event’ semantics (isQueued = true).

[rte_sws_1290] A blocking `Rte_Receive` API shall be generated if a DataReceivePoint references a required DataElementPrototype with ‘event’ semantics (isQueued = true) that is, in turn, referenced by a DataReceivedEvent and the DataReceivedEvent is referenced by a WaitPoint.

When a DataReceivedEvent references a RunnableEntity and a required DataElementPrototype and no WaitPoint references the DataReceivedEvent, the runnable entity shall be activated when the event is received. rte_sws_1292 rte_sws_1135.

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

A DataReceivedEvent that references a runnable entity and is referenced by a WaitPoint shall be an invalid configuration. rte_sws_1313

Description: The `Rte_Receive` API call includes exactly one OUT parameter to pass back the received data.

The pointer to the OUT parameter must remain valid until the API call returns.

Return Value: The return value is used to indicate errors detected by the RTE during execution of the `Rte_Receive` API call or errors detected by the communication system.

- **[rte_sws_2598]** `RTE_E_OK` – data read successfully.
- **[rte_sws_1094]** `RTE_E_NO_DATA` – (explicit non-blocking read) no data returned and no other error occurred when the read was attempted.

- **[rte_sws_1095]** `RTE_E_TIMEOUT` – (explicit blocking read) no data returned and no other error occurred when the read was attempted.
- **[rte_sws_2572]** `RTE_E_LOST_DATA` – Indicates that some incoming data has been lost due to an overflow of the recieve queue or due to an error of the underlying communication layers. This is not an error of the data returned in the parameters. This `Overlayed Error` can be combined with any of the above.

The RTE_E_NO_DATA and RTE_E_TIMEOUT return value are not considered to be errors but rather indicate correct operation of the API call.

Notes: The API name includes an identifier <p>_<o> that indicates the read access point name and is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

5.6.9 Rte_Call

Purpose: Initiate a client-server communication.

Signature: **[rte_sws_1102]**

```
Std_ReturnType
Rte_Call_<p>_<o>([IN Rte_Instance <instance>],
                 [IN|IN/OUT|OUT] <data_1>...
                 [IN|IN/OUT|OUT] <data_n>)
```

Where <p> is the port name and <o> the operation within the client-server interface categorizing the port.

Existence: **[rte_sws_1293]** A synchronous `Rte_Call` API shall be generated if a SynchronousServerCallPoint references a required OperationPrototype.

[rte_sws_1294] An asynchronous `Rte_Call` API shall be generated if an AsynchronousServerCallPoint references a required OperationPrototype.

[rte_sws_1295] A configuration that includes both synchronous and asynchronous ServerCallPoints for a given OperationPrototype shall be invalid.

Description: Client function to initiate client-server communication. The `Rte_Call` API is used for both synchronous and asynchronous calls.

The `Rte_Call` API includes zero or more IN, IN/OUT and OUT parameters. IN parameters are passed by value for primitive data types and by reference for all other types, OUT parameters are always by reference and 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.

The pointers to all parameters passed by reference must remain valid until the API call returns.

Return Value: **[rte_sws_1103]** The return value shall be used to indicate infrastructure errors detected by the RTE during execution of the `Rte_Call` call and, for synchronous communication, infrastructure and application errors during execution of the server.

- **[rte_sws_1104]** `RTE_E_OK` – The API call completed successfully.
- **[rte_sws_1105]** `RTE_E_LIMIT` – The same client component has multiple outstanding asynchronous client-server communications to the same server.
- **[rte_sws_1106]** `RTE_E_COMMS_ERROR` – A communications error occurred - indicates that the request has *not* been successfully passed to the communication service. The buffers of the return parameters shall not be modified.
- **[rte_sws_1107]** `RTE_E_TIMEOUT` – (synchronous inter-task and inter-ECU only) No reply was received within the configured timeout. The buffers of the return parameters shall not be modified.
- **[rte_sws_2577]** The application error (synchronous client-server) from a server shall only be returned if none of the above infrastructure errors (other than `RTE_E_OK`) have occured.

Note that the RTE_E_OK return value indicates that the `Rte_Call` API call completed successfully. In case of a synchronous client server call it also indicates successful processing of the request by the server.

An asynchronous server invocation is considered to be outstanding until either the client retrieved the result successfully, a timeout was detected by the RTE in inter-ECU communication or the server runnable has terminated after a timeout was detected in intra-ECU communication.

[rte_sws_1209] When the `RTE_E_TIMEOUT` error occurs any subsequent reply from the server received before another call is made shall be discarded by the RTE.

Notes: **[rte_sws_1109]** The interface operation's OUT parameters shall be omitted for an *asynchronous* call.

For asynchronous communication the `Rte_Call` should include only IN and IN/OUT parameters – the OUT parameters are required when the client collects the result (e.g. using `Rte_Result`).

5.6.10 Rte_Result

Signature: **[rte_sws_1111]**

```
Std_ReturnType
Rte_Result_<p>_<o>([IN Rte_Instance <instance>],
                   [OUT <param 1>]...
                   [OUT <param n>])
```

Where <p> is the port name and <o> the operation within the client-server interface categorizing the port.

The signature can include zero or more OUT parameters depending on the signature of the operation in the client-server interface.

Existence: **[rte_sws_1296]** A non-blocking `Rte_Result` API shall be generated if an AsynchronousServerCallReturnsEvent references a required OperationPrototype and no WaitPoint references the AsynchronousServerCallReturnsEvent.

[rte_sws_1297] A blocking `Rte_Result` API shall be generated if an AsynchronousServerCallReturnsEvent references a required OperationPrototype and a WaitPoint references the AsynchronousServerCallReturnsEvent.

[rte_sws_1298] If an AsynchronousServerCallReturnsEvent references a RunnableEntity and a required OperationPrototype the runnable entity shall be activated when the operation's result is available or when a timeout was detected by the RTE rte_sws_1133.

Requirement rte_sws_1298 merely affects when the runnable is activated – an API call should still be created to actually read the reply based on requirement rte_sws_1296.

[rte_sws_1312] An AsynchronousServerCallReturnsEvent that references a runnable entity and is referenced by a WaitPoint is invalid.

Description: The `Rte_Result` API is used by a client to collect the result of an *asynchronous* client-server communication.

The `Rte_Result` API includes zero or more OUT parameters to pass back results.

The pointers to all parameters passed by reference must remain valid until the API call returns.

Return Value: The return value is used to indicate errors from either the `Rte_Result` call itself or communication errors detected before the API call was made.

- **[rte_sws_1112]** `RTE_E_OK` – The API call completed successfully.

- **[rte_sws_1113]** `RTE_E_NO_DATA` – (non-blocking read) The server's result is not available but no other error occurred within the API call. The buffers for the OUT parameters shall not be modified.
- **[rte_sws_1114]** `RTE_E_TIMEOUT` – The server's result is not available within the specified timeout but no other error occurred within the API call. The buffers for the OUT parameters shall not be modified.
- **[rte_sws_2578]** Application Errors – The error code of the server shall only be returned, if none of the above infrastructure errors or indications have occured.

The RTE_E_NO_DATA and RTE_E_TIMEOUT return value are not considered to be errors but rather indicate correct operation of the API call.

When RTE_E_NO_DATA occurs, a component is free to invoke `Rte_Result` again and thus repeat the attempt to read the server's result.

Notes: The API name includes an identifier <p>_<o> that indicates the read access point name and is formed from the port and operation item names. See Section 5.2.6.4 for details on the naming convention.

5.6.11 Rte_Pim

Purpose: Provide access to the defined per-instance memory (section) of a software component.

Signature: **[rte_sws_1118]**

```
<type>
Rte_Pim_<name>([IN Rte_Instance <instance>])
```

Where `<name>` is the (short) name of the per-instance name.

Existence: **[rte_sws_1299]** An `Rte_PIM` API shall be created for each defined PerInstanceMemory within the AUTOSAR software-component (description).

Description: The `Rte_PIM` API provides access to the per-instance memory (section) defined in the context of a InternalBehavior of a software-component description.

Return Value: **[rte_sws_1119]** The API returns a typed reference (in C a typed pointer) to the per-instance memory.

Notes: The software-component shall define the return type `<type>` in the attribute `<typeDefinition>` of PerInstanceMemory, if it is a complex AUTOSAR data type. It is assumed that this attribute contains a String that represents a C type definition (typedef) in valid C syntax (see rte_sws_2304).

5.6.12 Rte_CData

Purpose: Provide access to the defined the configuration data (a.k.a. *Characteristics*) of a software component. The configuration data is used to define per-component instance read-only data.

Signature: **[rte_sws_1252]**

```
<return>
Rte_CData_<name>([IN Rte_Instance <instance>])
```

Where `<name>` is the configuration data name.

Existence: **[rte_sws_1300]** An `Rte_CData` API shall be created for each defined Characteristic within AUTOSAR software-component.

Description: The `Rte_CData` API provides access to the defined configuration data within a component description. The actual data values for a software-component instance may be set after component compilation.

Return Value: The API returns a reference (in C, a pointer) to the configuration data.

[rte_sws_1254] A pointer to the configuration data.

The return type of `Rte_CData` is specified by the Characteristic element within the software component description. Thus the component does not need to use type casting to access the configuration data.

[rte_sws_1255] For a given configuration data and software-component instance, the value return from `Rte_CData` shall be the same each time the call is invoked.

Requirement rte_sws_1255 ensures that a component may cache the return from `Rte_CData` in a local variable since it is guaranteed that the value shall not change across executions of a runnable entity.

Notes: None.

5.6.13 Rte_IRead

Purpose: Provide **read** access to the data elements defined with `DataReadAccess` semantics.

Signature: **[rte_sws_3741]**

```
<return>
Rte_IRead_<re>_<p>_<d>([IN Rte_Instance])
```

Where `<re>` is the runnable entity name, <p> the port name and <d> the data element name.

Existence: **[rte_sws_1301]** An `Rte_IRead` API shall be created for a required DataElementPrototype if the RunnableEntity has DataReadAccess that refers to the DataElementPrototype.

Description: The `Rte_IRead` API provides access to the data elements declared as accessed by a runnable using DataReadAccess. The API function is guaranteed to be have constant execution time – it will typically be implemented by a macro – and therefore can be used within category 1A runnable entities.

No error information is provided by this API. If required, the error status can be picked up with a separate API, see 5.6.16

The data value can always be read. To provide the required consistency access by a runnable is to a *copy* of the data that is guaranteed never to be modified by the RTE during the lifetime of the runnable entity.

Implicit data read access by a SW-C should always return defined data.

[rte_sws_1268] The RTE shall ensure that implicit read accesses will not deliver undefined data item values.

In case where there may be an implicit read access before the first data reception an initial value has to be provided as the result of this implicit read access.

Return Value: **[rte_sws_3743]** The return type if the `Rte_IRead` is dependent on the data element type.

