

**RTA-RTE V6.3.0**  
User Guide



## Copyright

---

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

©Copyright 2017 ETAS GmbH, Stuttgart.

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

**Document:** 10666-UG-001 EN - 10-2017

**Revision:** 68240 [RTA-RTE 6.3.0]

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

## Contents

---

<b>1</b>	<b>About this Manual</b>	<b>7</b>
1.1	Document Conventions . . . . .	9
1.2	Acronyms and Abbreviations . . . . .	9
<b>I</b>	<b>Introduction and Overview</b>	<b>11</b>
<b>2</b>	<b>Introduction to the RTE</b>	<b>12</b>
2.1	What is a Run-Time Environment (RTE)? . . . . .	13
2.2	Software Components . . . . .	14
2.3	Interfaces . . . . .	18
2.4	Software Component Behavior . . . . .	21
2.5	Services . . . . .	25
<b>3</b>	<b>Introduction to RTE Configuration</b>	<b>26</b>
3.1	Understanding XML . . . . .	26
3.2	Understanding AUTOSAR XML . . . . .	27
3.3	ECU Configuration Description . . . . .	33
<b>4</b>	<b>Working with the RTE Generator</b>	<b>36</b>
4.1	Contract Phase . . . . .	37
4.2	RTE Phase . . . . .	39
4.3	Basic Software Phase . . . . .	41
4.4	The Development Process . . . . .	42
4.5	Samples . . . . .	42
<b>II</b>	<b>Developing Software Components</b>	<b>45</b>
<b>5</b>	<b>Type System</b>	<b>46</b>
5.1	Application Data Types . . . . .	46
5.2	Base Types . . . . .	50
5.3	Implementation Data Types . . . . .	51
5.4	Type Mappings . . . . .	60
5.5	Application Types And Numerical Representations . . . . .	61
5.6	Expressing Values in Physical Units or Numerical Representation . . . . .	66
<b>6</b>	<b>Interfaces</b>	<b>69</b>
6.1	Sender-Receiver . . . . .	69
6.2	Nv-Data . . . . .	70
6.3	Mode-Switch . . . . .	71
6.4	Client-Server . . . . .	72
6.5	Calibration . . . . .	73
6.6	Trigger . . . . .	74
<b>7</b>	<b>Software Component Types</b>	<b>75</b>
7.1	Ports . . . . .	76
7.2	Communication Specifications . . . . .	77

<b>8</b>	<b>Internal Behavior</b>	<b>83</b>
8.1	RTE Events . . . . .	84
8.2	Runnable Entities . . . . .	86
8.3	Responding to Periodic Events . . . . .	87
8.4	Sending to a Port . . . . .	88
8.5	Receiving from a Port . . . . .	90
8.6	Responding to a Server Request on a Port . . . . .	94
8.7	Making a Client Request on a Port . . . . .	95
8.8	Direct Trigger of a Runnable Entity . . . . .	100
8.9	Exclusive Areas . . . . .	102
8.10	Inter-Runnable Variables . . . . .	106
8.11	Accessing Modes . . . . .	107
8.12	Per-instance Memory . . . . .	107
8.13	Port Options . . . . .	108
8.14	Supporting Multiple Instantiation . . . . .	109
8.15	Memory Allocation . . . . .	110
<b>9</b>	<b>Modes</b>	<b>111</b>
9.1	Defining Modes . . . . .	111
9.2	Mode Communication . . . . .	111
9.3	Using Modes . . . . .	115
9.4	Understanding Mode Instances . . . . .	118
9.5	Fast Init . . . . .	119
9.6	Synchronizing Modes . . . . .	120
9.7	Non-AUTOSAR Functionality . . . . .	120
<b>10</b>	<b>Implementing Software Components</b>	<b>122</b>
10.1	Basic Concepts . . . . .	122
10.2	Application Source Code . . . . .	124
10.3	Single and Multiple Instances . . . . .	125
10.4	Runnable Entities . . . . .	126
10.5	Sender-Receiver Communication . . . . .	132
10.6	Client-Server Communication . . . . .	135
10.7	Using Inter-Runnable Variables . . . . .	136
10.8	Accessing Parameters . . . . .	136
10.9	Accessing Per Instance Memory . . . . .	137
10.10	Concurrency Control with Exclusive Areas . . . . .	137
10.11	Starting and Stopping the RTE . . . . .	139
<b>11</b>	<b>NVRAM</b>	<b>140</b>
11.1	NV-Block Software Component Types . . . . .	140
11.2	Interaction with Application SWC . . . . .	143
11.3	Interaction with the NVRAM Manager . . . . .	143
<b>III</b>	<b>Developing Basic Software</b>	<b>145</b>
<b>12</b>	<b>Basic Software</b>	<b>146</b>
12.1	Basic Software Modules . . . . .	146

12.2	Interaction with the RTE . . . . .	147
12.3	API . . . . .	152
<b>IV</b>	<b>Composition</b>	<b>154</b>
<b>13</b>	<b>Composing Software Components</b>	<b>155</b>
13.1	Composition Type Definition . . . . .	155
13.2	Component Instances . . . . .	156
13.3	Connector Prototypes . . . . .	156
13.4	FlatMaps . . . . .	167
13.5	Data Conversion . . . . .	173
<b>14</b>	<b>Composing Basic Software</b>	<b>180</b>
14.1	Instantiation . . . . .	180
14.2	BSW Scheduler . . . . .	181
14.3	Connections . . . . .	181
<b>15</b>	<b>Synchronizing BSW and SWC</b>	<b>183</b>
15.1	Configuration . . . . .	183
15.2	Synchronized Mode Groups . . . . .	185
15.3	Synchronized Triggers . . . . .	186
15.4	Synchronized Runnable Entities . . . . .	187
<b>16</b>	<b>Accessing NVRAM</b>	<b>189</b>
16.1	Configuration . . . . .	189
16.2	Access from Application SWCs . . . . .	191
16.3	Access from the NVRAM manager . . . . .	193
<b>V</b>	<b>Deployment</b>	<b>196</b>
<b>17</b>	<b>Defining the ECUs and the Networks</b>	<b>197</b>
17.1	ECU Type Definition . . . . .	198
17.2	ECU Instances . . . . .	199
<b>18</b>	<b>Mapping Software Components to ECUs</b>	<b>200</b>
18.1	Mapping Component Prototypes . . . . .	200
18.2	SWC Implementation Selection . . . . .	201
18.3	Configuring Service Components on an ECU . . . . .	202
18.4	Mapping Runnable Entities to Tasks . . . . .	203
18.5	How Runnables get activated . . . . .	205
<b>19</b>	<b>Inter-ECU Communication</b>	<b>210</b>
19.1	System Communications . . . . .	210
19.2	Inter-ECU Sender-Receiver Communication . . . . .	215
19.3	Inter-ECU Client-Server Communication . . . . .	218
<b>20</b>	<b>Using the OS and COM Configurations</b>	<b>223</b>
20.1	Operating System Configuration . . . . .	224
20.2	Communication Stack Configuration . . . . .	226

<b>21 Debugging Implementations with VFB Tracing</b>	<b>228</b>
21.1 Enabling Tracing . . . . .	228
21.2 Supported Trace Events . . . . .	228
21.3 Configuration . . . . .	231
21.4 Trace Event Header File . . . . .	231
21.5 Implementing Hook Functions . . . . .	231
<b>22 Data Transformation</b>	<b>233</b>
22.1 Extent of support . . . . .	233
22.2 Enabling data transformation in an input configuration . . . . .	236
22.3 Working with data transformation . . . . .	239
<b>VI Advanced Concepts</b>	<b>240</b>
<b>23 RTE Generation</b>	<b>241</b>
23.1 Identifier length . . . . .	241
<b>24 Operating System Considerations</b>	<b>242</b>
24.1 OS Trigger Selection . . . . .	242
24.2 Task Recurrence . . . . .	242
24.3 Schedule Points . . . . .	243
24.4 Basic and Extended Tasks . . . . .	244
24.5 Forced-basic semantics . . . . .	245
<b>25 Understanding Deployment Choices</b>	<b>248</b>
25.1 Intra-task . . . . .	248
25.2 Inter-task . . . . .	250
25.3 Inter-task Client-Server . . . . .	252
25.4 Inter-ECU . . . . .	253
<b>26 Optimization</b>	<b>260</b>
26.1 Buffers for Inter-ECU Reception . . . . .	260
26.2 Direct invocation of the RTE API . . . . .	260
26.3 Sender-Receiver Communication . . . . .	261
26.4 Client-Server Communication . . . . .	262
26.5 Function Elision . . . . .	263
26.6 Init Runnables . . . . .	264
26.7 Data Consistency . . . . .	265
26.8 Tips . . . . .	266
<b>27 RTE Architecture</b>	<b>267</b>
27.1 Component Data Structure . . . . .	267
27.2 Component Instance Handle . . . . .	268
27.3 The RTE API Implementation . . . . .	269
<b>VII Support</b>	<b>272</b>
<b>28 Contact, Support and Problem Reporting</b>	<b>273</b>

# 1 **About this Manual**

---

The *RTA-RTE User Guide* describes how to install the ETAS AUTOSAR Run-Time Environment (RTE) generation tools and how to configure, build and deploy RTE-based software components on electronic control units.

The *RTA-RTE User Guide* is structured along the lines of a top-down development using the RTE, taking users from software component configuration and development with the RTE, through system configuration and software component deployment to final integration of an RTE-based ECU.

- **Part I, *Introduction and Overview***
  - Chapter 2 introduces the fundamental building blocks for applications designed for the AUTOSAR software architecture. The chapter explains how the RTE provides the environment by which AUTOSAR applications run, interact and exchange data at runtime.
  - Chapter 3 outlines the key aspects of the RTA-RTE configuration language.
  - Chapter 4 explains the wider scope of an AUTOSAR development process and how RTA-RTE supports the process.
- **Part II, *Developing Software Components*** presents a guide to the RTE for Software Component Engineers.
  - Chapter 5 explains the AUTOSAR type system and how implementation and application types interact.
  - Chapter 6 explains how to define sender-receiver and client-server interfaces that use the data types and can be used by software components to communicate.
  - Chapter 7 describes how to define the external view of a software component.
  - Chapter 8 shows how to define the threads of control, called runnable entities, that will be executed to trigger or respond to RTE communication.
  - Chapter 9 explains how to use the RTA-RTE provided support for AUTOSAR application modes and how software components can trigger activity when a mode changes.
  - Chapter 10 shows how to write code to interface with the RTE, use the RTE API in application code to communicate with other applications, build in run-time fault tolerance and protect critical data.
  - Chapter 11 shows how to write code to create non-volatile data and to interface with the AUTOSAR NVRAM manager module.
- **Part III, *Developing Basic Software*** presents a guide to the development of Basic Software using RTA-RTE.
  - Chapter 12 explains how to create Basic Software modules for use with the RTA-RTE.

- Part [IV](#), *Composition* presents a guide to the RTE for Software System Integrators.
  - Chapter [13](#), *Composing Software Components* explains how to create instances of software components, configure instance attributes like the number of queued data elements that can be received and compose the instances of software components to create a logical software system.
  - Chapter [14](#), *Composing Basic Software* explains how to create basic software module instances and how to configure their connections.
  - Chapter [15](#), *Synchronizing Basic Software* explains how to synchronize runnables, modes and triggers between basic software module instances and SWC instances.
  - Chapter [16](#), *Accessing NVRAM* explains how to access non-volatile data from application software.
- Part [V](#), *Deployment* presents a guide to the RTE for ECU and Vehicle Integrators.
  - Chapter [17](#) shows how to define the types of ECU, protocol and communication frames that will be present in your system.
  - Chapter [18](#) describes how to map the software architecture created during the composition stage onto the hardware architecture created in the previous chapter and how to map the runnables of a software component into tasks provided by the AUTOSAR OS.
  - Chapter [19](#) describe how to map sender-receiver and client-server communication between software components located on different ECUs onto the underlying AUTOSAR COM communication basic software module.
  - Chapter [20](#) shows how to use the generated OS and COM configuration files in your application.
  - Chapter [21](#) explains how to use the hook calls embedded within a generated RTE to trace RTE events and communication.
  - Chapter [22](#) explains how to serialize and transform communication between software components.
- Part [VI](#), *Advanced Concepts*
  - Chapter [23](#) explains additional configuration options for generation of RTEs using RTA-RTE.
  - Chapter [24](#) explains Basic and Extended tasks, and how the RTE interacts with these OS features.
  - Chapter [25](#) explains how the RTE API calls work when software components are mapped to the same task, the same ECU or different ECUs.
  - Chapter [26](#) explains how the RTA-RTE optimizes the generated RTE and how application mapping can affect the scope for optimization.
  - Chapter [27](#) explains how the RTE application specific headers provide the abstractions that make the RTE work and how they encapsulate optimizations.
- Part [VII](#), *Support*



- Chapter 28 explains how to contact ETAS to obtain technical support for RTA-RTE.

### 1.0.1 Related Documents

---

The *RTA-RTE Reference Manual* provides a complete reference for the RTE including:

- The syntax of the RTE configuration language.
- The features and functionality of the RTE Generation tool.
- The RTE naming conventions.
- A complete RTE API call reference.
- Dependencies on AUTOSAR Basic Software Modules.

### 1.0.2 Who Should Read this Manual?

---

You should read the *RTA-RTE User Guide* if you need to configure, generate or use a run-time environment for an embedded electronic control unit (ECU). Basic familiarity with the concepts of the AUTOSAR software architecture and knowledge of C programming are assumed.

Readers of this document are assumed to be familiar with the *RTA-RTE Getting Started Guide*.

First time users of the RTE should read this document for a detailed description of what features and facilities the RTE provides to software components.

## 1.1 Document Conventions

---



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



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

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

### 1.2 Acronyms and Abbreviations

---

AUTOSAR	AUTomotive Open System ARchitecture - a standardized software architecture targeted at automotive applications aimed at fostering the reuse of application software over multiple vehicle platforms. See <a href="http://www.autosar.org">http://www.autosar.org</a> .
BSW	AUTOSAR Basic Software, see Chapter 12.
C/S	Client-server communication, see Section 6.4.
ECUC	AUTOSAR ECU Configuration
IOC	Inter-OsApplication Communication
RTA-OS	An AUTOSAR SC1-SC4 and OSEK 2.2.3 compatible operating system from ETAS GmbH.
RTA-OSEK	An AUTOSAR SC1 and OSEK 2.2.3 compatible operating system from ETAS GmbH.
RTE	AUTOSAR Run-Time Environment. See “Introduction to the RTE” (Chapter 2) for further details.
RTA-RTE	The ETAS AUTOSAR RTE Generator Product. This includes the AUTOSAR RTE Generator Tool responsible for reading the AUTOSAR XML configuration and generating the RTE and associated C header files. RTA-RTE distributions also include the RTE library, all user documentation and an example application.
SchM	BSW Scheduler (a BSW module defined by AUTOSAR).
S/R	Sender-receiver communication, see Section 6.1.
SWC	An AUTOSAR Software component, see Chapter 7.
VFB	Virtual Function Bus. See “Introduction to the RTE” (Chapter 2) for further details.
XML	eXtensible Markup Language used to describe AUTOSAR configurations. RTA-RTE processes the input configuration to generate the required RTE

## **Part I**

# **Introduction and Overview**

## 2 Introduction to the RTE

The RTE forms part of a layered software architecture based on the AUTOSAR layered software architecture shown in Figure 2.1.