The return type of `Rte_IRead` is determined by the data element type. Thus the component does not need to use type casting to convert access the data.

Notes: None.

5.6.14 Rte_IWrite

Purpose: Provide **write** access to the data elements defined with `DataWriteAccess` semantics.

Signature: **[rte_sws_3744]**

```
void
Rte_IWrite_<re>_<p>_<d>([IN RTE_Instance],
                        IN <type>)
```

Where `<re>` is the runnable entity name, <p> the port name and <d> the data element name.

Existence: **[rte_sws_1302]** An `Rte_IWrite` API shall be created for a provided DataElementPrototype if the RunnableEntity has DataWriteAccess that refers to the DataElementPrototype.

Description: The `Rte_IWrite` API provides write access to the data elements declared as accessed by a runnable using DataWriteAccess. The API function is guaranteed to be have constant execution time – it will typically be implemented by a macro – and therefore can be used within category 1A runnable entities.

No access error information is required for the user – the value can always be written. To provide the required write-back semantics the RTE only makes written values available to other entities after the writing runnable entity has terminated.

[rte_sws_3746] The `Rte_IWrite` API call include exactly one IN parameter for the data element – this is passed by value for primitive data types and by reference for all other types.

Return Value: **[rte_sws_3747]** `Rte_IWrite` has no return value.

For C/C++ rte_sws_3747 means using a return type of `void`.

Notes: None.

5.6.15 Rte_IInvalidate

Purpose: Invalidate a data element defined with `DataWriteAccess` semantics.

Signature: **[rte_sws_3800]**

```
void
Rte_IInvalidate_<re>_<p>_<d>([IN Rte_Instance <instance>])
```

Where `<re>` is the runnable entity name, <p> the port name and <d> the data element name.

Existence: **[rte_sws_3801]** An `Rte_IInvalidate` API shall be created for a provided DataElementPrototype if the RunnableEntity has DataWriteAccess that refers to the DataElementPrototype and canInvalidate is enabled.

Description: The `Rte_IInvalidate` API takes no parameters other than the instance handle – the return value is used to indicate the success, or otherwise, of the API call to the caller.

[rte_sws_3802] The `Rte_IInvalidate` shall be implemented as a macro that writes the invalid value rte_sws_in_5031 to the buffer.

[rte_sws_3778] If `Rte_IInvalidate` is followed by an `Rte_IWrite` for the same data element prototype call or vice versa, the RTE shall use the last value written before the runnable entity terminates (last-is-best semantics).

rte_sws_3778 states that an `Rte_IWrite` overrules an `Rte_IInvalidate` call if it occurs after the `Rte_IInvalidate`, since `Rte_IWrite` overwrites the contents of the internal buffer for the data element prototype before it is made known to other runnable entities.

Return Value: **[rte_sws_3803]** `Rte_IInvalidate` has no return value.

For C/C++ rte_sws_3803 means using a return type of `void`.

Notes: The communication service configuration determines whether the signal receiver(s) receive an "invalid signal" notification or whether the invalidated signal is silently replaced by the signal's initial value.

5.6.16 Rte_IStatus

Purpose: Provide the error status of a `data element` defined with `DataReadAccess` semantics.

Signature: **[rte_sws_2599]**

```
Std_ReturnType
Rte_IStatus_<re>_<p>_<d>([IN Rte_Instance])
```

Where `<re>` is the runnable entity name, <p> the port name and <d> the data element name.

Existence: **[rte_sws_2600]** An `Rte_IStatus` API shall be created for a required DataElementPrototype if a RunnableEntity has DataReadAccess refering to the DataElementPrototype and if either

- `data element outdated` notification or
- `data element invalidation`

is activated for this `data element`.

[rte_sws_ext_2601] The `Rte_IStatus` API shall only be used by a RunnableEntity that either has a DataReadAccess refering to the DataElementPrototype or is triggered by a DataReceiveErrorEvent refering to the DataElementPrototype.

Description: The `Rte_IStatus` API provides access to the current status of the data elements declared as accessed by a runnable using DataReadAccess. The API function is guaranteed to be have constant execution time – it will typically be implemented by a macro – and therefore can be used within category 1A runnable entities.

To provide the required consistency access by a runnable is to a *copy* of the status together with the data that is guaranteed never to be modified by the RTE during the lifetime of the runnable entity.

Return Value: The return value is used to indicate errors detected by the communication system.

- **[rte_sws_2602]** `RTE_E_OK` – no errors.
- **[rte_sws_2603]** `RTE_E_INVALID` – `data element invalid`.
- **[rte_sws_2604]** `RTE_E_MAX_AGE_EXCEEDED` – `data element outdated`. This `Overlayed Error` can be combined with any of the above error codes.
- **[rte_sws_1356]** `RTE_E_COMMS_ERROR` – a communications error was detected by the RTE.

Notes: None.

5.6.17 Rte_IrvIRead

Purpose: Provide **read** access to the *InterRunnableVariables with implicit* behavior of an AUTOSAR SW-C.

Signature: **[rte_sws_3550]**

```
<return>
Rte_IrvIRead_<re>_<name>([IN RTE_Instance <instance>])
```

Where `<re>` is the name of the runnable entity the API might be used in, `<name>` is the name of the InterRunableVariables.

Existence: **[rte_sws_1303]** An `Rte_IrvIRead` API shall be created for each read InterRunnableVariable.

Description: The `Rte_IrvIRead` API provides read access to the defined InterRunnableVariables with *implicit* behavior within a component description.

The return value is used to deliver the requested data value. The return value is not required to pass error information to the user because no inter-ECU communication is involved and there will always be a readable value present.

Requirement rte_sws_3581 is valid for InterRunnableVariables with implicit and InterRunnableVariables with explicit behavior:

[rte_sws_3581] The RTE has to ensure that read accesses to an InterRunnableVariables won't deliver undefined data item values. In case write access before read access cannot be guaranteed by configuration an initial values for the InterRunnableVariable has to be written to it.

This initial value has to be an input for the RTE generator and might be initially defined in the AUTOSAR SW-C description.

Return Value: **[rte_sws_3552]** The `Rte_IrvIRead` call returns the actual value of the accessed InterRunnableVariable.

The return type of `Rte_IrvIRead` is dependent on the InterRunnableVariable data type. Thus the component does not need to use type casting to convert access the InterRunnableVariable data.

[rte_sws_3556] The return value of the `Rte_IrvIRead` API call shall pass a value.

[rte_sws_3558] The `Rte_IrvIRead` API call does not support complex data types.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.4.6. More details about InterRunnableVariables with *implicit* behavior is explained in section 4.2.4.6.1.

5.6.18 Rte_IrvIWrite

Purpose: Provide **write** access to the *InterRunnableVariables with implicit behavior* of an AUTOSAR SW-C.

Signature: **[rte_sws_3553]**

```
void
Rte_IrvIWrite_<re>_<name>([IN RTE_Instance <instance>], IN <data>)
```

Where `<re>` is the name of the runnable entity the API might be used in, `<name>` is the name of the InterRunnableVariable to access and `<data>` is the placeholder for the data the InterRunnableVariable shall be set to.

Existence: **[rte_sws_1304]** An `Rte_IrvIWrite` API shall be created for each written InterRunnableVariable.

Description: The `Rte_IrvIWrite` API provides write access to the InterRunnableVariables with *implicit* behavior within a component description. The runnable entity name in the signature allows runnable context specific optimizations.

The data given by `Rte_IrvIWrite` is dependent on the InterRunnableVariable data type. Thus the component does not need to use type casting to write the InterRunnableVariable.

The return value is unused. The return value is not required to pass error information to the user because no inter-ECU communication is involved and the value can always be written.

[rte_sws_3557] The `Rte_IrvIWrite` API call include exactly one IN parameter for the data element - which is a pass by value.

[rte_sws_3559] The `Rte_IrvIWrite` API call does not support complex data types.

Return Value: **[rte_sws_3555]** `Rte_IrvIWrite` shall have no return value.

For C/C++, requirement rte_sws_3555 means using a return type of `void`.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.4.6. Further details about InterRunnableVariables with *implicit* behavior are explained in Section 4.2.4.6.1.

5.6.19 Rte_IrvRead

Purpose: Provide **read** access to the *InterRunnableVariables with explicit behavior* of an AUTOSAR SW-C.

Signature: **[rte_sws_3560]**

```
<return>
Rte_IrvRead_<re>_<name>([IN RTE_Instance <instance>])
```

Where `<re>` is the name of the runnable entity the API might be used in, `<name>` is the name of the InterRunableVariables.

Existence: **[rte_sws_1305]** An `Rte_IrvIRead` API shall be created for each read InterRunnableVariable using explicit access.

Description: The `Rte_IrvRead` API provides read access to the defined InterRunnableVariables with *explicit* behavior within a component description.

The return value is used to deliver the requested data value. The return value is not required to pass error information to the user because no inter-ECU communication is involved and there will always be a readable value present.

Return Value: **[rte_sws_3562]** The `Rte_IrvRead` call returns the actual value of the accessed InterRunnableVariable.

The return type of `Rte_IrvRead` is dependent on the InterRunnableVariable data type. Thus the component does not need to use type casting to convert access the InterRunnableVariable data.

[rte_sws_3563] The return value of the `Rte_IrvRead` API call shall pass a value.

[rte_sws_3564] The `Rte_IrvRead` API call does not support complex data types.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.4.6. Further details about InterRunnableVariables with *explicit* behavior are explained in Section 4.2.4.6.2.

5.6.20 Rte_IrvWrite

Purpose: Provide **write** access to the *InterRunnableVariables with explicit behavior* of an AUTOSAR SW-C.

Signature: **[rte_sws_3565]**

```
void
Rte_IrvWrite_<re>_<name>([IN RTE_Instance <instance>], IN <data>)
```

Where `<re>` is the name of the runnable entity the API might be used in, `<name>` is the name of the InterRunnableVariable to access and `<data>` is the placeholder for the data the InterRunnableVariable shall be set to.

Existence: **[rte_sws_1306]** An `Rte_IrvIWrite` API shall be created for each written InterRunnableVariable using explicit access.

Description: The `Rte_IrvWrite` API provides write access to the InterRunnableVariables with *explicit* behavior within a component description.

The data given by `Rte_IrvWrite` is dependent on the InterRunnableVariable data type. Thus the component does not need to use type casting to write the InterRunnableVariable.

The return value is unused. The return value is not required to pass error information to the user because no inter-ECU communication is involved and the value can always be written.

[rte_sws_3567] The `Rte_IrvWrite` API call include exactly one IN parameter for the data element - which is a pass by value.

[rte_sws_3568] The `Rte_IrvWrite` API call does not support complex data types.

Return Value: **[rte_sws_3569]** `Rte_IrvWrite` shall have no return value.

For C/C++, requirement rte_sws_3569 means using a return type of `void`.

Notes: The runnable entity name in the signature allows runnable context specific optimizations.