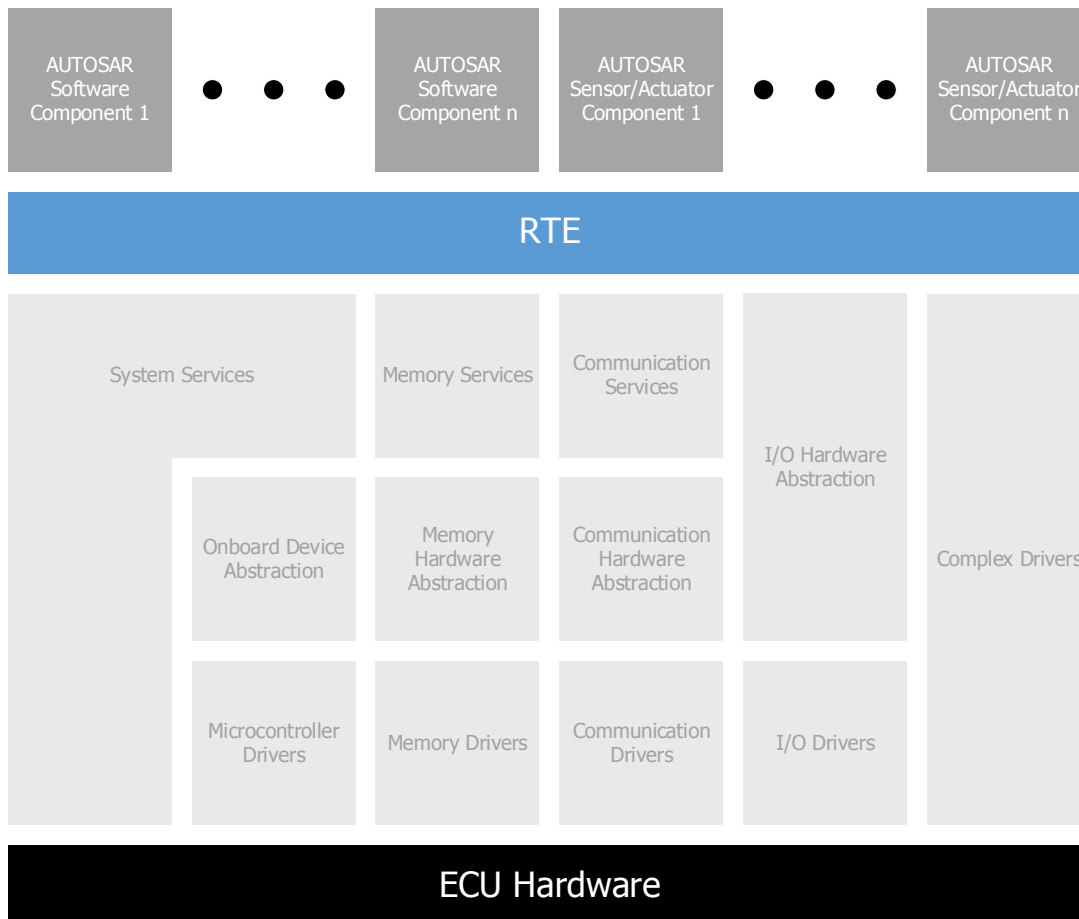


Figure 2.1: Overview of the AUTOSAR software architecture

Application software is the name given in AUTOSAR to vehicle functions, e.g. braking system, window wiper, seat control etc. Each application is decomposed into one or more software components that provide a cohesive part of the application.

Software components are the atomic unit of application distribution in AUTOSAR. When an AUTOSAR system is built, all the constituent parts of a component (the code, data etc.) must live on the same ECU.

AUTOSAR defines two principle types of application software components:

- AUTOSAR software component—generic application-level software components that are designed to be both CPU and location neutral. An AUTOSAR application software component can be mapped to any available ECU during system configuration, subject to constraints imposed by the system designer. The AUTOSAR soft-

ware component is therefore the atomic unit of distribution in an AUTOSAR system.

- AUTOSAR Sensor/Actuator component—software components that are closely coupled to some aspect of ECU hardware and are therefore typically not re-locatable (onto a different ECU) due to their high degree of dependence on features of a specific ECU.

In an AUTOSAR system, software component interaction is designed against a Virtual Function Bus (VFB). The VFB provides a design abstraction that allows interacting software components to be specified and built without detailed knowledge about which ECU they will be allocated at integration time, how ECUs in the vehicle network communicate, the vehicle network topology etc.

In the VFB model, software components interact through ports which are typed by interfaces. The interface controls what can be communicated and the semantics of communication. The port provides the software component access to the interface. The combination of port and interface is known as an AUTOSAR interface. Software components are characterized by component type attributes that define how the components interact with an interface over a port. Attributes might allow you to specify things like:

- A software component waits for data to be provided on a port.
- A software component polls a port for new data periodically.
- Acknowledgment of receipt of a sent message is required.

The VFB provides sufficient information about the logical interaction of software components to allow software systems to be integrated and tested before system integrators have decided on the allocation of software components to ECUs. This means that the entire functional behavior of a system can be prototyped before the electrical architecture of a vehicle, in terms of ECUs on networks, is known.

The next stage in development is system integration where software components are allocated to ECUs and the abstract notions of communication embedded in their VFB design are mapped to signals sent over the vehicle network. Software components must also be bound onto the computing platform, for example, integrated into tasks for scheduling by an operating system.

## 2.1 What is a Run-Time Environment (RTE)?

The VFB provides the abstraction that allows components to be reusable. The RTE encapsulates the mechanisms required to make the VFB abstraction work at runtime. The RTE is therefore, in the simplest case, an implementation of the VFB. However, the RTE must provide the necessary interfacing and infrastructure to allow software components to:

1. be implemented without reference to an ECU (the VFB model); and

2. be integrated with the ECU and the wider vehicle network once this is known (the Systems Integration Model) without changing the application software itself.

More specifically, the RTE must:

- Provide a communication infrastructure for software components.  
This includes both communication between software components on the same ECU (intra-ECU ) and communication between software components located on different ECUs (inter-ECU).
- Arrange for real-time scheduling of software components.  
This typically means mapping software components onto tasks provided by an operating system according to timeliness constraints specified at design time.

Application software components have no direct access to the basic software below the abstraction implemented by the RTE. This means that components cannot, for example, directly access operating system or communication services. So, the RTE must present an abstraction over such services that remains consistent irrespective of where the software components are located. All interaction between software components therefore happens through standardized RTE interface calls.

The RTE also binds the software component architecture onto one or more ECUs. To make the RTE efficient, this binding is done statically at build time. The standardized RTE interfaces are automatically implemented by an RTE generation tool that makes sure that interface behaves in the correct way for the specified component interaction and the specified component allocation.

For example, if two software components reside on the same ECU they can use internal ECU communication, but if one is moved to a different ECU, communication now needs to occur across the vehicle network.

The generated RTE therefore encapsulates the variability in the software that arises from different mappings of components to ECUs by:

- Presenting a consistent interface to the software components so they can be reused—they can be designed and written once but used multiple times.
- Binding that interface onto the underlying AUTOSAR basic software implement the VFB design abstraction.

## 2.2 Software Components

---

Within the two broad types outlined above, AUTOSAR defines multiple sub-types of software component:

- Application software components.

- Sensor/actuator software components.
- Service components.
- Complex device driver components.
- ECU abstraction components.

From the perspective of the RTE, however, the multiple types are largely identical and hence, for reasons of brevity, the term “software component” is used to refer to all types.

Software components interact through ports. There are two classes of Port:

- **PPorts** are used by a software component to provide data or services to other software components. Provided ports implement senders and servers.
- **RPorts** are used by a software component to require data or services from other software components. Required ports implement receivers and clients.

Figure 2.2 shows a software component with PPorts and RPorts.

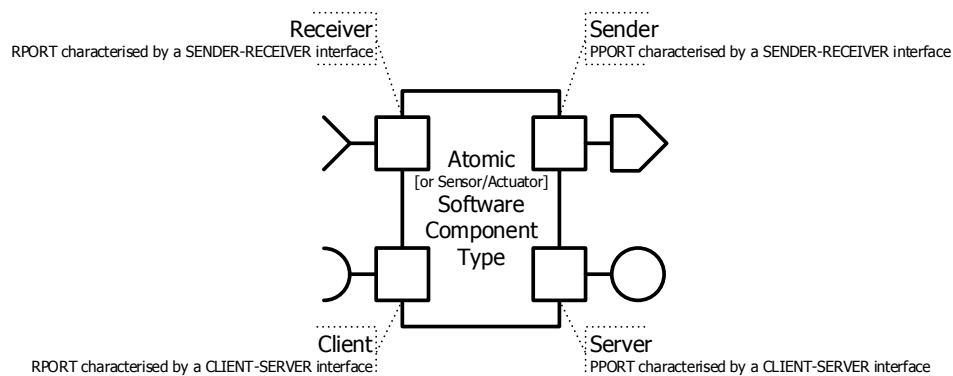


Figure 2.2: Software Component with PPorts and RPorts

### 2.2.1 Types

All configuration items in AUTOSAR are typed. Before any particular object can be created it is first necessary to define its type.

Each type defines the fixed characteristics of the object. In a simple case you might define a numeric type called `MyNumberType` that has a range 0 to 10. This then allows you to create a `MyNumberType` which can take the values in the underlying type.

The same concept applies to AUTOSAR software-components. Before you can create a software component you must first define its component type that identifies the fixed characteristics of a component, e.g. the port names and how ports are typed by interfaces, how the component behaves etc. The component type is named and the name

must be unique within the system. Component type names are typically assigned when you create components.

### 2.2.2 Component Types and Instances

To allow the same component to be used it needs to be instantiated at configuration time. Furthermore, it is possible to configure a software component so that multiple instantiation is possible. For example, you might write a software component to calculate a rolling average that is used many times by different applications but with different data set sizes. Each application could use its own instance of the same core algorithm.

Figure 2.3 shows the relationship between a component type and its instance. Here the type is ETASoffice. Multiple instances of the ETASoffice can be defined and each one is an instance.

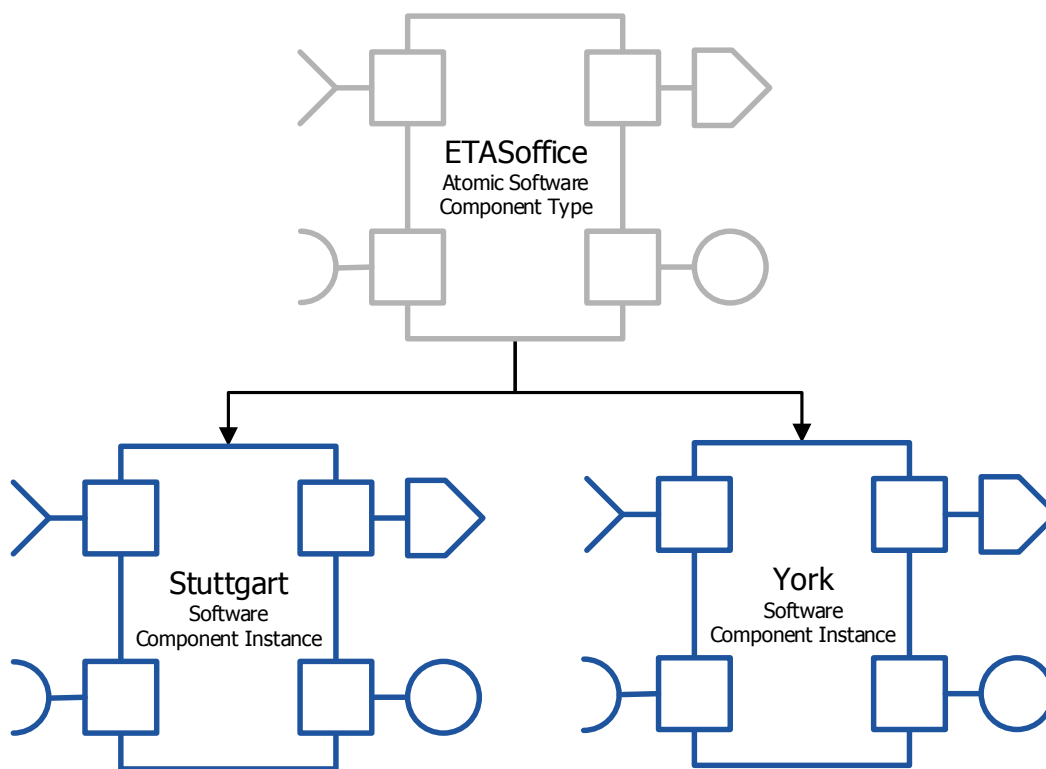


Figure 2.3: Component Types and Instances

The distinction between type and an instance is analogous to types and variables in convention programming languages. You would define an application wide unique type name (the component type) and you would declare one or more uniquely named variables of the type (one or more component prototypes). For example, where we would have `int x`, in a programming language we have component type instance in the RTE.

When there are multiple instances of a software component all instances of a component type share the same code, but each instance may have private state that is not



shared between different instances of the component. Consequently, each instance you declare is given a “copy” of the private state to which it has exclusive access through an instance handle that is generated by the RTE. Each time you call an RTE service you must pass in the instance handle so that the component type code knows which component instance data it should use as shown in Figure 2.4. The idea of common code and private state that is passed to functions in the RTE is analogous to the concept of abstract data types provided by some programming languages.

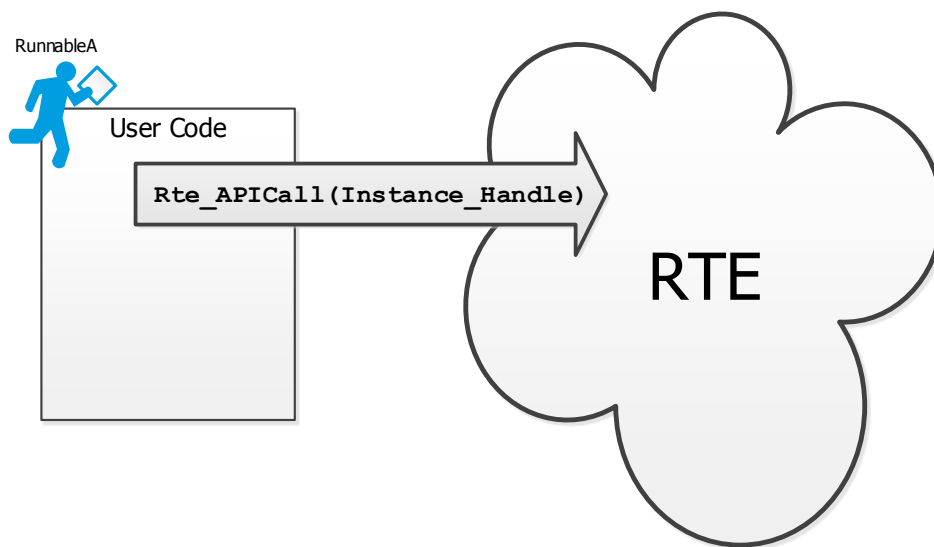


Figure 2.4: Passing a Component Instance handle to the RTE

All software component instances mapped to a given ECU share the same code, but the presence of multiple runnable entities means that a software component is multi-threaded and there exists the possibility for multiple runnable entities to be simultaneously executing within the same component.