The concept of InterRunnableVariables is explained in section 4.2.4.6. Further details about InterRunnableVariables with *explicit* behavior are explained in Section 4.2.4.6.2.

5.6.21 Rte_Enter

Purpose: Enter an exclusive area.

Signature: **[rte_sws_1120]**

```
void
Rte_Enter_<name>([IN Rte_Instance <instance>])
```

Where `<name>` is the exclusive area name.

Existence: **[rte_sws_1307]** An `Rte_Enter` API shall be created for each ExclusiveArea that is declared RunnableEntityCanEnterExclusiveArea.

Description: The `Rte_Enter` API call is invoked by an AUTOSAR software-component to define the start of an exclusive area.

Return Value: None.

Notes: The RTE is not required to support nested invocations of `Rte_Enter` for the same exclusive area.

[rte_sws_1122] The RTE shall permit calls to `Rte_Enter` and `Rte_Exit` to be nested as long as regions are exited in the reverse order they were entered.

Within the AUTOSAR OS an attempt to lock a resource cannot fail because the lock is already held. The lock attempt can only fail due to configuration errors (e.g. caller not declared as accessing the resource) or invalid handle. Therefore the return type from this function is `void`.

5.6.22 Rte_Exit

Purpose: Leave an exclusive area.

Signature: **[rte_sws_1123]**

```
void
Rte_Exit_<name>([IN Rte_Instance <instance>])
```

Where `<name>` is the exclusive area name.

Existence: **[rte_sws_1308]** An `Rte_Exit` API shall be created for each ExclusiveArea that is declared RunnableEntityCanEnterExclusiveArea.

Description: The `Rte_Exit` API call is invoked by an AUTOSAR software-component to define the end of an exclusive area.

Return Value: None.

Notes: The RTE is not required to support nested invocations of `Rte_Exit` for the same exclusive area.

Requirement rte_sws_1122 permits calls to `Rte_Enter` and `Rte_Exit` to be nested as long as regions are exited in the reverse order they were entered.

5.6.23 Rte_Mode

Purpose: Provides the currently active mode of a mode port.

Signature: **[rte_sws_2628]**

```
Rte_ModeType_<m>
Rte_Mode_<p>_<o>([IN Rte_Instance <instance>],
                 OUT <data>)
```

Where <m> is the ModeDeclarationGroup name, <p> is the port name, and <o> the ModeDeclarationGroupPrototype name within the sender-receiver interface categorizing the port.

Existence: **[rte_sws_2629]** An `Rte_Mode` API shall be created for each required ModeDeclarationGroupPrototype.

Description: The `Rte_Mode` API tells the AUTOSAR Software-Component which mode of a ModeDeclarationGroup of a given port is currently active. This is the information that the RTE uses for the ModeDisablingDependencies. This information does not change immediately after the reception of a mode switch indication from a `mode manager`, see section 4.4.3. But it will be the same for all `mode ports` that are connected to the same `mode port` of the `mode manager` (see rte_sws_2630).

Return Value: currently active mode of the given instance of a ModeDeclarationGroupPrototype.

Notes: None.

5.7 Runnable Entity Reference

An AUTOSAR component defines one or more “runnable entities”. A runnable entity is a piece of code with a single entry point and an associate set of data.

[rte_sws_1015] A software-component description shall provide definitions for each runnable entity.

For components implemented using C or C++ the entry point of a runnable entity is implemented by a function with global scope defined within a software-component’s source code. The following sections consider the function signature and prototype.

5.7.1 Signature

The definition of all runnable entities, whatever the `RTEEvent` that triggers their execution, follows the same basic form.

[rte_sws_1126]
```
<void|Std_ReturnType> <name>([IN Rte_Instance <instance>],
                             [role parameters])
```

Where `<name>` [4] is the symbol describing the runnable's entry point rte_sws_in_0053. The definition of the *role parameters* is defined in Section 5.7.3.

Section 5.2.6.4 contains details on a recommended naming conventions for runnable entities based on the `RTEEvent` that triggers the runnable entity. The recommended naming convention makes explicit the functions that implement runnable entities as well as clearly associating the runnable entity and the applicable data element or operation.

5.7.2 Entry Point Prototype

The RTE determines the required role parameters, and hence the prototype of the entry point, for a runnable entity based on information in the input information (see Appendix B). The entry point defined in the component source *must* be compatible with the parameters passed by the RTE when the runnable entity is triggered by the RTE and therefore the RTE generator is required to emit a prototype for the function.

[rte_sws_1132] The RTE generator shall emit a prototype for the runnable entity's entry point in the application header file.

The prototype for a function implementing the entry point of a runnable entity is emitted for both "RTE Contract" and "RTE Generation" phases. The function name for the prototype is the runnable entity's entry point. The prototype of the entry point function includes the runnable entity's instance handle and its role parameters, see Figure 5.2.

[rte_sws_1016] The function implementing the entry point of a runnable entity shall define an instance handle as the first formal parameter.

The RTE will ensure that when the runnable entity is triggered the instance handle parameter indicates the correct component instance. The remaining parameters passed to the runnable entity depend on the `RTEEvent` that triggers execution of the runnable entity.

[4] Runnable entities have two "names" associated with them in the Software-Component Template; the runnable's identifier and the entry point's symbol. The identifier is used to reference the runnable entity within the input data and the symbol used within code to identify the runnable's implementation. In the context of a prototype for a runnable entity, "name" is the runnable entity's entry point symbol.

5.7.3 Role Parameters

The *role parameters* are optional and their presence and types depend on the `RTEEvent` that triggers the execution of the runnable entity. The role parameters that are necessary for each triggering `RTEEvent` are defined in Section 5.7.5.

5.7.4 Return Value

A function in C or C++ is required to have a return type. The RTE only uses the function return value to return application error codes of a server operation.

[rte_sws_1130] A function implementing a runnable entity entry point shall only have the return type `Std_ReturnType`, if the runnable entity represents a server operation and the AUTOSAR interface description of that client server communication lists potential application errors. All other functions implementing a runnable entity entry point shall have a return type of `void`.

5.7.5 Triggering Events

The RTE is the *sole* entity that can trigger the execution of a runnable entity. The RTE triggers runnable entities in response to different `RTEEvents`.

The most basic `RTEEvent` that can trigger a runnable entity is the `TimingEvent` that causes a runnable entity to be periodically triggered by the RTE. In contrast, the remaining `RTEEvents` that can trigger runnable entities all occur as a result of communication activity or as a result of mode switches.

All runnable entities, whatever `RTEEvent` triggers their execution, shall be defined in the RTE input. (rte_sws_1015).

The following subsections describe the conditions that can trigger execution of a runnable entity. For each triggering event the signature of the function (the "entry point") that implements the runnable entity is defined. The signature definition includes two classes of parameters for each function;

1. The instance handle – the parameter type is always `Rte_Instance`. (rte_sws_1016)
2. The role parameters – used to pass information required by the runnable entity as a consequence of the triggering condition. The presence (and number) of role parameters depends solely on the triggering condition.

5.7.5.1 TimingEvent

Purpose:	Trigger a runnable entity periodically at a rate defined within the software-component description.
Signature:	**[rte_sws_1131]**

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

5.7.5.2 ModeSwitchEvent

Purpose: Trigger of a runnable entity as a result of a mode switch. See also sections 4.4.3 and 4.4.5 for reference.

Signature: **[rte_sws_2512]**

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

5.7.5.3 AsynchronousServerCallReturnsEvent

Purpose: Triggers a runnable entity used to "collect" the result and status information of an asynchronous client-server operation.

Signature: **[rte_sws_1133]**

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

Notes: The runnable entity triggered by an AsynchronousServerCallReturnsEvent `RTEEvent` should use the `Rte_Result` API to actually receive the result and the status of the server operation.

5.7.5.4 DataReceiveErrorEvent

Purpose: Triggers a runnable entity used to "collect" the error status of a `data element` with "data" semantics (isQueued = false) on the receiver side.

Signature: **[rte_sws_1359]**

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

Notes: The runnable entity triggered by a DataReceiveErrorEvent `RTEEvent` should use the `Rte_IStatus` API to actually read the status.

5.7.5.5 OperationInvokedEvent

Purpose: An `RTEEvent` that causes the RTE to trigger a runnable entity whose entry point provides an implementation for a client-server operation. This event occurs in response to a received request from a client to execute the operation.

Signature: **[rte_sws_1166]**

```
<void|Std_ReturnType> <name>([IN Rte_Instance <instance>],
                             [IN           <portDefArg 1>, ...
                              IN           <portDefArg n>],
                             [IN|INOUT|OUT] <param 1>, ...
                             [IN|INOUT|OUT] <param n>)
```

Where `<portDefArg 1>, ..., <portDefArg n>` represent the port-defined argument values (see Section 4.3.2.4) and `<param 1>, ... <param n>` indicates the operation IN, INOUT and OUT parameters.

The data type of each port defined argument is taken from the software component template, as defined in rte_sws_in_1361.

Note that the port-defined argument values are optional, depending upon the server's internal behavior.

Return Value: If the AUTOSAR interface description of the client server communication lists possible error codes, these are returned by the function using the return type `Std_ReturnType`. If no error codes are defined for this interface, the return type shall be `void` (see rte_sws_1130).

5.7.5.6 DataReceivedEvent

Purpose: A runnable entity triggered by the RTE to receive and process a signal received on a sender-receiver interface.

Signature: **[rte_sws_1135]**
`void <name>([IN Rte_Instance <instance>])`

Notes: The data or event is not passed as an additional parameter. Instead, the previously described reception API should be used to access the data/event. This approach permits the same signature for runnables that are triggered by time (TimingEvent) or data reception.

Caution: For intra-ECU communication, the DataReceivedEvent is fired after each completed write operation to the shared data. While for inter-ECU communication, the DataReceivedEvent is fired by the RTE after a callback from COM due to data reception. Over a physical network, 'data' is commonly transmitted periodically and hence not only will the latency and jitter of DataReceivedEvents vary depending on whether a configuration uses intra or inter-ECU communication, but also the number and frequency of these RTEEvents may change significantly. This means that a TimingEvent should be used to periodically activation of a runnable rather than relying on the periodic transmission of data.

5.7.5.7 DataSendCompletedEvent

Purpose: A runnable entity triggered by the RTE to receive and process transmit acknowledgment notifications.

Signature:	**[rte_sws_1137]** `void <name>([IN Rte_Instance <instance>])`
Notes:	The runnable entity triggered by a DataSendCompletedEvent `RTEEvent` should use the `Rte_Feedback` API to actually receive the status of the acknowledgement.

5.7.6 Reentrancy

A runnable entity is declared within a software-component type. The RTE ensures that concurrent activation of same instance of a runnable entity is only allowed if the runnables attribute ”canBeInvokedConcurrently” is set to TRUE (see Section 4.2.5).