For example, two software component prototypes mapped to different ECUs may both attempt to invoke operations in the same server in a third software component instance. This implies some form of concurrency control.

The RTE controls potentially conflicting access to shared resources through two mechanisms:

- Serialization
- Exclusive areas

Serialization is implemented automatically within RTA-RTE generated code and provides a coarse grained mechanism that prevents concurrent execution of certain runnable entities, such as servers.

Exclusive areas provide a mechanism that allows users to have fine control over concurrency. All exclusive areas must be defined at configuration time and are either:

- Explicitly entered and exited in user code using an RTE API; or,
- Implicitly entered and exited in RTA-RTE generated code.

The scope of exclusive areas is the component instance and they cannot therefore be used to control concurrency between different instances of the same component type or between instances of different component types. Consequently, exclusive areas cannot be used to prevent concurrent access to a non-AUTOSAR resource on an ECU. If such a mechanism is required, then a complex device driver should be implemented.

## 2.3 Interfaces

---

When an application consists of multiple software components, it may be necessary for the software components to communicate, either to exchange data or to trigger some function. Software components from different applications may also need to communicate, for example a climate control system will need to communicate with an engine management system to set the engine idle speed sufficiently high that the engine does not stall when the air conditioning compressor is switched on.

Communication between AUTOSAR application software components is designed in terms of ports and interfaces. The following interface types are available:

1. Sender-receiver (signal passing)
2. Client-Server (function invocation)
3. Calibration
4. Trigger

These communication models are known as interfaces in AUTOSAR. Each port on a software component type must define the type of interface it provides or requires.

Software-component types can define individual attributes, for example the maximum number of data elements buffered on a receiver interface, so the same interface may have different behavior for each port in which it is used.

When a system is composed from component instances, the ports of component instances are connected with an assembly connector. The assembly connector must connect a sender to a receiver and a client to a server. Each assembly connector can also specify attributes for the behavior between two components.

### 2.3.1 Sender-Receiver

---

Sender-receiver communication involves the transmission and reception of signals consisting of atomic data elements sent by one component and received by one or more components.

An AUTOSAR configuration can define multiple sender-receiver interfaces and a software component can reference a different interface from each port.

Each sender-receiver interface can contain multiple data elements each of which can be sent and received independently. Data items within the interface can be simple types like integer, float, natural etc. or more complex types like records, arrays etc. to support simultaneous transmission where atomic transmission of related data values is required.

Figure 2.5 shows the sender side of a sender-receiver interface that includes three simple data elements.

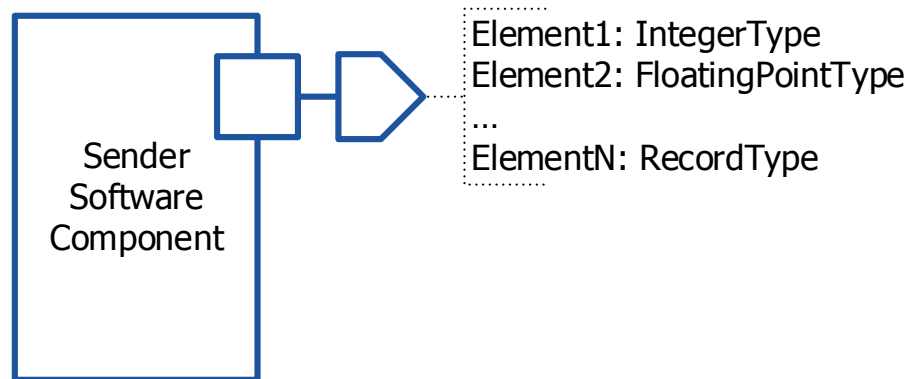


Figure 2.5: Sender-Receiver Interface Data Elements (Sender Side)

Sender-receiver communication is one way—any reply sent by the receiver must be sent as a separate sender-receiver communication.

An RPort of a component that requires an AUTOSAR sender-receiver interface can read the data elements described in the interface and a PPort that provides the interface can write the data elements.

Sender-receiver communication can be “1:n” (single sender, multiple receivers) and “n:1” (multiple senders, single-receiver) communication shown in Figure 2.6 and Figure 2.7 respectively.

Sender-receiver “1:n” communication provides a multicast mechanism in the RTE. When the sender transmits a signal, it is available at each receiver.

When signals are sent by multiple senders to a single receiver (“N:1” communication), as illustrated in Figure 2.7, there is no synchronization imposed on the senders.

#### Application Modes

An AUTOSAR system can be configured to operate in one or more application modes and can configure runnable entities that are activated on either entry or exit from a mode.

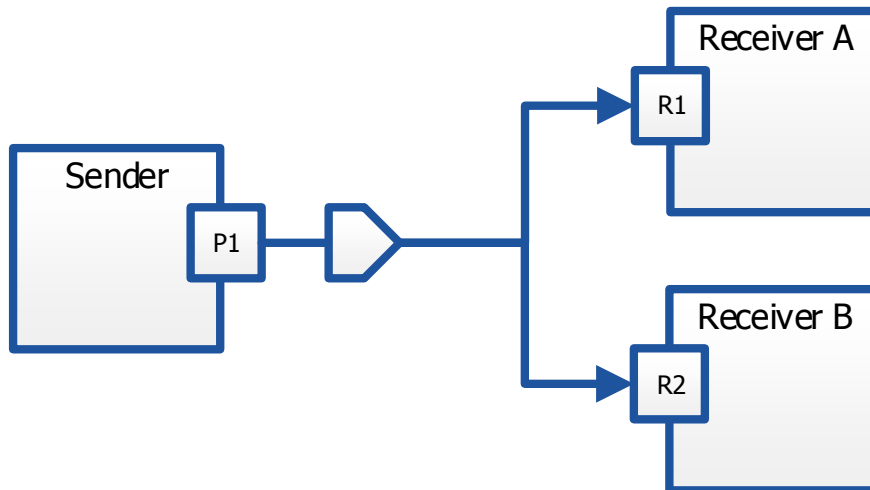


Figure 2.6: Sender-Receiver 1:N Communication

Modes are communicated using sender-receiver communication and are thus defined within a SENDER-RECEIVER interface.

### 2.3.2 Client-Server

Client-server communication involves a component invoking a defined “server” function in another component which may or may not return a reply.

A component type can define multiple ports categorized by client-server interfaces.

Each client-server interface can contain multiple operations each of which can be invoked separately. A port of a component that requires an AUTOSAR client-server interface to the component can independently invoke any of the operations defined in the interface, by making a client-server call to a port providing the service. A port that provides the client-service interface provides implementations of the operations.

Figure 2.8 shows the server side of a client-server interface that serves trigonometric functions to clients.

RTA-RTE supports multiple clients invoking the same server (i.e. “N:1” communication where  $N > 0$ ) illustrated in Figure 2.9.

Note that it is not possible for a client to invoke multiple servers with a single request (i.e. “1:N” communication). A client can, of course, call more than one server by making more than one request.

### 2.3.3 Calibration

Calibration interfaces are used for communication with Calibration components.

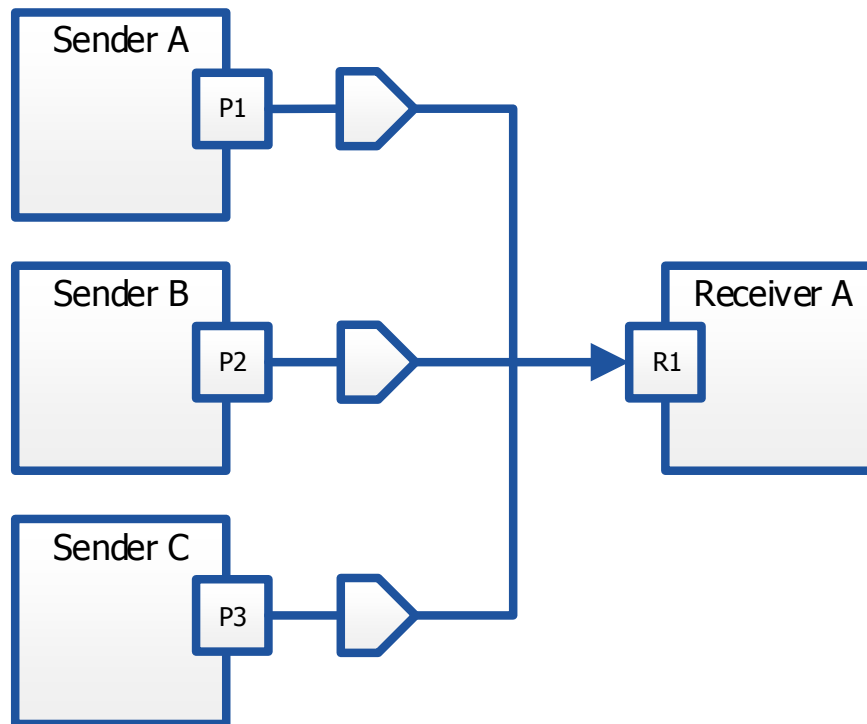


Figure 2.7: Sender-Receiver N:1 Communication

Each calibration interface can contain multiple calibration parameters. A port of a software-component that requires an AUTOSAR calibration interface to the component can independently access any of the parameters defined in the interface, by using the RTE API generated for that calibration parameter on the required port. Calibration components provide the calibration interface and thus provide implementations of the calibration parameters.

## 2.4 Software Component Behavior

The software component description we have looked at so far only describes the external interfaces—i.e. those required for component integration. It does not define how the code that implements the component interacts with the ports. The description of this is called the software component's internal behavior and defines what triggering events the component responds to and how threads of execution (called Runnable Entities) execute when events occur.

A runnable entity, or simply runnable, is a piece of code in a software component that is triggered by the RTE at runtime<sup>1</sup>.

Figure 2.10 shows the internal behaviour for a software component. Note that a

<sup>1</sup>A runnable is similar in concept to a task in an OSEK Operating System.

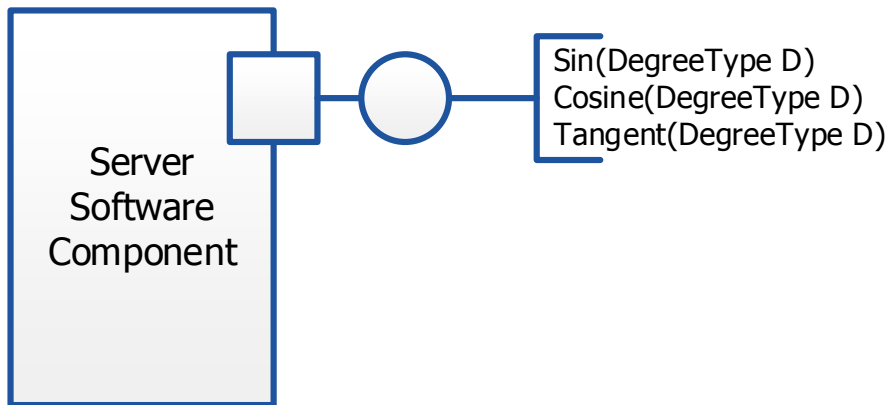


Figure 2.8: Client-Server Interface Operation (Server Side)

runnable may do multiple things, for example receive from one interface and send on another, or act as a server but receive from an interface in order to complete server processing.

A software component comprises one or more runnable entities and each runnable entity must have a unique handle so that the RTE can access it at runtime

Runnable entities are triggered by three classes of events:

- **Timing events** represent some periodic scheduling event, e.g. a periodic timer tick. The runnable entity provides the entry point for component execution for things that need to be executed on a regular basis. For example, a component may want to poll its port for incoming data and would use a runnable triggered by a timing event to do this.
- **Communication events** provide a link between the external (port) view of the software component and the runnable entities internal to the software component. For example, when a software component needs to respond to a client request on a server port it will use a runnable to perform the server operation. Similarly, if a software component needs to perform processing every time a new data item is received then it will use a runnable that responds to a data received communication event.
- **Mode switch events** are the third class of events that trigger runnable entities. When a mode switch is triggered at runtime, the RTE can invoke a runnable entity in the software component to perform the function associated with the mode switch.

#### Runnable Categories

As well as knowing how a runnable entity is triggered, we must also know how the runnable entity behaves once triggered. This is captured by the runnable entity cate-

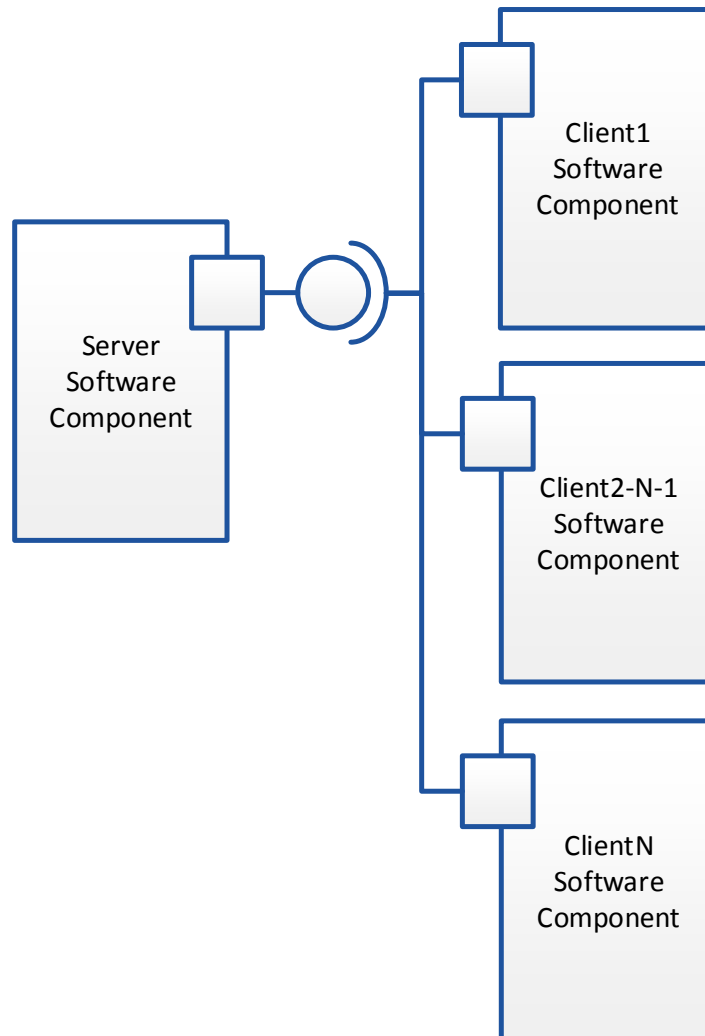


Figure 2.9: Client-Server N:1 Communication

gory. RTA-RTE supports two categories of runnable entity:

- **Category 1** runnable entities must be non-blocking (start, execute, terminate) and have a finite execution time. A Category 1 runnable entity has limited access to RTE facilities – for example, it is not permitted to make RTE calls that can block such as synchronous client-server calls to components located on other ECUs. A Category 1 runnable entity is similar in concept to a basic task in the OSEK Operating System. The Category 1 runnable entities are further split into categories 1a and 1b. Only category 1b runnable entities can use explicit communication.
- **Category 2** entities are permitted to block (start, execute, wait for interaction and (optionally) terminate) and need not have a finite execution time. A Category 2

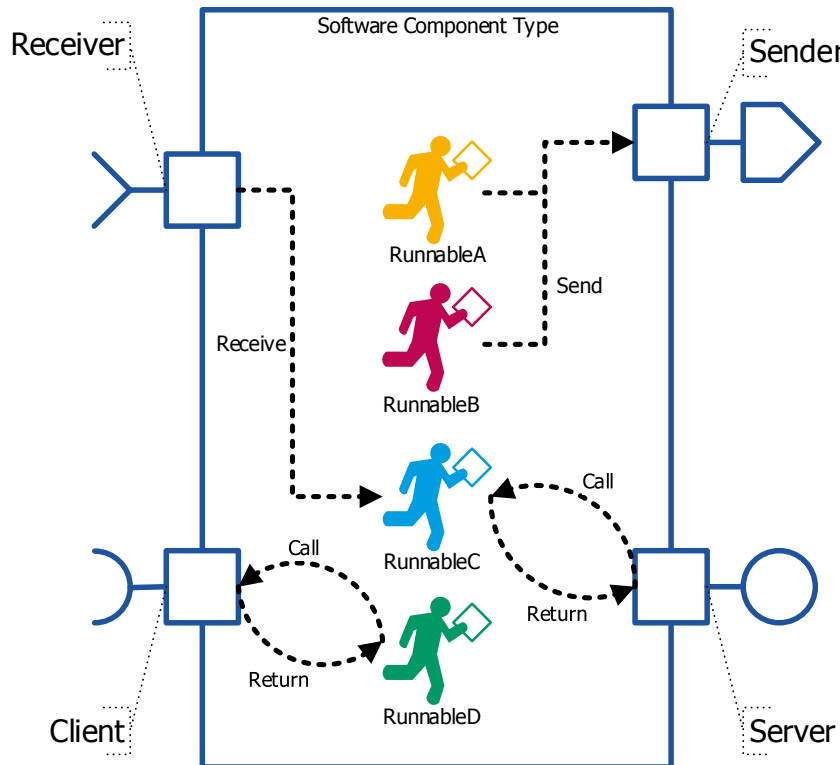


Figure 2.10: Software Component Internal Behaviour

runnable entity is similar in concept to an extended task in the OSEK Operating System.

### Implementing Runnable Entities

You must implement the entry points (similar to task bodies in an OSEK operating system) of all the runnable entities defined for your component. You must also map runnable entities onto operating system tasks at ECU deployment time.

When a runnable entity is part of a software component that supports multiple instantiation, the runnable must know the component instance on which it operates at runtime. The RTE does this by passing the component instance handle to the runnable entity at runtime when it is activated by the RTE.

The instance handle that is passed to the runnable entity by the RTE is passed back to the RTE through each API call made from the runnable entities thread of execution. This mechanism is shown in Figure 2.11 where Instance\_handle represents the same handle in both communications between user code and the RTE.



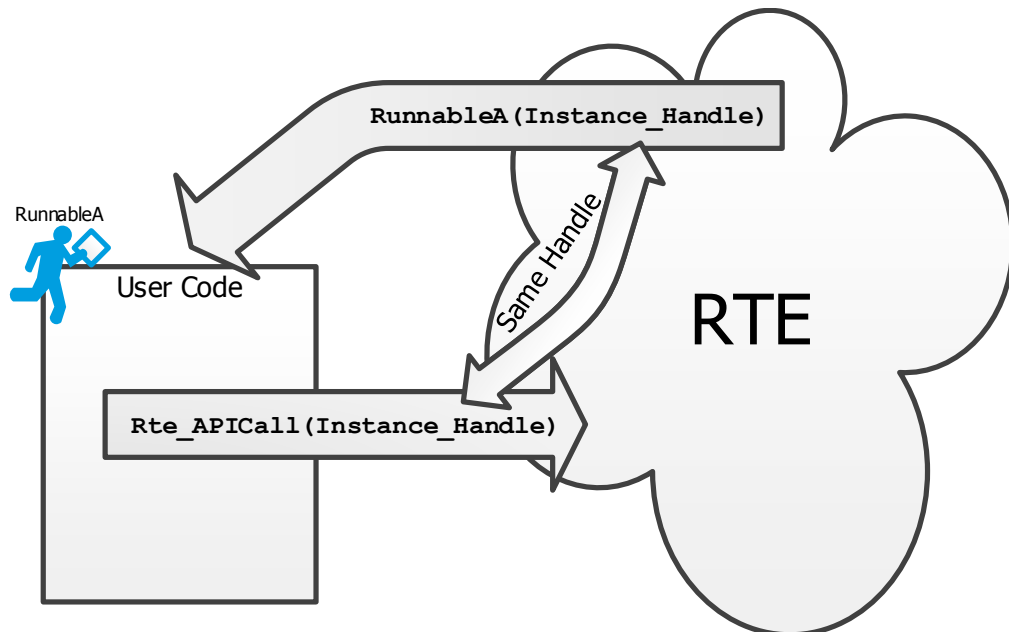


Figure 2.11: Instance handle flow

## 2.5 Services

Services provide standardized functionality to AUTOSAR applications through an AUTOSAR interface. A service can thus be seen as a hybrid between application software components and the basic software modules that provide access to low-level ECU-wide services.

An AUTOSAR application component makes use of services by connecting to them in the same way that it does to other software-components within the application, i.e. using ports characterized by interfaces. A service component may present provider or requirer ports (or both), and may participate in either sender-receiver or client-server communications. The interfaces that characterize the ports of a service component are marked as “Service” Interfaces to indicate their standardized status.

In common with application components, service components have internal behaviors and are triggered by RTE events. In fact, the RTE treats service components in much the same way as application software components with the following important exceptions:

- There is at most only a single instance of each service component type on one ECU,
- Service components are not mapped to ECUs via the SYSTEM mapping.

## 3 Introduction to RTE Configuration

---

The RTE generator generates the RTE from one or more XML files that define the entire required behaviour. The structure of the XML files is standardized by AUTOSAR.

This chapter gives a short overview of XML in general and then a description of the key concepts in AUTOSAR XML that you will need to understand to write RTE configuration files.

### 3.1 Understanding XML

---

XML is a mark-up language for documents that uses a series of tags to convey structured information.

XML does not define either the name of the tags or the semantics of the data enclosed by the tags. Instead, the names and structure of the tags is defined in by an XML schema.

Each XML document you write needs to specify the schema that is used for processing the remaining file content.

With schema definitions it is possible to define a namespace for the tags which allows you to create a single XML file that references configuration from multiple schemas.

There are basically three types of mark-up that you need to understand to work with AUTOSAR configuration:

- Elements
- Attributes
- Comments

#### 3.1.1 Elements

---

An element is delimited using angle brackets and is the most common construct you will see. For elements that contain content, including sub-elements, the general structure is:

```
<ELEMENT-TAG>  
...  
</ELEMENT-TAG>
```

An element can be empty which means that it has no content, this is represented using the following shorthand:

```
<EMPTY-ELEMENT-TAG/>
```

#### 3.1.2 Attributes

---

An attribute is a name-value pair that occurs inside start-tags after the element name. For example,

```
<ELEMENT-TAG SOME-ATTRIBUTE="some value">
```

All attribute values must be quoted.

### 3.1.3 Comments

Sometimes you will need to add comments to your XML document. Comments begin with `<!--` and end with `-->` and can contain any data except the literal string `"—"`. You can place comments between mark-up anywhere in your document.

```
<!-- This is a comment in XML -->
```

Comments are not part of the textual content of an XML document and are ignored by the XML processor.

## 3.2 Understanding AUTOSAR XML

AUTOSAR defines the tags and their semantics using an XML schema definition.

### 3.2.1 Namespace declaration

AUTOSAR XML descriptions (XML files describing all or part of an AUTOSAR configuration) must declare the AUTOSAR namespace as the default namespace. A namespace is declared using an `xmlns` attribute on the root element of an XML file.

The AUTOSAR namespace for all AUTOSAR releases in the 4.x series is <http://autosar.org/schema/r4.0>.



*RTA-RTE will reject a configuration if any of its input files do not declare the AUTOSAR 4.x namespace.*

### 3.2.2 Schema Validation

RTA-RTE validates each XML input file that declares the AUTOSAR 4.x namespace against an AUTOSAR XML schema.

Each 4.x schema from AUTOSAR is a superset of all previous 4.x schemas, allowing all instance documents to be validated against the latest schema. Due to the relaxation of some constraints, configurations that would have been rejected when validated against earlier AUTOSAR schema versions may be accepted when validated against a later version.

The schema version used by RTA-RTE for input validation is AUTOSAR 4.3.0.



*RTA-RTE will reject a configuration if any of its input files fail to validate against the supplied AUTOSAR 4.3.0 schema.*

Validating against a newer AUTOSAR revision than supported

If you are using a schema newer than RTA-RTE's current supported schema, input files may not validate against the supported AUTOSAR 4.3.0 schema. If you do not require

the use of RTE-relevant features from the new schema version, you can override the schema RTA-RTE uses for validation, allowing these files to be accepted.



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

To override the validation schema, you must edit the RTEGen.ini file to supply the location of the new schema file. For more details see *Overriding the validation schema* in the *RTA-RTE Reference Manual*.

### 3.2.3 Object Names

All objects are named using the <SHORT-NAME> tag:

```
<SHORT-NAME>ThisIsMyNamedObject</SHORT-NAME>
```

The names allocated in the configuration for software component prototypes, ports, runnable entities, interfaces etc. are used by the RTE generator to generate object handles and customized API calls for use at runtime. This means that names you give objects in the RTE configuration must be valid C identifiers.



*The AUTOSAR schema restricts the maximum permitted length of short names to 127 characters. A short name may only begin with an alphabetic character (a-z and A-Z).*

### 3.2.4 Packages

The <AUTOSAR> element is a container for exactly one top level packages <AR-PACKAGES> element. The top-level packages element represents the root of an XML object tree from which all objects in all configuration files can be accessed.

The top-level packages itself then contains one or more packages each defined with the <AR-PACKAGE> element. Each <AR-PACKAGE> defines a group of AUTOSAR elements.

A package definition is named using the <SHORT-NAME> element. Each package should have a unique name so that the elements contained within the package can be referenced by other packages, for example:

```
<AUTOSAR>
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>MyPackage</SHORT-NAME>
      <DESC>This is one of my packages</DESC>
    </AR-PACKAGE>
    ...
    <AR-PACKAGE>
      <SHORT-NAME>MyOtherPackage</SHORT-NAME>
      <DESC>This is another</DESC>
    </AR-PACKAGE>
```

```
</AR-PACKAGES>  
</AUTOSAR>
```

The <AR-PACKAGE> element is used to define the package name as well as acting as a container for the <ELEMENTS> descriptions. <ELEMENTS> is a container for the components of an AUTOSAR configuration including:

- The definition of atomic software component types
- The description of component internal behavior
- The composition descriptions including software component prototype instantiation
- The definition of ECU types
- The system description including ECU instantiation

The <SUB-PACKAGES> element permits a hierarchical arrangement of packages to be formed:

```
<AUTOSAR>  
  <AR-PACKAGES>  
    <AR-PACKAGE>  
      <SHORT-NAME>MyPackage</SHORT-NAME>  
      <DESC>This is one of my packages</DESC>  
      <SUB-PACKAGES>  
        <AR-PACKAGE>  
          <SHORT-NAME>SWCs</SHORT-NAME>  
          ...  
        </AR-PACKAGE>  
        <AR-PACKAGE>  
          <SHORT-NAME>Interfaces</SHORT-NAME>  
          ...  
      </AR-PACKAGE>  
    ...  
  </AR-PACKAGES>  
</AUTOSAR>
```

AUTOSAR does not permit an arbitrary split of XML definitions between files. Instead, files can only be split at the top-level packages level. When you need to work with multiple XML files you must therefore split them at the top-level packages level.

In the previous example, we might have decided to split this file into two different files, in which case in File 1 we would have:

```
<?xml version="1.0" encoding="UTF-8"?>  
<AUTOSAR>  
  <AR-PACKAGES>  
    <AR-PACKAGE>  
      <SHORT-NAME>MyPackage</SHORT-NAME>  
      <DESC>This is one of my packages</DESC>  
      ...  
    </AR-PACKAGE>  
  </AR-PACKAGES>  
</AUTOSAR>
```

```
</AR-PACKAGE>  
</AR-PACKAGES>  
</AUTOSAR>
```

In File 2 we would have the second AR-PACKAGE:

```
<?xml version="1.0" encoding="UTF-8"?>  
<AUTOSAR>  
  <AR-PACKAGES>  
    <AR-PACKAGE>  
      <SHORT-NAME>MyOtherPackage</SHORT-NAME>  
      <DESC>This is another</DESC>  
      ...  
    </AR-PACKAGE>  
  </AR-PACKAGES>  
</AUTOSAR>
```

With the exception of Module and ECU configuration elements within the ECUC Description and some “splittable” elements, RTA-RTE will reject configurations that have the elements with the same name at the same level in the package hierarchy.

### 3.2.5 Referencing Objects

The elements in AUTOSAR XML define a hierarchical structure that is rooted at the top-level packages element. Each <SHORT-NAME> named object within this structure can be referenced.

The AUTOSAR XML makes extensive use of this referencing concept – objects are created using a set of elements and then referenced from other objects. For example, when defining the ports of a software component you will need to reference interface descriptions.

All references are indicated by XML elements with a tag ending \*REF.