[rte_sws_3590] The RTE has to reject configurations where concurrent activation of an runnable entity is requested and the associated attribute ”canBeInvokedConcurrently” is not set to TRUE.

When a software-component is multiply instantiated each separate instance has its own instance of the runnable entities in the software-component. Whilst instances of a software-component are independent, the runnable entities instances share the same code (rte_sws_2017).

> **Example 5.19**
>
> Consider a component `c1` with runnable entity `re1` and entry point `ep` that is instantiated twice on the same ECU.
>
> The two instances of `c1` each has a separate *instance* of `re1`. Software-component instances are scheduled independently and therefore each instance of `re1` could be concurrently executing `ep`.

The potential for concurrent execution of runnable entities when multiple instances of a software-component are created means that each entry point should be reentrant.

[rte_sws_3749] The RTE has to reject configurations where multiple instantiation of an AUTOSAR SW-Cs is requested and the associated attribute ”supportsMultipleInstantiation” is not set to TRUE.

5.8 RTE Lifecycle API Reference

This section documents the API functions used to start and stop the RTE.

[rte_sws_1139] RTE Lifecycle API functions shall not be invoked from AUTOSAR software-components.

5.8.1 Rte_Start

Purpose:	Initialize the RTE itself.

Signature: **[rte_sws_2569]**
`Std_ReturnType Rte_Start(void)`

Existence: **[rte_sws_1309]** The `Rte_Start` API is always created.

Description: `Rte_Start` is intended to allocate and initialise system resources and communication resources used by the RTE.

[rte_sws_2582] `Rte_Start` shall be called only once by the EcuStateManager after the basic software modules required by RTE are initialized. These modules include:

- OS
- COM
- memory services

The `Rte_Start` API shall not be invoked from AUTOSAR software components.

[rte_sws_2585] `Rte_Start` shall return within finite execution time – it must not enter an infinite loop.

`Rte_Start` may be implemented as a function or a macro.

Return Value: If the allocation of a resource fails, `Rte_Start` shall return with an error.

- **[rte_sws_1261]** `RTE_E_OK` – No error occurred.
- **[rte_sws_1262]** `RTE_E_LIMIT` – An internal limit has been exceeded. The allocation of a required resource has failed.

Notes: `Rte_Start` is declared in the lifecycle header file `Rte_Main.h`. The initialization of AUTOSAR software-components takes place after the termination of `Rte_Start` and is triggered by a mode change event on entering run state.

5.8.2 Rte_Stop

Purpose: finalize the RTE itself

Signature: **[rte_sws_2570]**
`Std_ReturnType Rte_Stop(void)`

Existence: **[rte_sws_1310]** The `Rte_Stop` API is always created.

Description: `Rte_Stop` is used to finalize the RTE itself. This service releases all system and communication resources allocated by the RTE.

[rte_sws_2583] `Rte_Stop` shall be called by the EcuStateManager before the basic software modules required by RTE are shut down. These modules include:

- OS
- COM
- memory services

`Rte_Stop` shall not be called by an AUTOSAR software component.

[rte_sws_2584] `Rte_Stop` shall return within finite execution time.

`Rte_Stop` may be implemented as a function or a macro.

Return Value:
- **[rte_sws_1259]** `RTE_E_OK` – No error occurred.
- **[rte_sws_1260]** `RTE_E_LIMIT` – a resource could not be released.

Notes: `Rte_Stop` is declared in the lifecycle header file `Rte_Main.h`.

5.9 RTE Call-backs Reference

This section documents the call-backs that are generated by the RTE that must be invoked by other components, such as the communication service, and therefore must have a well-defined name and semantics.

[rte_sws_1165] A call-back implementation created by the RTE generator is not permitted to block.

Requirement rte_sws_1165 serves to constrain RTE implementations so that all implementations can work with all basic software.

5.9.1 RTE-COM Message Naming Conventions

The COM signals used for communication are defined in the meta-model (Section B).

[rte_sws_3007] The RTE shall initiate an inter-ECU transmission using the COM API with the handle id of the corresponding COM signal for primitive data element rte_sws_in_0063.

[rte_sws_3008] The RTE shall initiate an inter-ECU transmission using the COM API with the handle id of the corresponding COM signal group for complex data element or operation arguments rte_sws_in_0064.

5.9.2 Communication Service Call-backs

Purpose: Implement the call-back function that AUTOSAR COM invokes as a result of inter-ECU communication, where:

- A data item/event is ready for reception by a receiver.
- A transmission acknowledgment shall be routed to a sender.

- An operation shall be invoked by a server.
- The result of an operation is ready for reading by a client.

Signature: **[rte_sws_3000]**

`COMCallback(CallbackRoutineName)`

Where `CallbackRoutineName` is the name of the callback function (refer to Section 5.9.3 for details on the naming convention).

Description: The `COMCallback` API provided by COM takes the parameter `<callback-RoutineName>`.

Return Value: The `COMCallback` API is defined as returning no value.

5.9.3 Naming convention of callbackRoutineName

In the following table, the naming convention of `<callbackRoutineName>` are defined:

Calling Situation	callbackRoutineName	Comments
A primitive data item/event is ready for reception by a receiver.	**[rte_sws_3001]** `Rte_COMCbk_<sn>`	<sn> is the name of the COM signal. This callback function indicates that the signal of the primitive data item/event or the single argument of an operation is ready for reception.
A transmission acknowledgment of a primitive data item/event shall be routed to a sender.	**[rte_sws_3002]** `Rte_COMCbkTAck_<sn>`	“TAck” is literal text indicating transmission acknowledgment. This callback function indicates that the signal of the primitive data item/event is already handed over by COM to the PDU router.
A transmission error notificatoin of a primitive data item/event shall be routed to a sender.	**[rte_sws_3775]** `Rte_COMCbkTErr_<sn>`	“TErr” is literal text indicating transmission error. This callback function indicates that an error occurred when the signal of the primitive data item/event was handed over by COM to the PDU router.
A signal invalidation of a primitive data item shall be routed to a receiver.	**[rte_sws_2612]** `Rte_COMCbkInv_<sn>`	“Inv” is literal text indicating signal invalidation. This callback function indicates that COM has received a signal and parsed it as “invalid”.
A signal of a primitive data item is outdated. No new data is available.	**[rte_sws_2610]** `Rte_COMCbkTOut_<sn>`	“TOut” is literal text indicating signal time out. This callback function indicates that the aliveTimeout after the last successful reception of the signal of the primitive data item/event has expired (`data element outdated`).

Calling Situation	callbackRoutineName	Comments
A complex data item/event or the arguments of an operation is ready for reception by a receiver.	**[rte_sws_3004]** `Rte_COMCbk_<sg>`	`<sg>` is the name of the COM signal group, which contains all the signals of the complex data item/event or an operation. This callback function indicates that the signals of the complex data item/event or the arguments of an operation are ready for reception.
A transmission acknowledgment of a complex data item/event shall be routed to a sender.	**[rte_sws_3005]** `Rte_COMCbkTAck_<sg>`	"TAck" is literal text indicating transmission acknowledgment. This callback function indicates that the signals of the complex data item/event is already handed over by COM to the PDU router.
A transmission error notificatoin of a complex data item/event shall be routed to a sender.	**[rte_sws_3776]** `Rte_COMCbkTErr_<sg>`	"TErr" is literal text indicating transmission error. This callback function indicates that an error occurred when the signal of the complex data item/event was handed over by COM to the PDU router.
A signal group of a complex data item is outdated. No new data is available.	**[rte_sws_2611]** `Rte_COMCbkTOut_<sg>`	"TOut" is literal text indicating signal time out. This callback function indicates that the aliveTimeout after the last successful reception of the signal group carrying the complex data item has expierd (`data element outdated`).

Table 5.7: RTE COM Callback Function Naming Conventions

Where:

- `<sn>` is a COM signal name.
- `<sg>` is a COM signal group name.

5.10 VFB Tracing Reference

The RTE's "VFB Tracing" functionality permits the monitoring of AUTOSAR signals as they are sent and received across the VFB.

The RTE operates in at least two builds (some implementations may provide more than two builds). The first, production, does not enable VFB tracing whereas the second, debug, can be configured to trace some or all "interesting events".

[rte_sws_1327] The RTE generator shall support 'production' build where no VFB events are traced.

[rte_sws_1328] The RTE generator shall support 'debug' build that traces (configured) VFB events.

The RTE generator's 'debug' build is enabled or disabled through definitions in the RTE Configuration file rte_sws_1322 and rte_sws_1323. Note that this 'debug' build is intended to enable debugging of software components and not the RTE itself.

5.10.1 Prinicple of Operation

The "VFB Tracing" mechanism is designed to offer a lightweight means to monitor the interactions of AUTOSAR software-components with the VFB.

The VFB tracing in 'debug' build is implemented by a series of "hook" functions that are invoked automatically by the generated RTE when "interesting events" occur. Each hook function corresponds to a single event.

The supported trace events are defined in Section 5.10.2. A mechanism is described in Section 5.10.3 for configuring which of the many potential trace events are of interest.

5.10.2 Trace Events

5.10.2.1 RTE API Trace Events

RTE API trace events occur when an AUTOSAR software-component interacts with the generated RTE API.

5.10.2.1.1 RTE API Start

Description: RTE API Start is invoked by the RTE when an API call is made by a component.

Signature: **[rte_sws_1238]**

```
void Rte_<api>Hook_<c>_<ap>_Start([const Rte_CDS_<c>>*, ]<param>)
```

Where `<api>` is the RTE API Name (Write, Call, etc.), `<c>` is the component type name and `<ap>` the access point name (e.g. port and data element or operation name, exclusive area name, etc.). The parameters of the API are the same as the corresponding RTE API. As with the API itself, the instance handle is included if and only if the software component's SupportsMultipleInstantiation (rte_sws_in_0004) attribute is set to true. Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous.

5.10.2.1.2 RTE API Return

Description: RTE API Return is a trace event that is invoked by the RTE just before an API call returns control to a component.

Signature: **[rte_sws_1239]**

```
void Rte_<api>Hook_<c>_<ap>_Return([const Rte_CDS_<c>*, ]<param>)
```

Where <api> is the RTE API Name (Write, Call, etc.), <c> is the component type name and <ap> the access point name (e.g. port and data element or operation name, exclusive area name, etc.). The parameters of the API are the same as the corresponding RTE API. As with the API itself, the instance handle is included if and only if the software component's SupportsMultipleInstantiation (rte_sws_in_0004) attribute is set to true. Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous.

5.10.2.2 COM Trace Events

COM trace events occur when the generated RTE interacts with the AUTOSAR communication service.

5.10.2.2.1 Signal Transmission

Description: 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_UpdateShadowSignal` is invoked.

Signature: **[rte_sws_1240]**

```
void Rte_ComHook_<signalName>_SigTx(<data>)
```

Where `<signalName>` is the system signal name and `<data>` a pointer to the signal name to be transmitted.