AUTOSAR XML allows references to be made using either absolute or relative paths. An absolute path references an object from top of the XML tree and must start with a forward-slash. For example:

```
<*REF>/MySystem/MyComponent/MyObject</*REF>
```

The following example shows the use of absolute referencing:

```
<AR-PACKAGE>  
  <SHORT-NAME>PackageA</SHORT-NAME>  
  <ELEMENTS>  
    <ELEMENT>  
      <SHORT-NAME>X</SHORT-NAME>  
      ...  
    </ELEMENT>  
    ...  
  </ELEMENTS>  
</AR-PACKAGE>
```

```

<AR-PACKAGE>
  <SHORT-NAME>PackageB</SHORT-NAME>
  <ELEMENTS>
    <ANOTHER-ELEMENT>
      <SHORT-NAME>X</SHORT-NAME>
      <!-- Absolute reference to X in PackageA -->
      <ELEMENT-REF>/PackageA/X</ELEMENT-REF>
      ...
    </ANOTHER-ELEMENT>
    <YET-ANOTHER-ELEMENT>
      <SHORT-NAME>Y</SHORT-NAME>
      <!-- Absolute reference to X in PackageB -->
      <ANOTHER-ELEMENT-REF>/PackageB/X</ANOTHER-ELEMENT-REF>
      ...
    </YET-ANOTHER-ELEMENT>
  </ELEMENTS>
</AR-PACKAGE>

```

### 3.2.6 Relative References



*A relative reference can be distinguished from an absolute reference since the latter always start with the '/' character. See the RTA-RTE Reference Manual for more details on the use of absolute references within RTA-RTE.*

The relative reference mechanism defines optional *reference bases* for each AUTOSAR package. Each reference base defines the prefix to be used for relative references that are associated with the reference base. For example, assume a package defines the following reference base:

```

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

```

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

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

Is equivalent to the absolute reference:

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

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

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

### 3.2.7 Instance References

---

There is a special type of reference in AUTOSAR called an instance reference that is used whenever you need to refer to a particular instance of an element. Instance references are indicated with tags of the form `<*-IREF>`.

An instance reference itself comprises multiple reference elements each of which defines one part of the instance reference. The composite references depend on context; for example they might include a reference to a component prototype, a port prototype and a data element prototype.

When resolving an instance reference, RTA-RTE indirects through each context reference in turn to complete the reference. For example, the following instance reference:

```
<DATA-IREF>
  <R-PORT-PROTOTYPE-REF>
    /system/SWC/InputPort
  </R-PORT-PROTOTYPE-REF>
  <DATA-ELEMENT-PROTOTYPE-REF>
    /interfaces/SR/InputValue
  </DATA-ELEMENT-PROTOTYPE-REF>
</DATA-IREF>
```

Is interpreted as “find the context InputPort in software component SWC in AR-PACKAGE system then look for a <DATA-ELEMENT-PROTOTYPE> called InputValue in the interface categorizing the port”. An instance reference is important to the correct operation of RTA-RTE since it enables the RTE generator to extract multiple pieces of information from a reference, e.g. in the above example both the port prototype and data element prototype can be located.

### 3.2.8 AUTOSAR Elements

---

The <ELEMENTS> element provides the encapsulating element used to assemble all AUTOSAR definitions within an <AR-PACKAGE> definition.

The <ELEMENTS> element is not named since it is merely a container for other, named, elements. There are essentially six sets of <ELEMENTS> definitions:

1. Those that define basic types for the system.
2. Those that define the software components.
3. Those that define a logical software architecture built from the components independently of any hardware.
4. Those that define the physical hardware including the types of ECUs and how they are connected to busses.
5. Those that define the network communication frames and protocols on busses between ECUs.



6. Those that define how software components from the logical software architecture are mapped to ECUs and how logical communication is mapped into network communication.

```
<AR-PACKAGE>
  <SHORT-NAME>MyPackage</SHORT-NAME>
  <DESC>My AUTOSAR Test</DESC>
  <ELEMENTS>
    <!-- Type definitions -->
    <!-- Software component elements -->
    <!-- Communication Interfaces -->
    <!-- Software Components -->
    <!-- Mode -->
    <!-- Software (Logical) System Topology -->
    <!-- Hardware (Physical) System Topology -->
    <!-- Network Communication -->
    <!-- Software/Hardware/Network Mapping -->
  </ELEMENTS>
</AR-PACKAGE>
```

The definition of AUTOSAR primitive and complex types is considered in Chapter 5.

The <SENDER-RECEIVER-INTERFACE> and <CLIENT-SERVER-INTERFACE> elements are used to define AUTOSAR interface types—see Chapter 6.

The application SWC-type <APPLICATION-SW-COMPONENT-TYPE> and <SENSOR-ACTUATOR-COMPONENT-TYPE> elements describe software components and are considered in detail in Chapter 7. The <INTERNAL-BEHAVIOR> of the components, which includes which runnable entities exist, how they are triggered with RTE events, how they define exclusive areas etc., is discussed in Chapter 8. Application modes for software components and their interaction use <MODE-DECLARATION-GROUPS> discussed in Chapter 9.

The software component types defined using the SWC-type element are instantiated when a <COMPOSITION-TYPE> element is defined – see Chapter 13.

The <ECU> and <ECU-INSTANCE> combine to define the hardware components that are used to build a vehicle network. These are discussed in Chapter 17.

The <SYSTEM> element maps the software component instances on the composition onto the hardware topology instance. This is covered in Chapter 18. However, this is not the final stage in configuration. When a mapping is such that communication between software components occurs between ECUs (inter-ECU communication) you need to provide additional configuration to tell the RTE how to achieve this using AUTOSAR COM. This is discussed in Chapters 19 and 20.

### 3.3 ECU Configuration Description

RTA-RTE makes use of AUTOSAR OS for executing threads of control, providing time-outs, signaling events etc., and AUTOSAR COM for communication between ECUs.

For the RTE to be generated, RTA-RTE needs to know a small set of configuration data for these AUTOSAR basic software modules.

AUTOSAR basic software uses a different configuration concept from the rest of AUTOSAR that is held in the ECU configuration description file. This file is also an XML file, but the use of XML is significantly different from the rest of AUTOSAR configuration.

Rather than define a dedicated configuration for each basic software module, the ECU configuration description defines how to structure a ECUC module configuration with containers that hold configuration data. An ECUC Container element can hold sub-containers and thus a hierarchy of configuration containers is formed.<sup>1</sup>

```
<ELEMENTS>
  <ECUC-MODULE-CONFIGURATION-VALUES>
    <SHORT-NAME>...</SHORT-NAME>
    <DEFINITION-REF>
      <!-- Reference to module config description -->
      ...
    </DEFINITION-REF>
    <CONTAINERS>
      <ECUC-CONTAINER-VALUE>
        <CONTAINER>
          <!-- Some configurable item -->
          <SUB-CONTAINERS>
            <ECUC-CONTAINER-VALUE>
              <!-- Some nested configurable item -->
            </ECUC-CONTAINER-VALUE>
          </SUB-CONTAINERS>
        </ECUC-CONTAINER-VALUE>
      </CONTAINERS>
    </ECUC-MODULE-CONFIGURATION-VALUES>
  </ELEMENTS>
```

Each ECUC Module Configuration container contains a <DEFINITION-REF> that defines what containers there are and how many of each type are needed for the referenced basic software module.

All definition references have the form /AUTOSAR/<ModuleName>. For the RTE configuration you will use:

- /AUTOSAR/Com
- /AUTOSAR/0s
- /AUTOSAR/Rte

ECU configuration descriptions are verbose, so for clarity this user guide will describe ECU configuration description in terms of which parameters referenced by the

<sup>1</sup>The ECU configuration description is basically a meta-language embedded within XML that is instantiated for each basic software module.

<DEFINITION-REF> are needed, for example: “An /AUTOSAR/0s/0sTask must be created for each RTE task you require.” /AUTOSAR/0s tells you the module for which configuration is needed and 0sTask tells you that an OS task needs to be created. This is equivalent to the following configuration:

```
<ELEMENTS>
  <ECUC-MODULE-CONFIGURATION-VALUES>
    <SHORT-NAME>0s</SHORT-NAME>
    <DEFINITION-REF>/AUTOSAR/0s</DEFINITION-REF>
    <CONTAINERS>
      ...
      <ECUC-CONTAINER-VALUE>
        <SHORT-NAME>RTE_Task1</SHORT-NAME>
        <DEFINITION-REF>/AUTOSAR/0s/0sTask</DEFINITION-REF>
        <PARAMETER-VALUES>
          ...
        </PARAMETER-VALUES>
      </ECUC-CONTAINER-VALUE>
    </CONTAINERS>
  </ECUC-MODULE-CONFIGURATION-VALUES>
</ELEMENTS>
```

## 4 Working with the RTE Generator

---

The separation of the development and integration phases in AUTOSAR is reflected in a two-phase software component development process:

1. Software Component Development: the specification, design and implementation of software components; then
2. Software Component Deployment: the allocation of components to ECUs and the integration of components with the basic software on the ECU.

The two phases of operation allow for initial software component configurations to be made and integrated onto the VFB (through some auxiliary design and development process) then the RTE interface to be generated so that the software components can be implemented before the prototypes are defined and their particular allocation onto an ECU are known.

AUTOSAR R4.0 also defines a third development phase for integration of basic software into an ECU. The basic software phase enables the creation of a “Basic Software Scheduler” that invokes multiple basic software modules at the correct time and integrates their execution with application software components.

The phased development process means that there can be some time between the development of a component type and the allocation of its component prototypes to an ECU. A component could be developed once and re-used multiple times over many generations of vehicles. The component could also be supplied to an integrator in binary form for integration in an ECU with other components that have not yet been written.

The RTE generator supports the phased process by allowing the interface to the RTE to be generated before the component prototype/ECU allocation has been decided. Given a software component description, the RTE generator has enough information to generate interface definition files, allowing engineers to start developing software components. The interface defines the contract between the RTE and the component, that is, what that component must provide if future integration work is to happen easily. This is known as contract phase.

When the system is integrated, and the mapping of software components to ECUs is known, the RTE itself can be generated. However, we now know how many instances of a software component exist, where runnable entities are executing, which communication is local to an ECU and which must be routed across the network etc. The RTE generator can use this information to re-generate the interface definition files to include optimizations based on this additional context. This is known as “RTE phase”.

The following sections discuss the contract and RTE phases in more detail.

## 4.1 Contract Phase

In AUTOSAR, the interface to a software component is captured in a “software component description” written in XML.

The component description is a promise that a software component will have certain properties, for example:

- It will provide a set of ports.
- The ports are typed by defined interfaces.
- It will provide a set of runnable entities.
- It will respond to certain RTE events.

The software component description also contains obligations for the ECU integrator, that is, the person who will create instances of the software component prototype and integrate those prototypes with the basic software on an ECU. For example:

- RTE events must be scheduled at the specified rates.
- Only ports that use the same, compatible, or convertible interfaces can be connected.

The software component description must contain enough information to generate an RTE interface that encapsulates the functional aspects of the contract.

The component descriptions are generated early in the process and can be used with the RTE generation tools to build a framework for developing the components. The “Software Component Descriptions” form the input to the first part of the RTE generation process where the interface contracts, in the form of C header files, can be generated without reference to the final system or to the allocation of specific components to ECUs.

In the contract phase, the RTE generator produces header files to be used in the components you write.

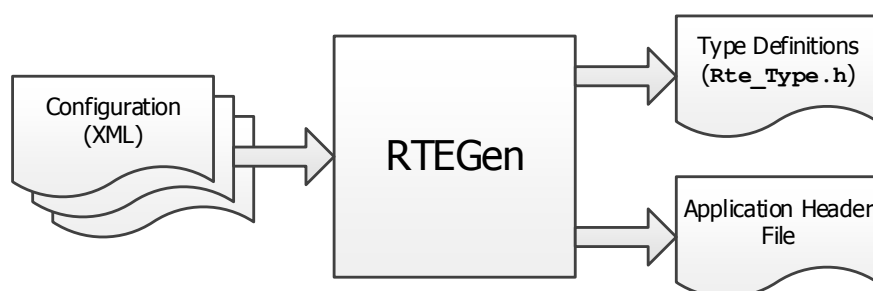


Figure 4.1: Contract Phase

#### 4.1.1 Generation

---

In contract phase, RTA-RTE generates software component-specific application header files in their most general form. The header files define the contract between the component and the system as a whole, and are suitable for both binary-code and source-code components. When running in the contract phase, the RTE generator only needs access to the software component description file(s). It is not necessary to have any information about system deployment.

During contract phase, the RTE is generated for a named software component with the following command line:

```
RTEGen -c <component> [options] <files>.arxml
```

Where <component> must be an absolute reference to a software-component type defined in the input. For details on additional options please see the *RTA-RTE Reference Manual*.

Each execution of the RTE generator in contract phase produces the application header file for a single component.

#### 4.1.2 Software Component-Specific Header Files

---

Access to the RTE API for each declared component type is provided by a component type-specific “application header file”. This file will be generated twice during your system development process:

1. During the contract phase where a generic version of the file is generated.
2. During the RTE phase where an optimized version is generated.

The name of the application header file is derived from the component type name assigned by the vendor, rather than the name assigned by the system integrator to the component prototype during deployment.

The header file names have the following format:

```
Rte_<ComponentTypeName>.h
```

The component type name is used for two reasons:

1. The same component code is used for all component prototypes.
2. Object-code software components may be compiled before the number, or names, of all the component prototypes used in the ECU are known.

The definitions in the XML file are used to define the APIs, so only valid runnable entities may be declared without an error occurring when the component is compiled.

The RTE generator also generates an “application types file” that defines types specific to a particular SW-C type. The name of this file is

`Rte_<ComponentTypeName>_Type.h`

### Component Type & Instance Declarations

---

For each component type declaration, the RTE generator creates a component data structure type. The definition of this type is unique for each component and is written to the component type header file.

When the RTE generator is run in the RTE phase, the RTE implementation includes optimizations that can reduce the run-time cost of the instance handle to zero. For example, when only one prototype of a component is mapped to an ECU, the instance handle is not needed and can be elided.

#### 4.1.3 Object-code Support

---

The RTE supports both application software components where the source is available (“source-code software components”) and application software components where only the object code (“object-code software components”) is available.

The header files generated in contract phase contain sufficient detail to allow a component to be developed and shipped as object files to the ECU integrator.

The only effect of shipping components in object-code format is that they cannot make use of the optimizations that the RTE generator may make when the allocation of component instances to ECU instances is known.

#### 4.2 RTE Phase

---

Before using RTA-RTE in RTE phase, a significant amount of system engineering will be needed. The AUTOSAR development process assumes that there are several inputs to the system engineering process:

- “Software component descriptions” that define the software components, their ports, internal behaviors, implementation characteristics, and the interfaces provided and required by their ports. These are the same descriptions as used in contract phase.
- “ECU resource descriptions” that define the ECU hardware characteristics (e.g. communication ports).
- A “System constraint description” that defines aspects of the system (e.g. communication protocols).

To build an AUTOSAR system (i.e. a set of software components mapped to ECUs that communicate over a network) you must define:


- “ECU configuration description” that defines which software components are mapped to which ECUs, the resources available on the ECU, and so on.

- “System configuration description” that defines the network topology, how inter-ECU communication is mapped to the physical network, and so on.
- “ECU Configuration” that defines the mapping between elements; for example, the mapping of runnable entities to AUTOSAR Operating System tasks and the mapping of AUTOSAR signals to AUTOSAR COM signals.