5.10.2.2.2 Signal Reception

Description: A trace event indicating a successful attempt to read an Inter-ECU signal by the RTE. Invoked by the RTE after return from `Com_ReceiveSignal` or `Com_ReceiveShadowSignal`.

Signature: **[rte_sws_1241]**

```
void Rte_ComHook_<signalName>_SigRx(<data>)
```

Where `<signalName>` is the system signal name and `<data>` a pointer to the signal data received.

5.10.2.2.3 COM Callback

Description: A trace event indicating the start of a COM call-back. Invoked by generated RTE code on entry to the COM call-back.

Signature: **[rte_sws_1242]**
`void Rte_ComHook_<signalName>(void)`

Where `<signalName>` is the system signal name.

5.10.2.3 OS Trace Events

OS trace events occur when the generated RTE interacts with the AUTOSAR operating system.

5.10.2.3.1 Task Activate

Description: A trace event that is invoked by the RTE immediately prior to the activation of a task containing runnable entities.

Signature: **[rte_sws_1243]**
`void Rte_Task_Activate(TaskType task)`

Where `task` is the OS's handle for the task.

5.10.2.3.2 Task Dispatch

Description: A trace event that is invoked immediately an RTE generated task (containing runnable entities) has commenced execution.

Signature: **[rte_sws_1244]**
`void Rte_Task_Dispatch(TaskType task)`

Where `task` is the OS's handle for the task.

5.10.2.3.3 Set OS Event

Description: A trace event invoked immediately before generated RTE code attempts to set an OS Event.

Signature: **[rte_sws_1245]**
`void Rte_Task_SetEvent(TaskType task, EventType ev)`

Where `task` is the OS's handle for the task for which the event is being set and `ev` the OS event mask.

5.10.2.3.4 Wait OS Event

Description: Invoked immediately before generated RTE code attempts to wait on an OS Event. This trace event does *not* indicate that the caller has suspended execution since the OS call may immediately return if the event was already set.

Signature: **[rte_sws_1246]**

```
void Rte_Task_WaitEvent(TaskType task, EventType ev)
```

Where `task` is the OS's handle for the task (that is waiting for the event) and `ev` the OS event mask.

5.10.2.3.5 Received OS Event

Description: Invoked immediately after generated RTE code returns from waiting on an event.

Signature: **[rte_sws_1247]**

```
void Rte_Task_WaitEventRet(TaskType task, EventType ev)
```

Where `task` is the OS's handle for the task (that was waiting for an event) and `ev` the event mask indicating the received event.

5.10.2.4 Runnable Entity Trace Events

Runnable entity trace events occur when a runnable entity is started.

5.10.2.4.1 Runnable Entity Invocation

Description: Event invoked by the RTE just before execution of runnable entry starts via its entry point. This trace event occurs after any copies of data elements are made to support the `Rte_IRead` API Call.

Signature: **[rte_sws_1248]**

```
void Rte_Runnable_<c>_<reName>_Start([const RTE_CDS_<c>*])
```

Where <c> is the SW-C type name and `reName` the runnable entity name. The instance handle is included if and only if the software component's SupportsMultipleInstantiation (rte_sws_in_0004) attribute is set to true. Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous.

5.10.2.4.2 Runnable Entity Termination

purpose: Event invoked by the RTE immediately execution returns to RTE code from a runnable entity. This trace event occurs before any write-back of data elements are made to support the `Rte_IWrite` API Call.

Signature: **[rte_sws_1249]**
`void Rte_Runnable_<c>_<reName>_Return([const Rte_CDS_<c>*])`

Where <c> is the SW-C type name and `reName` the runnable entity name. The instance handle is included if and only if the software component's SupportsMultipleInstantiation (rte_sws_in_0004) attribute is set to true. Note that `Rte_Instance` cannot be used directly, as there will be pointers to multiple components' structure types within the single VFB Tracing header file, and `Rte_Instance` would therefore be ambiguous.

5.10.3 Configuration

The VFB tracing mechanism works by the RTE invoking the tracepoint *hook* function whenever the tracing event occurs.

The support trace events and their hook function name and signature are defined in Section 5.10.2. There are many potential trace events and it is likely that only a few will be of interest at any one time. Therefore The RTE generator supports a mechanism to configure which trace events are of interest.

In order to minimise RTE Overheads, trace events that are not enabled should have no run-time effect on the generated system.

[rte_sws_1235] An RTE generator shall ensure that unconfigured trace events have no run-time effect.

This mechanism used to implement rte_sws_1235 uses the generated conditional definitions within the VFB Tracing header file (see Section 5.3.5) and the user supplied definitions from the RTE Configuration Header file (see Section 5.3.6).

The definition of trace event hook functions is contained within user code. If a definition is encapsulated within a `#if` block, as follows, the definition will automatically be omitted when the trace event is disabled.

```
#if !defined(<trace event>)
void <trace event>(<params>)
{
  /* Function definition */
}
#endif
```

The configuration of which individual trace events are enabled is entirely under the control of the user via the definitions included in the RTE Configuration header file.

5.10.4 Interaction with Object-code Software-Components

VFB tracing is only available during the “RTE Generation” phase rte_sws_1319 and therefore hook functions never appear in an application header file created during “RTE Contract” phase. However, object-code software-components are compiled against the “RTE Contract” phase header and can therefore only trace events that are inserted into the generated RTE. In particular they cannot trace events that require invocation of hook functions to be inserted into the API mapping such as the `Rte_Pim` API. However, many trace events are applicable to object-code software-components including trace events related to the explicit communication API, to task activity and for runnable entity start and stop.

This approach means that the external interactions of the object-code software-component can be monitored without requiring modification of the delivered object-code and without revealing the internal activity of the software-component. The approach is therefore considered to be consistent with the desire for IP protection that prompts delivery of a software-component as object-code. Finally, tracing can easily be disabled for a production build without invalidating tests of the object-code software-component.

A Metamodel Restrictions

This chapter lists all the restrictions to the AUTOSAR meta-model this version of the AUTOSAR RTE specification document relies on.

A.1 Restriction concerning WaitPoint

1. **[rte_sws_1358]** A runnable entity cannot have WaitPoint connected to the following RTEEvents:

    - OperationInvokedEvent
    - ModeSwitchEvent
    - TimingEvent
    - ExternalEvent
    - DataReceiveErrorEvent

    The runnable can only be started with these events.

    Rational: For OperationInvokedEvents, ModeSwitchEvents and TimingEvents it suffices to allow the activation of a runnable entity. ExternalEvents are not supported by the RTE SWS of AUTOSSAR Release 2.0.

A.2 Restriction concerning RTEEvent

1. **[rte_sws_3526]** The RTE shall **NOT** support a runnable entity triggered by an RTEEvent *OperationInvokedEvent* to be triggered by any other RTEEvent except for other *OperationInvokedEvents* of compatible operations.

    Rational: The signature of the runnable entity is dependent on its connected RTEEvent.

2. **[rte_sws_3010]** One runnable entity shall only be resumed by one single RTEEvent on its WaitPoint. The RTE doesn't support the WaitPoint of one runnable entity connected to several RTEEvents.

    Rational: The WaitPoint of the runnalbe entity is caused by calling of the RTE API. One runnable entity can only call one RTE API at a time, and so it can only wait for one RTEEvent.

A.3 Restriction concerning isQueued attribute of DataElementPrototype

1. **[rte_sws_3012]** Access with DataReadAccess is only allowed for DataElementPrototypes with their isQueued attribute set to false.

   Rational: By access with DataReadAccess always the last value of the DataElementPrototype will be read in the runnable. There is no meaning to provide a queue of values by DataReadAccess.

2. **[rte_sws_3018]** RTE does not support receiving with WaitPoint for DataElementPrototypes with their isQueued attribute set to false.

   Rational: ”isQueued=false” indicates that the receiver shall not wait for the DataElementPrototype.

3. **[rte_sws_3013]** All the DataReceivePoints refering to one DataElementPrototype through one RPortPrototype are considered to have the same behaviour by reception.

   Rational: The API RTEReceive/RTERead is dependent on the port name and the data element name, not on the DataReceivePoints. For each combination of one data element prototype and one port only one API will be generated and implemented for reception.

4. **[rte_sws_3011]** All the DataSendPoints refering to one DataElementPrototype through one PPortPrototype are considered to have the same behaviour by sending and acknowledgement reception.

   Rational: The API RTESend/RTEWrite is dependent on the port name and the DataElementPrototype name, not on the DataSendPoints. For each combination of one DataElementPrototype and one port only one API will be generated and implemented for sending or acknowledgement reception.

A.4 Restriction concerning ServerCallPoint

1. **[rte_sws_3014]** All the ServerCallPoints refering to one OperationPrototype through one RPortPrototype are considered to have the same behaviour by calling service.

   Rational: The API RTECall is dependent on the port name and the operation name, not on the ServerCallPoints. For each combination of one operation and one port only one API will be generated and implemented for calling a service.

A.5 Restriction concerning multiple instantiation of software components

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

   Rationale: For AUTOSAR release 2 it was decided to solely concentrate on code sharing and not to support code duplication.

A.6 Restriction concerning runnable entity

1. **[rte_sws_3016]** The RTE only supports runnable entity of category 1 and 2, the RTE doesn't support runnable entity of category 3.

   Rationale: For AUTOSAR release 2 it was decided only to support runnable entity of category 1 and 2, not to support runnable entity of category 3.

2. **[rte_sws_3527]** The RTE does NOT support multiple Runnable Entities sharing the same entry point (symbol attribute of RunnableEntity).

   Rationale: The handle to data shared by DataReadAccess and DataWriteAccess has to be coded in the runnable code. An alternative would be an additional parameter to the runnable (a runnable handle) to provide this indirection information.

A.7 Restriction concerning runnables triggered by ModeSwitchEvents

1. **[rte_sws_2500]** Only a category 1 runnable may reference a ModeSwitchEvent to start on. The RTE generator shall reject configurations with category 2 or 3 runnables connected to ModeSwitchEvents.

   Rationale: The runnables, triggered by a ModeSwitchEvent are executed on the transitions between different modes. Thus, they have to be executed within final execution time. A switch of mode disabelings of runnables can only be executed by the RTE, when all runnables have terminated that have been started due to leaving the old mode.

A.8 Restriction concerning OperationPrototype for Inter-ECU C/S communication

1. **[rte_sws_3575]** For Inter-ECU Client-Server communication the Client-Server call needs at least one paremeter for each direction, in and out.

   This may be at least

- one in and one out parameter or
- one in/out parameter or
- one in parameter and the use of application errors.

Rational: For Inter-ECU communication a Client-Server communication call has to be mapped (at least) to two signal communication calls because COM only supports Sender/Receiver communication which is a one-way communication. Client-Server communication needs communication in both directions between the two ECUs. In Client-Server calls with at least one in and one out parameter these two signal communication calls are present - one for each communication direction.
The limitation exists because a Client/Server communication protocol is missing.

This limitation is **not** valid for Intra-ECU communication!

A.9 Restriction concerning InterRunnableVariables

1. **[rte_sws_3518]** The usage of *InterRunnableVariables with implicit behavior* shall be valid for category 1a and 1b Runnable entities only.

   Rational: The update of *InterRunnableVariables with implicit behavior* done during a Runnable execution shall be made available to other Runnables after the Runnable execution has terminated (see rte_sws_3584). This limitation is not valid for *InterRunnableVariables with explicit behavior*.

   Runnable termination is not guaranteed for Runnables of category 2 or 3.

2. **[rte_sws_3588]** *InterRunnableVariables* don't support complex data types.

   Rational: In those cases when complex data is used for Intra AUTOSAR SW-C communication it should be sufficient to apply ExclusiveAreas (see section *4.2.4.5* and API in section *5.6.21* and *5.6.22*) to force the RTE guaranteeing data consistency.

   The mass of InterRunnableVariables is expected to be of primitive type. Support for complex data might be added in a later release. Reference passing would be required. Careful discussion of API naming convention for access to complete structures, substructures, subsubstructures, .. and single elements is needed before.

3. **[rte_sws_3591]** *InterRunnableVariables* don't support the AUTOSAR primitive type string

   Rational: In those cases when a string is used for Intra AUTOSAR SW-C communication it should be sufficient to apply ExclusiveAreas (see section *4.2.4.5* and API in section *5.6.21* and *5.6.22*) to force the RTE guaranteeing data consistency.

The mass of InterRunnableVariables is expected to be of other primitive types than stings. Support for strings might be added in a later release together with support of complex data. Both require reference passing. Also see rte_sws_3588.

A.10 Restriction concerning InternalBehavior

1. **[rte_sws_5034]** There shall only be one *InternalBehavior* provided for each *AtomicSoftwareComponentType*.

   Rational: For the generation of the application header file not only the *AtomicSoftwareComponentType* but also the *InternalBehavior* is relevant. In case two implementation for the same *AtomicSoftwareComponentType* – but different *InternalBehavior* – are mapped to the same ECU two application header files for the same *AtomicSoftwareComponentType* would be required. In this document release the application header file is defined based on the *AtomicSoftwareComponentType*, therefore it is not allowed to specify different *InternalBehavior* additionally.

   In a future release the application header file shall be based on the *InternalBehavior*, then this restriction is not valid anymore.

A.11 Restriction concerning Initial Value

1. **[rte_sws_4525]** Each instance of a data element within one ECU is imposed to use identical init values.

   Rationale: In the meta model init values are specified in the data receiver com spec. Since a separate data receiver com spec exists for each port that categorizes a specific interface, it would be (theoretically) possible to define a different init value for a certain data element in each port. But COM allows only one init value per signal.

A.12 Restriction concerning PerInstanceMemory

1. **[rte_sws_3790]** The `<typeDefinition>` attribute of a PerInstanceMemory is not allowed to contain a function pointer.

   Rationale: Using the typedefinition `typedef <typedefinition> <typename>` does not work for function pointers.

B Required Input Information

This chapter lists all the input information necessary for the RTE generator in a tabular form. The meanings of the individual field entries are described in the following table:

Requirement ID	Unique ID of the RTE SWS input requirement.
Object identifier	Unique identifier in the RTE SWS representing the metamodel object. It is used like a variable in the RTE SWS standing for the content of an instance of the described metamodel object.
Object information	Necessary object information required for RTE generation in terms of a short description.
Description	Description of the required object information in more detail. It may contain a listing of the possible values of the required input information and constraints.
Rationale	Reason why the described metamodel object is needed as an input to the RTE generation.
Template metamodel path	Metamodel path of the object in an AUTOSAR template, e. g. ”AUTOSAR Software Component Template”[18] or ”System Template” [15].
Required by	Lists all RTE SWS requirements that depend on the existence of the described metamodel object.
Contract phase	Specifies whether the input information is already required for the contract phase.

”M2” in the template metamodel path means ”metamodel” level (see ”AUTOSAR Template Modeling Guide” [17]). This document is especially important to understand the specific semantics of the AUTOSAR metamodel (like the semantics of the ”instanceRef” and ”isOfType” associations).

In certain cases, some attributes of a class are not given directly in the class-table, if they are inherited from the base classes. For example, attribute ”Identifier” of class ”ComponentType” is not included in the class-table, because it is inherited from the base class ”ARElement”, which again inherits the attribute from its base class ”Identifiable”.

B.1 SWC and instance

Requirement ID	**[rte_sws_in_0001]**
Object identifier	**SwcTypeName**
Object information	Name of each SWC type
Description	Defines the name of the software component type. Shall be unique within the ECU.
Rationale	To define the API mapping in the Application Header File. Define the Component Data Structure in the generated RTE.

Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Components ::ComponentType::Identifier
Required by	rte_sws_1003 rte_sws_1143 rte_sws_1155 rte_sws_1348 rte_sws_3714 rte_sws_3731
Contract phase	Yes

Requirement ID	**[rte_sws_in_0002]**
Object identifier	**SwcImplementationLanguage**
Object information	Implementation language of each SWC
Description	For the implementation language of software components currently only C/C++ are supported.
Rationale	To define the using of C linkage in the Application Header File
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Implementation ::programmingLanguage
Required by	rte_sws_1011
Contract phase	No

Requirement ID	**[rte_sws_in_0003]**
Object identifier	**SwcSourceCodeDelivery**
Object information	Source Code availability of the SWC
Description	Whether or not the source code is available for a SWC
Rationale	To decide if the Application Header File can be optimized again by RTE-Gen phase.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Implementation ::Code::type
Required by	rte_sws_1216
Contract phase	No

Requirement ID	**[rte_sws_in_0004]**
Object identifier	**supportsMultipleInstantiation**
Object information	Multi-Instantiability of the SWC
Description	Whether the SWC can be multiply instantiated
Rationale	To define the API mapping in the Application Header File.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::supportsMultipleInstantiation
Required by	rte_sws_2008 rte_sws_2009 rte_sws_3706 rte_sws_3707

Contract phase	Yes

Requirement ID	**[rte_sws_in_0009]**
Object identifier	**PerInstanceMemoryName**
Object information	Name of each PerInstanceMemory when attribute supportsMultipleInstantiation==TRUE
Description	The name of a PerInstanceMemory shall be unique within the SWC.
Rationale	To define the name of the PerInstanceMemory handle and the API mapping in the Application Header File and allocate the PerInstanceMemory in the generated rte.c.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::PerInstanceMemory::Identifier
Required by	rte_sws_1118 rte_sws_2305 rte_sws_2301
Contract phase	Yes

Requirement ID	**[rte_sws_in_0067]**
Object identifier	**PerInstanceMemoryType**
Object information	Name of the type of each PerInstanceMemory when attribute supportsMultipleInstantiation==TRUE
Description	The type name of a PerInstanceMemory.
Rationale	To define the type of the PerInstanceMemory handle in the Application Header File and allocate the PerInstanceMemory in the generated rte.c.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::PerInstanceMemory::type
Required by	rte_sws_1118 rte_sws_2303 rte_sws_2302
Contract phase	Yes

Requirement ID	**[rte_sws_in_0068]**
Object identifier	**PerInstanceMemoryTypeDef**
Object information	Type definition of each PerInstanceMemory when attribute supportsMultipleInstantiation==TRUE
Description	The type definition of a PerInstanceMemory shall be in valid c-syntax.
Rationale	To define the type of the PerInstanceMemory in the Application Header File.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::PerInstanceMemory::typeDefinition
Required by	rte_sws_1118 rte_sws_2304
Contract phase	Yes

Requirement ID	**[rte_sws_in_0010]**
Object identifier	**XmlBasedDescription**
Object information	XML-based description
Description	AUTOSAR software component template is using XML
Rationale	XML descriptions are used to exchange information between the different steps in the AUTOSAR development.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate
Required by	rte_sws_2053
Contract phase	Yes

Requirement ID	**[rte_sws_in_3750]**
Object identifier	**RequiredRteOperatingMode**
Object information	Required RTE Operating Mode
Description	An AUTOSAR software component shall indicate its required operating mode.
Rationale	Based on this attribute the RTE Generator can perform optimizations.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::Implementation::RTEVendor
Required by	rte_sws_1234
Contract phase	No

Requirement ID	**[rte_sws_in_5013]**
Object identifier	**Constants**
Object information	Published Constants
Description	Each constant defined in the SW-Component description will be accessed and published.
Rationale	The Application Header File shall make visible the constants encoutered in the input using the appropriate AUTOSAR data-types.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Datatype::Constants::Constant
Required by	rte_sws_1196 rte_sws_1275
Contract phase	Yes

Requirement ID	**[rte_sws_in_5015]**
Object identifier	**CharacteristicName**

Object information	Name of the characteristic definition
Description	Defines the characteristics used for this SWC.
Rationale	The characteristic access is defined in the Application Header File.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Components ::Characteristic::Identifier
Required by	rte_sws_1252 rte_sws_3737
Contract phase	Yes

Requirement ID	**[rte_sws_in_0065]**
Object identifier	**CharacteristicDataSource**
Object information	Source of characteristics data for each SWC instance
Description	RTE Generator will ensure that the `Rte_Ctc_xxx` APIs return the given pointer to the appropriate SWC instance.
Rationale	For correct configuration data of each software component instance
Template meta-model path	ECUC
Required by	rte_sws_1335
Contract phase	No

B.2 Runnable entity and task

Requirement ID	**[rte_sws_in_0012]**
Object identifier	**RunnableEntityName**
Object information	Name of each runnable entity
Description	Shall be unique within the SWC
Rationale	To define the API in the Application Header File.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::RunnableEntity::Identifier
Required by	rte_sws_3733 rte_sws_3741 rte_sws_3744
Contract phase	Yes

Requirement ID	**[rte_sws_in_0013]**
Object identifier	**RunnableEntityToTaskMapping**
Object information	Mapping of runnables to OS tasks
Description	Defines the mapping of the Runnbale Entity instances to OS Tasks.