Once you have configured your AUTOSAR system with an allocation of component prototypes to ECU instances the RTE generator is used in “RTE Generation” phase to create:

1. The implementation of the RTE itself, `Rte.c`, and the generated library `Rte.lib.c`.
2. Optimized component header files that exploit mapping knowledge provided by your configuration.
3. Operating system tasks that package your runnable entities.
4. (optional) An operating system configuration file for the RTE- generated objects and required behavior.
5. (optional) A communication stack configuration file for inter-ECU communication configuration.

Figure 4.2 illustrates the basic process.

 *The AUTOSAR configuration process includes tools for the generation of RTE and generation/configuration of the OS and the communication stack. Within RTA-RTE all three steps are integrated into the single RTE generator.*

#### 4.2.1 Generation

---

In RTE phase, the RTA-RTE generator generates optimized application header files suitable for compiling source-code components and, optionally, configuration files for the communication stack and operating system. When running in RTE phase, the RTE generator needs access to all system deployment information.

For RTE phase, the RTE is generated for a named ECU instance with the following command line:

```
RTEGen -r <ecu instance> [options] <files>.arxml
```

The RTE is generated for each software component and each ECU specified in the configuration file using the command-line RTE generator tool. Each execution of the RTE generator in RTE phase produces the generated RTE for a single ECU instance.

#### 4.2.2 RTE Implementation File

---

The RTE is generated as one or more C modules. Each module must be compiled according to the dependency information created by the RTE.

The module `Rte.c` contains the core of the generated RTE and, when the RTE generator is operating in compatibility mode, the generated task bodies.



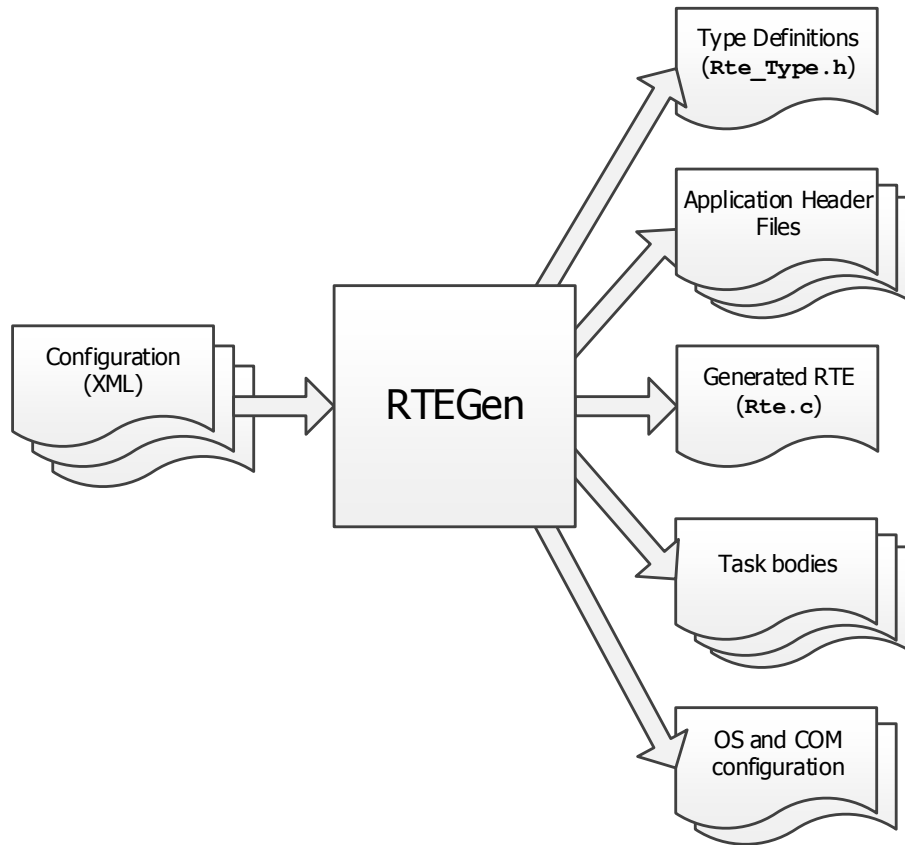


Figure 4.2: RTE Phase

#### 4.2.3 Task Bodies

During “RTE Generation” phase, the RTA-RTE RTE generator creates task bodies and assigns runnable entities to them based on the mapping specified in the ECU configuration.

The names of the tasks are used verbatim from the configuration file. If a task called MyTask is declared then a task called MyTask will be generated. All task names in an ECU must be unique.

**ETAS** *In compatibility mode, all RTE tasks are generated in the Rte.c source code file. When using vendor mode Each task body is written to a separate file named <taskname>.c. You must ensure that the tasks generated by the RTE generator are compiled and linked with the rest of your ECU code.*

The generated task bodies are appropriate for the selected OS, e.g. for AUTOSAR OS the task is declared using the TASK macro and the body includes the TerminateTask API call.

#### 4.3 Basic Software Phase

The primary inputs within this phase are the “Basic Software descriptions” that define the basic software modules’ internal behavior and implementation characteristics, in-

cluding the interfaces provided and required by the module.

The Basic Software Phase generates the APIs and code to support BSW code only. If the input XML contains Software Component configuration then the configuration is rejected by RTEGen.

#### 4.4 The Development Process

---

Figure 4.3 shows the five key stages in the generation of an RTE-based ECU:

1. The initial “contract” phase of RTE generation defines the interface (component-specific “application header” files) for each software component based on information from the software component descriptions. A software component compiled with a contract phase application header can be delivered to the integrator as object-code.
2. The “RTE” phase generates an RTE for a specific ECU, configured from the “ECU Extract”. The ECU Extract is a slice of the System Configuration Description, containing only information related to a single ECU. This can be distributed to the developers of the ECU without the risk of disseminating confidential information about other ECUs.  
  
A software component compiled with an RTE phase application header can be delivered to the integrator as source-code.
3. The RTE generator (optionally) creates configuration files for the ECU, including OS configuration and dependency information for the creation of “makefiles” or build scripts for building the RTE.
4. The generated RTE is compiled along with application software components using build information created by the RTE generator.
5. The RTE library file is compiled and linked with the other application code to form the executable ECU image.

#### 4.5 Samples

---

RTA-RTE can generate sample application source files using the `--samples=swc` command-line option. The name of the generated sample file is `Rte_<swc>.c` where `swc` is the software component’s short name.

The generated sample file contains an empty function for each runnable entity. For example, given a software component `swcA` that supports multiple instantiation and contains a single time triggered runnable with entry point `swcA_re_te1`, the generated sample file would contain:

```
/** @file      Rte_swcA.c
 *
 * @brief      RTE Sample SWC implementation skeleton file
 */
```

```
#include "Rte_swCA.h"

FUNC(void, RTE_APPL_CODE) swCA_re_te1(CONSTP2CONST(Rte_CDS_swCA,
    AUTOMATIC, RTE_CONST) self)
{
    /* ... */
}
```

The `--samples=swc` option is most useful in contract phase before starting implementation of a software component.

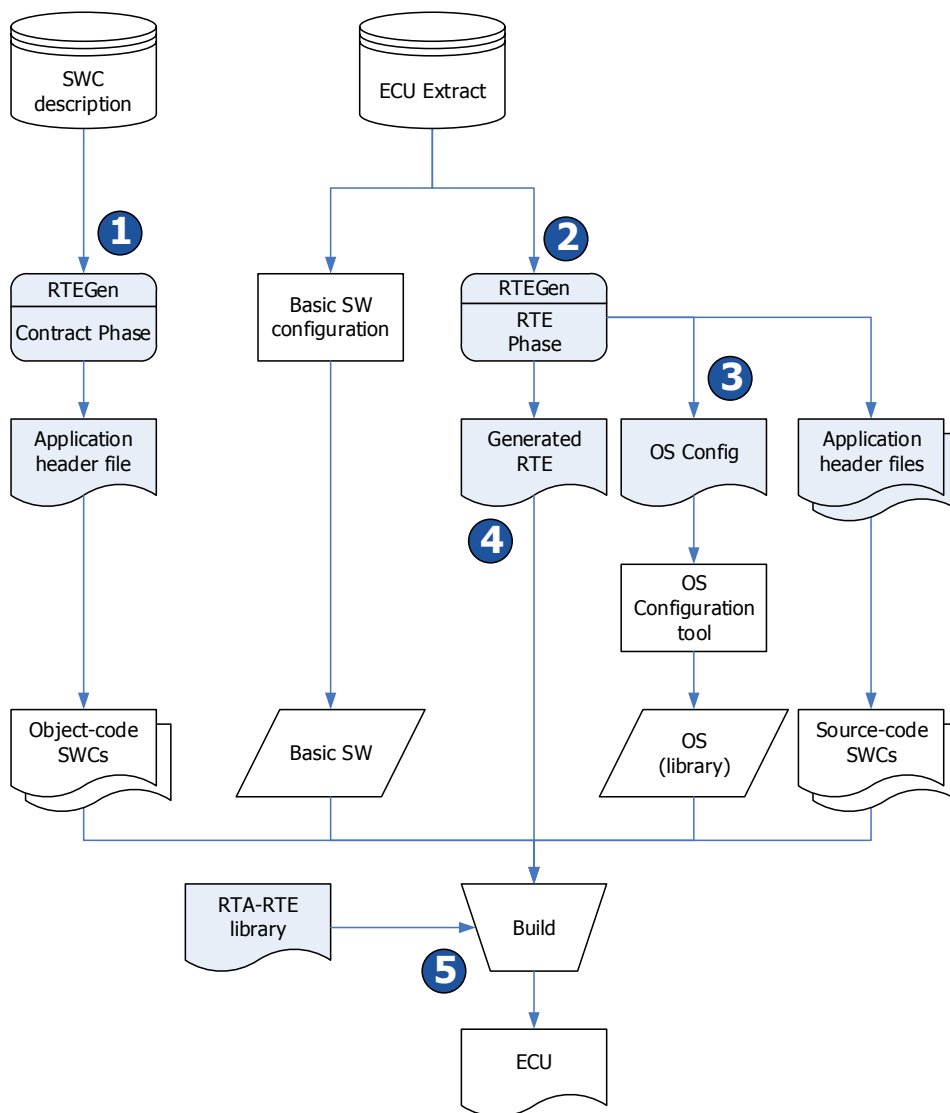


Figure 4.3: AUTOSAR Design Flow—Supplier based RTE Development

## **Part II**

# **Developing Software Components**

## 5 Type System

---

AUTOSAR defines three layers of data type abstraction as illustrated in Figure 5.1.

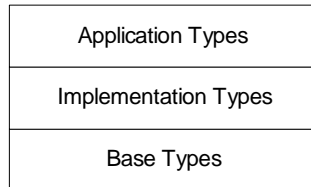


Figure 5.1: Abstraction Levels for Describing Data Types

ApplicationDataTypes are defined in physical terms, i.e. by reference to some CompuMethod, Unit and PhysicalDimension. This allows Application authors to create Software Components without deciding the C data type too early in the lifecycle. ApplicationDataTypes also support automatic conversion of values from one Unit to another. (See section 5.5 and section 13.5 for more information.)

ImplementationDataTypes model C types in the generated code. AUTOSAR specifies a set of common ImplementationDataTypes known as the Platform Types, to make platform-independent modeling and design easier.

Finally, Base Types describe the hardware-specific aspect of the data type, e.g. the width, the alignment and the encoding. In particular, the shortName of the SwBaseType element does not appear in the generated code.

The following sections expand these topics.

### 5.1 Application Data Types

---

Application data types are defined in physical terms, i.e. by reference to some Unit which, in turn, references some PhysicalDimension. It is recommended to model VariableDataPrototypes by referencing ApplicationDataTypes:

- Using ApplicationDataTypes allows the System Integrator to choose the ImplementationDataType later in the development cycle,
- RTE supports data conversion of values between different ApplicationDataTypes (Section 5.5, section 13.5),
- ApplicationDataTypes contain the necessary information to support measurement and calibration tools.

The shortName of an ApplicationDataType is only used within the scope of a SoftwareComponentType, so it is supported to have multiple ApplicationDataTypes with the same name when integrating several SWCTs on a single ECU (but not within a single SWCT).

The shortName of an ApplicationDataType is not used in generated code. In particular, the RTE APIs are defined in terms of the mapped ImplementationDataTypes (according to the DataTypeMapping—see Section 5.4).

The attributes of ApplicationDataTypes include the range of physical values, the relationship to physical space (i.e. references to CompuMethod and/or Unit) and, for complex types, the data structure. The definition of an ApplicationDataType does not consider implementation details such as width in bits, endianness, etc.

For example, an ApplicationPrimitiveDataType might be declared as follows:

```
<PHYSICAL-DIMENSION>
  <SHORT-NAME>length_in_metres</SHORT-NAME>
  <LENGTH-EXP>1</LENGTH-EXP>
</PHYSICAL-DIMENSION>
<UNIT>
  <SHORT-NAME>metres</SHORT-NAME>
  <FACTOR-SI-TO-UNIT>1</FACTOR-SI-TO-UNIT>
  <OFFSET-SI-TO-UNIT>0</OFFSET-SI-TO-UNIT>
  <PHYSICAL-DIMENSION-REF DEST='PHYSICAL-DIMENSION'>/package/
    length_in_metres</PHYSICAL-DIMENSION-REF>
</UNIT>
<COMPU-METHOD>
  <SHORT-NAME>cm_id_metres</SHORT-NAME>
  <CATEGORY>IDENTICAL</CATEGORY>
  <UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
</COMPU-METHOD>
<APPLICATION-PRIMITIVE-DATA-TYPE>
  <SHORT-NAME>MyMetresType</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-CALIBRATION-ACCESS>NOT-ACCESSIBLE</SW-CALIBRATION-ACCESS>
        <COMPU-METHOD-REF>/package/cm_id_metres</COMPU-METHOD-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>
```

Starting at the top, we have a PhysicalDimension representing e.g. Length or luminosity or temperature or speed etc.

Below that we have a Unit. For this example, we show the SI unit for length, the metre. Similarly we could model the kilometre by setting FactorSIToUnit at 1000.

The CompuMethod is of Category IDENTICAL which means that the value on the ECU at runtime is the same as the value of the physical quantity.