Rationale	Generate the task body content.
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::Tasks ::RunnableEntityMapping
Required by	rte_sws_2204 rte_sws_2205 rte_sws_2251
Contract phase	No

Requirement ID	**[rte_sws_in_5012]**
Object identifier	**TaskBodyName**
Object information	Name of the generated task body
Description	The names of the generated task bodies have to be unique on one ECU.
Rationale	Generate the C module containing the task body.
Template meta-model path	ECUC
Required by	rte_sws_1257 rte_sws_2251 rte_sws_4014
Contract phase	No

Requirement ID	**[rte_sws_in_0040]**
Object identifier	**OSObjects**
Object information	ECU configuration parameters of the AUTOSAR OS
Description	The RTE generator needs access to the ECU-Configuration parameters of the AUTOSAR OS.
Rationale	Determine the type of a task
Template meta-model path	ECUC
Required by	rte_sws_2251 rte_sws_4014
Contract phase	No

Requirement ID	**[rte_sws_in_0014]**
Object identifier	**RunnableEntitySequence**
Object information	Sequences of Runnable Entites in each OS task
Description	Defines the sequence the Runnable Entites are called within one task body.
Rationale	Generate the task body content.
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::Tasks ::PositionInTask

Required by	rte_sws_2207
Contract phase	No

Requirement ID	**[rte_sws_in_0053]**
Object identifier	**EntryPointSymbol**
Object information	Symbol describing a runnable's entry point
Description	A runnable is represented as a function in C/C++ code. This symbol represents the entry point of the function.
Rationale	The entry point symbol is considered to be the API name of the runnable.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::RunnableEntity::symbol
Required by	rte_sws_1126 rte_sws_1131 rte_sws_1133 rte_sws_1135 rte_sws_1137 rte_sws_1166 rte_sws_2512
Contract phase	Yes

Requirement ID	**[rte_sws_in_0015]**
Object identifier	**OsTaskPriority**
Object information	Priority of each OS task
Description	Provide the priority of each OS Task.
Rationale	The ECU Configuration has to ensure that a server runnable of a sysnchronous C/S call that cannot be invoked as a direct function call is mapped to a task with a higher priority than the calling client runnable
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::Services::OS ::Task::Priority
Required by	rte_sws_2251 rte_sws_4014
Contract phase	No

Requirement ID	**[rte_sws_in_0039]**
Object identifier	**OsEvent**
Object information	Name of the OSEvent
Description	The OSEvent to which the RTEEvent is assigned
Rationale	For the RTEEvents which are implemented with OSEvents the name of the OSEvents shall be defined.
Template meta-model path	–
Required by	rte_sws_2251 rte_sws_4014

Contract phase	No

Requirement ID	**[rte_sws_in_5016]**
Object identifier	**ExclusiveAreaName**
Object information	Name of the exclusive area
Description	The Internal Behavior does provide the list of defined exclusive areas.
Rationale	Define the name of the handle for the exclusive area.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::ExclusiveArea::ExclusiveArea::Identifier
Required by	rte_sws_3739
Contract phase	No

Requirement ID	**[rte_sws_in_5017]**
Object identifier	**InterRunnableVariableName**
Object information	Name of the Interrunnable Variable
Description	The Internal Behavior does provide this list of defined inter runnable variables.
Rationale	Generate the Application Header File for exclusive are access.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::InterRunnableVariable::Identifier
Required by	rte_sws_1120 rte_sws_1123 rte_sws_3550 rte_sws_3553 rte_sws_3560 rte_sws_3565
Contract phase	Yes

B.3 Port and interface

Requirement ID	**[rte_sws_in_0018]**
Object identifier	**PortName**
Object information	Name of the port
Description	Shall be unique within the SWC
Rationale	To identify different port prototype for API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Components ::PortPrototype::Identifier
Required by	rte_sws_1071 rte_sws_1072 rte_sws_1206 rte_sws_1083 rte_sws_1091 rte_sws_1092 rte_sws_1102 rte_sws_1111 rte_sws_3741 rte_sws_3744
Contract phase	Yes

Requirement ID	**[rte_sws_in_0019]**
Object identifier	**RPort/PPort**
Object information	Type of the port
Description	r- or p- port
Rationale	To indicate whether the port is provided or required port for configuration checking
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Components ::PortPrototype
Required by	rte_sws_5508
Contract phase	Yes

Requirement ID	**[rte_sws_in_1352]**
Object identifier	**InterfaceName**
Object information	Name of the interface
Description	Shall be unique within the system
Rationale	To ensure unique names for those things that are related to a particular interface rather than the ports that are characterized by the interface
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface ::Identifier
Required by	rte_sws_2576
Contract phase	Yes

Requirement ID	**[rte_sws_in_0069]**
Object identifier	**InterfaceIsService**
Object information	isService attribute of the PortInterface
Description	Whether port provides or requires the interface is a service port
Rationale	To distinguish the communication with normal SWC and the communication with Basic-SW services.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface::isService
Required by	rte_sws_3807 rte_sws_3808
Contract phase	Yes

Requirement ID	**[rte_sws_in_0020]**
Object identifier	**DataElementName**

Object information	Name of the data element
Description	Shall be unique within the SWC
Rationale	To identify different data element prototype for API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface ::DataElementPrototype::Identifier
Required by	rte_sws_1071 rte_sws_1072 rte_sws_1206 rte_sws_1083 rte_sws_1091 rte_sws_1092 rte_sws_3741 rte_sws_3744
Contract phase	Yes

Requirement ID	**[rte_sws_in_0060]**
Object identifier	**DataElementDatatype**
Object information	Data type of the data element
Description	Contains the information like upper/lower-limit for integer and real type
Rationale	For API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Datatype ::Datatypes
Required by	rte_sws_1071 rte_sws_1072 rte_sws_1206 rte_sws_1083 rte_sws_1091 rte_sws_1092 rte_sws_3741 rte_sws_3744
Contract phase	Yes

Requirement ID	**[rte_sws_in_0045]**
Object identifier	**DataElementIsQueued**
Object information	Specifies whether the data element is queued or not. VFB attribute: INFORMATION_TYPE
Description	Qualifies whether the content of the data element is queued. If it is queued then the data element has ”event” semantics - i.e. data elements are stored in a queue and all data elements are processed in ”first in first out” order. If it is not queued then the ”last is best” semantics applies.
Rationale	For configuration checking and API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface ::DataElementPrototype::isQueued
Required by	rte_sws_1071 rte_sws_1072
Contract phase	Yes

Requirement ID	**[rte_sws_in_0058]**
Object identifier	**OperationName**

Object information	Name of the operation
Description	Shall be unique within the SWC
Rationale	To identify different operation prototype for API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface ::OperationPrototype::Identifier
Required by	rte_sws_1102 rte_sws_1111
Contract phase	Yes

Requirement ID	**[rte_sws_in_0059]**
Object identifier	**ArgumentName**
Object information	Name of the argument of the operation
Description	Shall be unique within the operation
Rationale	For API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface ::OperationPrototype::ArgumentPrototype::Identifier
Required by	rte_sws_1102 rte_sws_1111
Contract phase	Yes

Requirement ID	**[rte_sws_in_0061]**
Object identifier	**ArgumentDirection**
Object information	Direction of the argument of the operation
Description	In/Out/Inout
Rationale	For API generation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::PortInterface ::OperationPrototype::ArgumentPrototype::Direction
Required by	rte_sws_1102 rte_sws_1111
Contract phase	Yes

Requirement ID	**[rte_sws_in_0021]**
Object identifier	**AssemblyConnectorPrototype**
Object information	Connection of communication partners (ports)
Description	Refers to one p-port and one r-port
Rationale	For API implementation

Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Composition::AssemblyConnectorPrototype
Required by	rte_sws_2200
Contract phase	No

Requirement ID	**[rte_sws_in_0055]**
Object identifier	**SInitValue**
Object information	Initial value of a data element prototype (isQueued = false) on the sender side. VFB attribute on sender side: INIT_VALUE.
Description	Refers to a constant value.
Rationale	To prevent caculation based on invalid values
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::DataSenderComSpec::InitValue
Required by	rte_sws_6009 rte_sws_6010
Contract phase	No

Requirement ID	**[rte_sws_in_0062]**
Object identifier	**RInitValue**
Object information	Initial value of a data element prototype (isQueued = false) on the sender side. VFB attribute on receiver side: INIT_VALUE
Description	Refers to a constant value.
Rationale	To prevent caculation based on invalid values
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::DataReceiverComSpec::InitValue
Required by	rte_sws_6010
Contract phase	No

Requirement ID	**[rte_sws_in_0023]**
Object identifier	**ServerRunnable**
Object information	for each operation the connected runnable entity
Description	Refers to the runnable entity which shall be activated when the OperationInvokedEvent is triggered
Rationale	For invocation of the server runnable
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::RTEEvents::OperationInvokedEvent::RunnableEntityRef
Required by	rte_sws_1166
Contract phase	No

Requirement ID	**[rte_sws_in_2574]**
Object identifier	**ApplicationErrorValues**
Object information	Application Error Value definition for each operation
Description	The definition of the Application Error Values used in exchange between SW-Components (with symbolic name and value)
Rationale	Application Errors shall be defined in the Application Header File. For definition of Rte_StatusType.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::VFBErrors::ApplicationError
Required by	rte_sws_2573 rte_sws_2575 rte_sws_2576
Contract phase	Yes

Requirement ID	**[rte_sws_in_5023]**
Object identifier	**CanInvalidate**
Object information	Can the sender invalidate the data element
Description	When specified the sender of a data element can set the value to the invalid value defined in the data semantics.
Rationale	For API generation of data element invalidation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::DataSenderComSpec::canInvalidate
Required by	rte_sws_5024
Contract phase	Yes

Requirement ID	**[rte_sws_in_5031]**
Object identifier	**InvalidValue**
Object information	Invalid value
Description	The value to be used when invalidating a data element.
Rationale	The value to be used for the invalid data indication must be the same for all partners in the communication.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Datatype ::Datatypes::PrimitiveTypeWithSemantics::invalidValue
Required by	rte_sws_3802 rte_sws_5025
Contract phase	Yes

Requirement ID	**[rte_sws_in_3777]**
Object identifier	**AcknowledgementRequest**

Object information	Request an acknowledgement
Description	Requests acknowledgements that data has been sent successfully.
Rationale	The sender of a data element can request an acknowledgement for successful or erroneous transmission using this attribute
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::AcknowledgementRequest
Required by	rte_sws_5504 rte_sws_5505
Contract phase	Yes

Requirement ID	**[rte_sws_in_1361]**
Object identifier	**PortDefinedArgumentType**
Object information	Data type of port-defined argument
Description	The data type that the server runnable entity requires to be passed.
Rationale	To enable correct function prototypes to be emitted
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::PortArgument::type
Required by	rte_sws_1166
Contract phase	Yes

Requirement ID	**[rte_sws_in_1362]**
Object identifier	**PortDefinedArgumentValue**
Object information	Value of port-defined argument
Description	Value to pass for a specific port-defined argument for a specific server SWC (instance).
Rationale	To enable correct values to be passed as the port-defined arguments for invocation of server runnables.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Internal-Behavior ::PortArgument::value
Required by	rte_sws_1360
Contract phase	No

B.4 Communication

Requirement ID	**[rte_sws_in_0067]**
Object identifier	**AliveTimeout**

Object information	The minimum time period for the reception of the data element (isQueued = false). VFB attribute: LIVELIHOOD
Description	When specified the receiver can monitor the time-out and inform a time-out to the software component.
Rationale	For API generation of the time-out notification
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::DataReceiverComSpec::aliveTimeout
Required by	rte_sws_5020 rte_sws_5021 rte_sws_5022
Contract phase	Yes

Requirement ID	**[rte_sws_in_0066]**
Object identifier	**RFiltering**
Object information	the filter mechanism on the receiver side. SWCT attribute: filter
Description	of class DataFilter
Rationale	For API implementation to filter the data element according to certain mechanism on the receiver side
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::Filter::DataFilter
Required by	rte_sws_5503
Contract phase	No

Requirement ID	**[rte_sws_in_0029]**
Object identifier	**QueueLength**
Object information	The length of the queue of the received data element (isQueued = true)
Description	of type Integer
Rationale	For API implementation
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::Communication ::EventReceiverComSpec::QueueLength
Required by	rte_sws_2521
Contract phase	No

Requirement ID	**[rte_sws_in_0063]**
Object identifier	**SignalMappingP**
Object information	mapping of primitive data element to COM signal(s)
Description	refers to data element instance and the COM signal(s) - the COM signal is the interface of COM to RTE.

Rationale	For API implementation by invocation of COM API
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::Communication::DataMappings
Required by	rte_sws_3007 rte_sws_4504 rte_sws_4505
Contract phase	No

Requirement ID	**[rte_sws_in_0064]**
Object identifier	**SignalMappingC**
Object information	mapping of complex data element to COM signal group(s)
Description	refers to data element instance and the COM signal group(s) - the COM signal group is the interface of COM to RTE.
Rationale	For API implementation by invocation of COM API
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::Communication::DataMappings
Required by	rte_sws_3008 rte_sws_4506 rte_sws_4507 rte_sws_4508 rte_sws_2557
Contract phase	No

B.5 Data consistency

Requirement ID	**[rte_sws_in_0033]**
Object identifier	**ExclusiveAreaExecutionOptimization**
Object information	ExclusiveArea data consistency mechanism request from SW-C description
Description	Attribute specifying a special data consistency mechnism to be applied to an ExclusiveArea
Rationale	Influence RTE behavior e.g. runnable execution or RTE code efficiency
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::ExclusiveArea::executionOptimization
Required by	rte_sws_3503 rte_sws_3504
Contract phase	No

Requirement ID	**[rte_sws_in_3592]**
Object identifier	**ExclusiveAreaImplementation**
Object information	ExclusiveArea data consistency mechanism request from ECU configuration
Description	Specification of a special data consistency mechnism to be applied to an ExclusiveArea. Dominates over ExclusiveAreaExecutionOptimization

Rationale	Influence RTE implementation
Template meta-model path	ECU configuration
Required by	rte_sws_3507
Contract phase	No

Requirement ID	**[rte_sws_in_3017]**
Object identifier	**IrvCommAppr**
Object information	Communication approach of InterRunnableVariable
Description	Whether the access to the InterRunnableVariable is explicit or implicit
Rationale	For generation of the API for accessing the InterRunnableVariable.
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::InternalBehavior ::InterRunnableVariable::communicationApproach
Required by	rte_sws_3580
Contract phase	Yes

B.6 Mode management

Requirement ID	**[rte_sws_in_0036]**
Object identifier	**ModeDisablingDependency**
Object information	Dependency between modes and disabling of runnable entities
Description	If a ModeDisablingDependency exists for a runnable entity execution of the runnable entity is disabled for the respective modes.
Rationale	The existence of a ModeDisablingDependency shall prevent the RTE to start a runnable in the corresponding mode
Template meta-model path	M2::AUTOSAR Templates::SWComponentTemplate::ModeDeclaration ::ModeDisablingDependency
Required by	rte_sws_2503
Contract phase	No

B.7 RTE configuration

Requirement ID	**[rte_sws_in_0037]**
Object identifier	**CompatibilityMode**
Object information	RTE generation compatibility mode

Description	RTE generation mode that ensures RTE API compatibility on object code level.
Rationale	The compatibility mode shall be supported by all RTE generators
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::GenerationParameters::RteGenerationMode::COMPATIBILITY_MODE
Required by	rte_sws_1151
Contract phase	Yes

Requirement ID	**[rte_sws_in_0038]**
Object identifier	**VendorMode**
Object information	RTE generation vendor mode
Description	RTE generation mode that provides an vendor-specific optimized RTE implementation
Rationale	An RTE Generator may optionally support vendor mode. RTE generators from different vendors are unlikely to be compatible when run in the vendor mode
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::GenerationParameters::RteGenerationMode::VENDOR_MODE
Required by	rte_sws_1152
Contract phase	Yes

Requirement ID	**[rte_sws_in_5018]**
Object identifier	**RteVfbTrace**
Object information	Enable VFB tracing
Description	RTE Generator will generate code to trace the communication on certain VFB communication
Rationale	The RTE generator shall be able to enable/disable VFB tracing.
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::GenerationParameters::RTE_VFB_TRACE
Required by	rte_sws_1322 rte_sws_1323 rte_sws_1327 rte_sws_1328
Contract phase	No

Requirement ID	**[rte_sws_in_5019]**
Object identifier	**RteVfbTraceFunction**
Object information	VFB tracing hook functions
Description	RTE Generator will generate VFB tracing calls only for the defined communications.

Rationale	To be able to select which communication should be traced.
Template meta-model path	M1::AUTOSAR Descriptions::ECUCParameterDefinition::RTE::GenerationParameters::RTE_VFB_TRACE_FUNCTION
Required by	rte_sws_1324 rte_sws_1325
Contract phase	No

C External Requirements

[rte_sws_ext_2006] The code of a runnable entity shall never modify itself, if reentrancy is required.

[rte_sws_ext_2010] The usage of global variables within runnable entities shall be prohibited, if reentrancy is required.

[rte_sws_ext_2011] The usage of local *static* variables within runnable entities shall be prohibited, if reentrancy is required.

[rte_sws_ext_2054] The RTE-Generator expects only one instance of the ECU Abstraction.

[rte_sws_ext_2559] The RTE configurator shall have access to the schedule table configuration (see also rte_sws_4014)

[rte_sws_ext_2542] Whenever any runnable entity is running, there shall always be exactly one mode active of each ModeDeclarationGroup.

[rte_sws_ext_2507] The mode switch shall be notified to the mode user (and RTE) locally on each ECU.

[rte_sws_ext_2601] The `Rte_IStatus` API shall only be used by a RunnableEntity that either has a DataReadAccess refering to the DataElementPrototype or is triggered by a DataReceiveErrorEvent refering to the DataElementPrototype.

D MISRA C Compliance

In general, all RTE code, whether generated or not, shall conform to the HIS subset of the MISRA C standard rte_sws_1168. This chapter lists all the MISRA C rules of the HIS subset that may be violated by the generated RTE.

The MISRA C standard was defined with having mainly hand-written code in mind. Part of the MISRA C rules only apply to hand-written code, they do not make much sense in the context of automatic code generation. Additonally, there are some rules that are violated because of technical reasons, mainly to reduce RTE overhead.

The rules listed in this chapter are expected to be violated by RTE code. Violations to the rules listed here do not need to be documented as non-compliant to MISRA C in the generated code itself.

MISRA rule	11
Description	Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
Violations	The defined RTE naming convention may result in identifiers with more than 31 characters. The compliance to this rule is under user's control.

MISRA rule	23
Description	All declarations at file scope should be static where possible.
Violations	E.g. for the purpose of monitoring during calibration or debugging it may be necessary to use non-static declarations at file scope.

MISRA rule	42
Description	The comma operator shall not be used, except in the control expression of a *for* loop.
Violations	Function-like macros may have to use the comma operator. Function-like macros are required for efficiency reasons [BSW00330].

MISRA rule	45
Description	Type casting from any type to or from pointers shall not be used.
Violations	For the implementation of exclusive areas (rte_sws_3740, Section 5.4.2.5) casting between pointer types is needed.

MISRA rule	54
Description	A null statement shall only occur on a line by itself, and shall not have any other text on the same line.

Violations	In an optimized RTE, API calls may result in a null statement. Therefore the compliance to this rule cannot be guaranteed.

E Interfaces of COM used by the RTE

The specification of the RTE requires the usage of the following COM API functions and COM callback functions.

COM API function	Context
`Com_SendSignal`	to transmit a data element of primitive type using COM.
`Com_ReceiveSignal`	to retrieve the new value of a data element of primitive type from COM.
`Com_UpdateShadowSignal`	to update a primitive element of a data element of complex type in preparation for sending the complex type using COM.
`Com_SendSignalGroup`	to initiate sending of a data element of complex type using COM.
`Com_ReceiveSignalGroup`	to retrieve the new value of a data element of complex type from COM.
`Com_ReceiveShadowSignal`	to retrieve the new value of a primitive element of a data element of complex type from COM.
`Com_InvalidateSignal`	to invalidate a data element of primitive type using COM.

Table E.1: COM API functions used by the RTE

Callback function	Configuration	Usage
`Rte_COMCbk_<sn>`	`COM_NOTIFICATION_SIGNAL` of `COM_SIGNAL`	Notification of data reception of a data element of primitive type
`Rte_COMCbkInv_<sn>`	`COM_RX_DATA_INVALID_INDICATION_FUNCTION` of `COM_RX_DATA_INVALID` of `COM_SIGNAL`	Notification of reception of an invalidated signal
`Rte_COMCbkTOut_<sn>`	`COM_NOTIFICATION_ERROR` of `COM_SIGNAL`	Notification of a deadline monitoring violation for a data element of primitive type (only present if aliveTimeout is present)
`Rte_COMCbk_<sg>`	`COM_NOTIFICATION_SIGNAL` of `COM_SIGNAL_GROUP`	Notification of data reception of a data element of complex type
`Rte_COMCbkTOut_<sg>`	`COM_NOTIFICATION_ERROR` of `COM_SIGNAL_GROUP`	Notification of a deadline monitoring violation for a data element of complex type (only present if aliveTimeout is present)

Table E.2: COM Callback functions provided by the RTE for signal reception

Callback function	Configuration	Usage
`Rte_COMCbkTAck_<sn>`	`COM_NOTIFICATION_SIGNAL` of `COM_SIGNAL`	Notification of successful transmission of a data element of primitive type (only present if acknowledgement request is specified)
`Rte_COMCbkTErr_<sn>`	`COM_NOTIFICATION_ERROR` of `COM_SIGNAL`	Notification of a transmission error of a data element of primitive type (only present if acknowledgement request is specified)
`Rte_COMCbkTAck_<sg>`	`COM_NOTIFICATION_SIGNAL` of `COM_SIGNAL_GROUP`	Notification of successful transmission of a data element of complex type (only present if acknowledgement request is specified)
`Rte_COMCbkTErr_<sg>`	`COM_NOTIFICATION_ERROR` of `COM_SIGNAL_GROUP`	Notification of a transmission error of a data element of complex type (only present if acknowledgement request is specified)

Table E.3: COM Callback functions provided by the RTE for signal transmission