Additionally, we might define another application type like below:

```
<COMPU-METHOD>
```

```

<SHORT-NAME>cm_centimetres</SHORT-NAME>
<CATEGORY>LINEAR</CATEGORY>
<UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
<COMPU-INTERNAL-TO-PHYS>
  <COMPU-SCALES>
    <COMPU-SCALE>
      <COMPU-RATIONAL-COEFFS>
        <COMPU-NUMERATOR>
          <V>0</V>
          <V>1</V>
        </COMPU-NUMERATOR>
        <COMPU-DENOMINATOR>
          <V>100</V>
        </COMPU-DENOMINATOR>
      </COMPU-RATIONAL-COEFFS>
    </COMPU-SCALE>
  </COMPU-SCALES>
</COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
<APPLICATION-PRIMITIVE-DATA-TYPE>
  <SHORT-NAME>MyCMTtype</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-CALIBRATION-ACCESS>NOT-ACCESSIBLE</SW-CALIBRATION-ACCESS>
        <COMPU-METHOD-REF>/package/cm_centimetres</COMPU-METHOD-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>

```

Here the CompuMethod `cm_centimetres` contains a CompuDenominator of 100. Since it also references Unit `metres` a VariableDataPrototype of type `MyCMTtype`, it can be connected to a VariableDataPrototype of type `MyMetresType` and RTA-RTE will create the necessary code so that, e.g. a transmission of `100cm` will arrive on the receiver as `1m`.

To support more complex data types, an `ApplicationDataType` can be composed of other `ApplicationDataTypes`. This form of recursive definition permits *records* and *arrays* to be defined.

When the RTE is generated, any used `Application Data Types` must be mapped to implementation types. See Section 5.3 and 5.4 for details.

### 5.1.1 Arrays

To configure an array type in the application domain, create an `ApplicationArrayDataType` that contains a reference to the desired `ApplicationDataType` of the elements and the `ArraySize` set to the number of elements required.



**ETAS** This release of RTA-RTE does not support dynamic-sized arrays. *ArraySize-Semantics* must therefore be set to *FIXED-SIZE*.

For example an array of four lengths in centimetres could be represented with the following type:

```
<APPLICATION-ARRAY-DATA-TYPE>
  <SHORT-NAME>MyCMArrayType</SHORT-NAME>
  <CATEGORY>ARRAY</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-CALIBRATION-ACCESS>NOT-ACCESSIBLE</SW-CALIBRATION-ACCESS>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
  <ELEMENT>
    <SHORT-NAME>element</SHORT-NAME>
    <CATEGORY>VALUE</CATEGORY>
    <TYPE-TREF DEST='APPLICATION-PRIMITIVE-DATA-TYPE'>/package/MyCMType</
      TYPE-TREF>
    <ARRAY-SIZE-SEMANTICS>FIXED-SIZE</ARRAY-SIZE-SEMANTICS>
    <MAX-NUMBER-OF-ELEMENTS>4</MAX-NUMBER-OF-ELEMENTS>
  </ELEMENT>
</APPLICATION-ARRAY-DATA-TYPE>
```

Arrays of complex types can also be created, by referencing an `ApplicationArrayDataType` or `ApplicationRecordDataType` as the element type and setting the element Category to `ARRAY` or `STRUCTURE` respectively.

### 5.1.2 Structures (also known as Record types)

To configure a record type in the application domain, to be mapped to a struct in the implementation domain, create an `ApplicationRecordDataType` that contains an `ApplicationRecordElement` for each structure member, referencing the desired `ApplicationDataType`.

For example, the coordinates, in centimetres, of a point on a surface could be represented with the following type:

```
<APPLICATION-RECORD-DATA-TYPE>
  <SHORT-NAME>MyPointType</SHORT-NAME>
  <CATEGORY>STRUCTURE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-CALIBRATION-ACCESS>NOT-ACCESSIBLE</SW-CALIBRATION-ACCESS>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
  <ELEMENTS>
    <APPLICATION-RECORD-ELEMENT>
      <SHORT-NAME>x</SHORT-NAME>
```

```

    <CATEGORY>VALUE</CATEGORY>
    <TYPE-TREF DEST='APPLICATION-PRIMITIVE-DATA-TYPE'>/package/MyCMType</
      TYPE-TREF>
  </APPLICATION-RECORD-ELEMENT>
</APPLICATION-RECORD-ELEMENT>
  <SHORT-NAME>y</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <TYPE-TREF DEST='APPLICATION-PRIMITIVE-DATA-TYPE'>/package/MyCMType</
    TYPE-TREF>
</APPLICATION-RECORD-ELEMENT>
</ELEMENTS>
</APPLICATION-RECORD-DATA-TYPE>

```

Record elements can themselves be complex types. Simply reference an `ApplicationArrayDataType` or `ApplicationRecordDataType` as the `ApplicationRecordElement` type and set the `ApplicationRecordElement` Category to `ARRAY` or `STRUCTURE` respectively.

## 5.2 Base Types

`SwBaseTypes` describe data types in hardware terms (size in bits, memory alignment, encoding). They form the basis on which the `Implementation Data Types` are built. A base type can be referenced by several `Implementation Data Types`. In this way, the C declarations of `Implementation Data Types` can be adjusted for different hardware platforms by changing only the attributes of the referenced `Base Type`.

For example, the semantics of two 8-bit integer base types might be modeled as follows:

```

<SW-BASE-TYPE>
  <SHORT-NAME>my_sint8</SHORT-NAME>
  <BASE-TYPE-SIZE>8</BASE-TYPE-SIZE>
  <BASE-TYPE-ENCODING>2C</BASE-TYPE-ENCODING>
  <MEM-ALIGNMENT>1</MEM-ALIGNMENT>
  <NATIVE-DECLARATION>char</NATIVE-DECLARATION>
</SW-BASE-TYPE>
<SW-BASE-TYPE>
  <SHORT-NAME>my_uint8</SHORT-NAME>
  <BASE-TYPE-SIZE>8</BASE-TYPE-SIZE>
  <BASE-TYPE-ENCODING>NONE</BASE-TYPE-ENCODING>
  <MEM-ALIGNMENT>1</MEM-ALIGNMENT>
  <NATIVE-DECLARATION>unsigned char</NATIVE-DECLARATION>
</SW-BASE-TYPE>

```



*The base type's shortName never appears in the generated code; it is only used as a reference target within the input model. Only Implementation Data Types are present in the generated code along with the nativeDeclaration of the referenced basetype.*

The encoding attribute can be either `NONE`, for unsigned types, or `2C`, `1C` or `SM` for signed types.

The `nativeDeclaration` attribute is optional but strongly recommended. It is usually an error to omit the `nativeDeclaration`, except as part of a deprecated scheme for suppressing the generation of a typedef. (See 5.3).

If present, it contains a raw C type-expression necessary to produce a type of the same width and signedness as the `SwBaseType` itself. This text will typically be written on the left hand side of a typedef of a referencing `ImplementationDataType`, e.g.

```
typedef unsigned int myImplementationDataType;
```



*It is not correct to write the name of a PlatformType in the nativeDeclaration, only raw C like unsigned int, signed long, etc. If you really think you need to do this, consider defining an ImplementationDataType of Category TYPE\_REFERENCE instead (See 5.3.6)*

### 5.3 Implementation Data Types

`ImplementationDataTypes` represent types in the generated C code. The `shortName` of an `ImplementationDataType` defines the symbol used in C to access the type, e.g. in APIs and in user code.

Typically, `ImplementationDataTypes` have their `typeEmitter` attribute set to `RTE` and result in typedef declarations being written to the file `Rte_Type.h`.

Some `ImplementationDataTypes` need to be modeled in the input but it is not desirable to generate corresponding typedefs in the RTE code, for example the Platform Types, whose C declarations are present in a standard header file not generated by the RTE Generator (see 5.3.1). To suppress the typedef, specify the `typeEmitter` attribute with a value other than `RTE`.

For backward compatibility, RTA-RTE also includes a deprecated mechanism for suppressing typedefs. If there is no `typeEmitter` attribute on the `ImplementationDataType`, the generation of the typedef depends on the `SwBaseType.nativeDeclaration`. If the `nativeDeclaration` is absent or empty, and there is no `typeEmitter` present on the corresponding `ImplementationDataType`, no typedef is written.



*It is an error to have a missing or empty SwBaseType.nativeDeclaration if the ImplementationDataType.typeEmitter is set to RTE.*

It is supported to define `VariableDataTypes` in SWCTs by reference to Application Data Types or Implementation Data Types, and it is supported to mix them freely. However it is recommended to use only Application Data Types in SWCTs for the reasons already given:

- Avoid deciding the C type too early in the lifecycle,
- Allow data conversion between values that represent the same physical quantity,
- Support measurement and calibration tools.

RTA-RTE always uses Implementation Data Types in generated APIs. If the corresponding VariableDataPrototype is defined by reference to an ApplicationDataType, the mapped ImplementationDataType is used in the API signature.

AUTOSAR data types must be declared in the XML input configuration if they are to be used in Interfaces, ARTypedPerInstanceMemorys, etc. In addition, *all PlatformTypes must be declared on the input regardless of whether they are used in the input model.* This is because RTA-RTE depends on PlatformTypes to create its internal types.

It is supported that multiple ImplementationDataTypes in the input XML have the same name, provided that the resulting definitions in the generated code are identical. This makes it easier to integrate packages from different third parties, which may for example include (re)definitions of Platform Types.

This section explains how to configure RTA-RTE to make types available to the C program and to the rest of the configuration input.

### 5.3.1 Platform Types

AUTOSAR specifies a set of platform types for use in C code. These are ImplementationDataTypes whose purpose is to provide a set of types with the same semantics across different target hardware. RTA-RTE uses PlatformTypes when it needs to create types for internal variables.

Unlike most ImplementationDataTypes, the PlatformTypes are also defined in C language in the file PlatformTypes.h. To support this, they reference SwBaseTypes with empty or missing nativeDeclaration attributes (See 5.2).

AUTOSAR (from R4.0.3) also specifies the correct definitions and package name of the Platform Types. RTA-RTE expects to find the package AUTOSAR\_Platform/ImplementationDataTypes on the input, and it will look in that package for any Platform Types that it needs. RTA-RTE will reject the configuration if it does not include such a package.

RTA-RTE ships with a suitable Platform Types package (in the file External/AUTOSAR/AUTOSAR\_MOD\_PlatformTypes.arxml). Note that the corresponding SwBaseTypes are defined in a separate file External/AUTOSAR/AUTOSAR\_MOD\_PlatformBaseTypes.arxml.

The Platform Types defined in *AUTOSAR Specification of Platform Types* and in the standard header file PlatformTypes.h are:

- sint8 – 8-bit signed integer.
- uint8 – 8-bit unsigned integer.
- sint16 – 16-bit signed integer.
- uint16 – 16-bit unsigned integer.

- sint32 – 32-bit signed integer.
- uint32 – 32-bit unsigned integer.
- float32 – single precision floating point.
- float64 – double precision floating point.
- uint8\_least – at least 8-bit unsigned integer.
- uint16\_least – at least 16-bit unsigned integer.
- uint32\_least – at least 32-bit unsigned integer.
- sint8\_least – at least 7-bit signed integer (plus sign bit).
- sint16\_least – at least 15-bit signed integer (plus sign bit).
- sint32\_least – at least 31-bit signed integer (plus sign bit).
- boolean – for use with TRUE/FALSE.

When a platform type is used in the input model, e.g. in an Interface, it might be referenced in the standard location (/AUTOSAR\_Platform/ImplementationDataTypes) or the definition of the platform type might be repeated in another ImplementationDataType element with the same shortName. Both styles are supported, provided the ImplementationDataTypes do not give rise to conflicting definitions. In particular, the SwBaseTypes should both have empty or missing nativeDeclaration, they should have matching baseTypeWidth and matching encoding.



*The correct way to represent a platform type in the XML input is by using an ImplementationDataType of Category VALUE with the ShortName equal to the name of the AUTOSAR Platform Type, which refers to a SwBaseType element that does not have a nativeDeclaration.*

```
<AR-PACKAGE>
  <SHORT-NAME>pkg</SHORT-NAME>
  <ELEMENTS>
    <DATA-CONSTR>
      <SHORT-NAME>dcI_uint8</SHORT-NAME>
      <DATA-CONSTR-RULES>
        <DATA-CONSTR-RULE>
          <INTERNAL-CONSTRS>
            <LOWER-LIMIT>0</LOWER-LIMIT>
            <UPPER-LIMIT>255</UPPER-LIMIT>
          </INTERNAL-CONSTRS>
        </DATA-CONSTR-RULE>
      </DATA-CONSTR-RULES>
    </DATA-CONSTR>

    <SW-BASE-TYPE>
      <SHORT-NAME>bt_null_uint8</SHORT-NAME>
```

```

    <BASE-TYPE-SIZE>8</BASE-TYPE-SIZE>
    <MEM-ALIGNMENT>1</MEM-ALIGNMENT>
    <!-- does not have NATIVE-DECLARATION -->
</SW-BASE-TYPE>

<IMPLEMENTATION-DATA-TYPE>
  <SHORT-NAME>uint8</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <BASE-TYPE-REF>/pkg/bt_null_uint8</BASE-TYPE-REF>
        <COMPU-METHOD-REF>/pkg/cm_id</COMPU-METHOD-REF>
        <DATA-CONSTR-REF>/pkg/dcI_uint8</DATA-CONSTR-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</IMPLEMENTATION-DATA-TYPE>
</ELEMENTS>
</AR-PACKAGE>

```

Note that the ShortName of the SwBaseType never appears in the generated code in any circumstances. It is only ever used as a means of referencing the SwBaseType in the XML.

### 5.3.2 Custom Primitive Implementation Data Types

It is supported to configure new ImplementationDataTypes, either by reference to existing platform types (recommended) or completely from scratch (not recommended). Enumeration (text table) types can be modeled in this way. The configured types are written to the file Rte\_Type.h.

It is not always necessary to define a completely new type just to give a semantically significant name to a type. Instead, an ImplementationDataType can be created with Category TYPE\_REFERENCE in order to emit a typedef from an existing ImplementationDataType to the more useful name, and to provide a single point at which the actual type of the chosen typename can be changed.

For example, the following definition of type MyCommandIdType:

```

<IMPLEMENTATION-DATA-TYPE>
  <SHORT-NAME>MyCommandIdType</SHORT-NAME>
  <CATEGORY>TYPE_REFERENCE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <IMPLEMENTATION-DATA-TYPE-REF>/AUTOSAR_Platform/
          ImplementationDataTypes/uint8</IMPLEMENTATION-DATA-TYPE-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</IMPLEMENTATION-DATA-TYPE>

```

Results in the following typedef in Rte\_type.h:

```
typedef uint8 MyCommandIdType;
```

The new TYPE\_REFERENCE type can then be referenced in Interfaces, ARTypedPerInstanceMemorys, InterRunnableVariables, etc.

It is possible (but not recommended) to define a new ImplementationDataType that is not defined by reference to an existing ImplementationDataType but instead references a SwBaseType that has a nativeDeclaration.

To do that, set the Category to VALUE and use a BaseTypeRef pointing to the chosen SwBaseType. The emitted typedef in Rte\_type.h uses the nativeDeclaration of the SwBaseType as in the following example, where the nativeDeclaration is unsigned int:

```
typedef unsigned int MyMaskType;
```



*This method is not recommended because the type may become hardware-platform dependent.*

The <DATA-CONSTR-REF> reference within a new ImplementationDataType declaration defines the constraints of the type. In particular, RTA-RTE needs to know the type limits:

```
<DATA-CONSTR>
  <SHORT-NAME>myDataConstraint</SHORT-NAME>
  <DATA-CONSTR-RULES>
    <DATA-CONSTR-RULE>
      <INTERNAL-CONSTRS>
        <LOWER-LIMIT>0</LOWER-LIMIT>
        <UPPER-LIMIT>6000</UPPER-LIMIT>
      </INTERNAL-CONSTRS>
    </DATA-CONSTR-RULE>
  </DATA-CONSTR-RULES>
</DATA-CONSTR>
```



*RTA-RTE does not validate that the range specified by the constraints is compatible with the specified native declaration.*

### 5.3.3 Enumerated Types

To declare an enumerated type, create a new ImplementationDataType of category TYPE\_REFERENCE and add an ImplementationDataTypeRef referencing a suitable PlatformType to store the enumeration values.

Create a CompuMethod of category TEXTTABLE and reference the CompuMethod from the new ImplementationDataType. The CompuMethod supplies the symbolic and numeric values for the enumerated type, for example:

```
<IMPLEMENTATION-DATA-TYPE>
  <SHORT-NAME>myEnum</SHORT-NAME>
```

```

<CATEGORY>TYPE_REFERENCE</CATEGORY>
<SW-DATA-DEF-PROPS>
  <SW-DATA-DEF-PROPS-VARIANTS>
    <SW-DATA-DEF-PROPS-CONDITIONAL>
      <COMPU-METHOD-REF DEST='COMPU-METHOD'>myEnumValues</COMPU-METHOD-
        REF>
      <IMPLEMENTATION-DATA-TYPE-REF DEST='IMPLEMENTATION-DATA-TYPE'>/
        AUTOSAR_Platform/ImplementationDataTypes/sint8</IMPLEMENTATION-
        DATA-TYPE-REF>
    </SW-DATA-DEF-PROPS-CONDITIONAL>
  </SW-DATA-DEF-PROPS-VARIANTS>
</SW-DATA-DEF-PROPS>
</IMPLEMENTATION-DATA-TYPE>
<COMPU-METHOD>
  <SHORT-NAME>myEnumValues</SHORT-NAME>
  <CATEGORY>TEXTTABLE</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <LOWER-LIMIT>0</LOWER-LIMIT>
        <UPPER-LIMIT>0</UPPER-LIMIT>
        <COMPU-CONST><VT>ZERO</VT></COMPU-CONST>
      </COMPU-SCALE>
      <COMPU-SCALE>
        <LOWER-LIMIT>1</LOWER-LIMIT>
        <UPPER-LIMIT>1</UPPER-LIMIT>
        <COMPU-CONST><VT>ONE</VT></COMPU-CONST>
      </COMPU-SCALE>
      <COMPU-SCALE>
        <LOWER-LIMIT>2</LOWER-LIMIT>
        <UPPER-LIMIT>2</UPPER-LIMIT>
        <COMPU-CONST><VT>TWO</VT></COMPU-CONST>
      </COMPU-SCALE>
      <COMPU-SCALE>
        <LOWER-LIMIT>3</LOWER-LIMIT>
        <UPPER-LIMIT>3</UPPER-LIMIT>
        <COMPU-CONST><VT>THREE</VT></COMPU-CONST>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>

```

RTA-RTE will reject invalid CompuMethods, for example, those with duplicate labels and/or numeric values with different lower- and upper-limits.

Now when the type is used within a SWC-type RTA-RTE creates definitions for use by the SWC implementation. In the Type Header File Rte\_Type.h:

```
typedef uint8 myEnum;
```

And in the Application Type Header File Rte\_<SWC>\_Type.h:



```
#define ZERO ((myEnum)0)
#define ONE ((myEnum)1)
#define TWO ((myEnum)2)
#define THREE ((myEnum)3)
```

#### 5.3.4 Arrays

To configure a fixed-size array, create an `ImplementationDataType` of category `ARRAY` that contains an `ImplementationDataTypeElement` having an `ArraySizeSemantics` of `FIXED-SIZE` and an `ArraySize` set to the number of elements you wish to have in the array. The `shortName` of the `ImplementationDataTypeElement` does not appear in the generated C.

**ETAS** *This release of RTA-RTE does not support dynamic-sized arrays.*

The category and configuration of the `ImplementationDataTypeElement` dictates the type of the array element.



*Rather than use Category VALUE, it is recommended to use TYPE\_REFERENCE. (See Section 5.3.6).*

#### Arrays as Function Parameters

AUTOSAR specifies that arrays used in API calls shall be passed as “pointer to base type”. This does not refer to `SwBaseType` but rather to the type information of the `ImplementationDataTypeElement`. RTA-RTE creates a formal parameter of a pointer to that type, for example:

```
typedef unsigned short myValueType;
typedef myValueType myArrayType[4];
...
// Write of the array type is by pointer to 'base type'
FUNC(Std_ReturnType, RTE_CODE) Rte_Write_myPort_myArrayData(P2CONST(
    myValueType, AUTOMATIC, RTE_APPL_DATA) data);
```

If the array is defined using an `ImplementationDataTypeElement` of a category other than `TYPE_REFERENCE`, there is no typedef for `myValueType`, which may cause the generated code to fail MISRA advisory rule 6.3:

```
typedef unsigned short myArrayType[4];
...
// Write of the array type is by pointer to 'base type'
FUNC(Std_ReturnType, RTE_CODE) Rte_Write_myPort_myData(P2CONST(
    unsigned short, AUTOMATIC, RTE_APPL_DATA) data);
// MISRA advisory 6.3: unsigned used outside of a typedef
// MISRA advisory 6.3: short used outside of a typedef
```

### 5.3.5 Structures (also known as Record types)

To configure a struct type, create an `ImplementationDataType` of category `STRUCTURE` that contains an `ImplementationDataTypeElement` for each structure member. The `ShortName` of the `ImplementationDataTypeElement` becomes the name of the structure member.

For example, the declaration of structure `MyStructType` in the input configuration:

```
<SW-BASE-TYPE>
  <SHORT-NAME>myuint8bt</SHORT-NAME>
  <BASE-TYPE-SIZE>8</BASE-TYPE-SIZE>
  <MEM-ALIGNMENT>1</MEM-ALIGNMENT>
  <NATIVE-DECLARATION>unsigned char</NATIVE-DECLARATION>
</SW-BASE-TYPE>

<IMPLEMENTATION-DATA-TYPE>
  <SHORT-NAME>MyStructType</SHORT-NAME>
  <CATEGORY>STRUCTURE</CATEGORY>
  <SUB-ELEMENTS>
    <IMPLEMENTATION-DATA-TYPE-ELEMENT>
      <SHORT-NAME>member_a</SHORT-NAME>
      <CATEGORY>TYPE_REFERENCE</CATEGORY>
      <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
          <SW-DATA-DEF-PROPS-CONDITIONAL>
            <IMPLEMENTATION-DATA-TYPE-REF>
              /pkg/uint8
            </IMPLEMENTATION-DATA-TYPE-REF>
          </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
      </SW-DATA-DEF-PROPS>
    </IMPLEMENTATION-DATA-TYPE-ELEMENT>
    <IMPLEMENTATION-DATA-TYPE-ELEMENT>
      <SHORT-NAME>member_b</SHORT-NAME>
      <CATEGORY>VALUE</CATEGORY>
      <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
          <SW-DATA-DEF-PROPS-CONDITIONAL>
            <BASE-TYPE-REF>/custom_types/myuint8bt</BASE-TYPE-REF>
            <COMPU-METHOD-REF>/compu_methods/cm_id</COMPU-METHOD-REF>
            <DATA-CONSTR-REF>/data_constrs/dcI_uint8</DATA-CONSTR-REF>
          </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
      </SW-DATA-DEF-PROPS>
    </IMPLEMENTATION-DATA-TYPE-ELEMENT>
  </SUB-ELEMENTS>
</IMPLEMENTATION-DATA-TYPE>
```

Produces the following struct definition in `Rte_Type.h`:

```
typedef struct {
    uint8 member_a; /* recommended */
```

```
    unsigned char member_b; /* Not so safe */
} MyStructType
```

The category and configuration of each `ImplementationDataTypeElement` dictates the type of the corresponding member.



*Rather than use Category VALUE, it is recommended to use TYPE\_REFERENCE. (See Section 5.3.6).*

### 5.3.6 Declaring the elements of Complex Types

The category of an `ImplementationDataTypeElement` can be any of `TYPE_REFERENCE`, `STRUCTURE`, `UNION` or `ARRAY`. `VALUE`, `HOST`, `DATA_REFERENCE` and `FUNCTION_REFERENCE` are also allowed according to the AUTOSAR specification, but are not recommended as noted in the following subsections.

**TYPE\_REFERENCE** This is the **recommended** category for `ImplementationDataTypeElement` instead of `VALUE` or `HOST`. By creating a separate `ImplementationDataType` for the `VALUE` or `HOST` type, and referencing it from the `ImplementationDataTypeElement`, you create a shortname that is used in the declaration and therefore makes the definition clear.

```
typedef uint8 MyElementType;
typedef MyElementType MyArrayType[4];
```

**STRUCTURE / UNION / ARRAY** By declaring an `ImplementationDataTypeElement` that is itself of a complex type, you can create nested complex types, e.g. an array of structs, a struct containing an array and a union, etc. This nesting can be continued without limitation, but remember that using a `TYPE_REFERENCE` can help to keep the configuration clearer.

**VALUE / HOST** An `ImplementationDataTypeElement` of category `VALUE` or `HOST` creates the element anonymously, i.e. there is no named type that characterizes the element. If the `ImplementationDateTypeElement` refers to a `SwBaseType` with a nativeDeclaration, the native declaration is written to the generated C. This approach is **not recommended** because it carries a risk of becoming hardware-dependent. Instead it is recommended to use an `ImplementationDataTypeElement` of category `TYPE_REFERENCE` and refer to a named, platform-safe `ImplementationDataType`.

If an `ImplementationDataTypeElement` references a `SwBaseType` with no nativeDeclaration, the configuration is rejected because there is not enough information to create the declaration in C. (Remember that the `ShortName` of a `SwBaseType` never appears in the C as the name of a type, only the short names of `ImplementationDataTypes` or the native declarations of `SwBaseTypes`.)



When an `ImplementationDataTypeElement` is of category other than `TYPE_REFERENCE`, it causes the C declaration (e.g. `nativeDeclaration`) to be written directly to the generated C where the type is used, e.g. in struct declarations and in API parameters, leading to the use of keywords like `unsigned` or `short` appearing outside of typedefs. This violates MISRA advisory rule 6.3.

**DATA\_REFERENCE / FUNCTION\_REFERENCE** These result in pointer types. It is generally dangerous and forbidden to use a pointer for any communication between SWCs or Runnables within a SWC. It is not recommended to use `DATA_REFERENCE` or `FUNCTION_REFERENCE` for communication.

## 5.4 Type Mappings

### 5.4.1 Data types

A SWC-specific data type mapping is used to map application types onto implementation types. The use of a separate mapping means that different SWC types can use different implementations of the same application type and permits, for example, changing data type characteristics if a SWC is moved to a new hardware platform without changing the SWC itself.



RTA-RTE requires a data type mapping for each application type that is used in the model in order to be able to generate the RTE.

The data type mapping for a SWC is held within a *data type mapping set* element:

```
<DATA-TYPE-MAPPING-SET>
  <SHORT-NAME>dts1</SHORT-NAME>
  <DATA-TYPE-MAPS>
    ...
  </DATA-TYPE-MAPS>
</DATA-TYPE-MAPPING-SET>
```

The data type mapping is associated with a SWC type via a reference to the mapping set from the SWC's internal behavior. If required, one type mapping could be reused for multiple SWCs, or the mapping could be modified without modifying the SWC itself.

A data type mapping contains one or more *data type maps*. Each map references a single application data type and a single implementation data type.

```
<DATA-TYPE-MAP>
  <APPLICATION-DATA-TYPE-REF>
    /pkg/ApplType1
  </APPLICATION-DATA-TYPE-REF>
  <IMPLEMENTATION-DATA-TYPE-REF>
    /pkg/SInt16
  </IMPLEMENTATION-DATA-TYPE-REF>
</DATA-TYPE-MAP>
```

As well as mappings within an individual SWC-type, RTA-RTE also supports a data type mapping for the root composition SWC-type. This means that mappings for a configuration can be specified, and changed, without modification of the individual SWCs.

#### 5.4.2 Mode types

Mode type mappings are used to map mode declaration groups onto implementation types.



RTA-RTE requires a mode type mapping for each used mode declaration group in order to be able to generate the RTE.

As with data type mappings, a mode type mapping is contained within a *data type mapping set* element:

```
<DATA-TYPE-MAPPING-SET>
  <SHORT-NAME>dts1</SHORT-NAME>
  <MODE-REQUEST-TYPE-MAPS>
    ...
  </MODE-REQUEST-TYPE-MAPS>
</DATA-TYPE-MAPPING-SET>
```

The mode type mapping is associated with a SWC type via a reference to the mapping set from the SWC's internal behavior.

A mode type mapping contains one or more *mode request type maps*. Each map references a single implementation data type and a single mode declaration group.

```
<MODE-REQUEST-TYPE-MAP>
  <IMPLEMENTATION-DATA-TYPE-REF>
    /pkg/SInt16
  </IMPLEMENTATION-DATA-TYPE-REF>
  <MODE-GROUP-REF>
    /pkg/mdg1
  </MODE-GROUP-REF>
</MODE-REQUEST-TYPE-MAP>
```

Multiple mode type mapping elements can reference the same implementation data type.

## 5.5 Application Types And Numerical Representations

### 5.5.1 Concepts

RTA-RTE Supports AUTOSAR Data Conversion (see 13.5), where a connection may be made between data items (or the sub-elements of complex-typed data items) of different, but related, `ApplicationPrimitiveDataTypes`. This depends on the correct modelling of `ApplicationPrimitiveDataTypes` and how their numerical values relate to physical quantities.

## Linear Data Conversion

When `DataPrototypes` with incompatible, but related, `ApplicationPrimitiveDataTypes` are explicitly connected via a `PortInterfaceMapping` (see section 13.3.4), RTA-RTE seeks to generate a linear conversion (rescaling and offset) of values between them. It evaluates the `CompuMethods` and `Units` for both `DataPrototypes`, or for the primitive sub-elements if the `DataPrototypes` have complex type, to derive what data transformation is needed.

Linear data conversion is also attempted between the primitive sub-elements of `DataPrototypes` with complex `ApplicationDataTypes`, provided that they are shape-compatible.

Conceptually, the data conversion function is a functional composition of the individual transformations from the transmitted value  $x$ , through the Physical representation, to the SI representation, to the Physical representation in the receiver's Unit, to the internal received value  $y$ :

$$f = cm_{rx} \circ unit_{rx} \circ unit_{tx} \circ cm_{tx}$$

or

$$y = cm_{rx}(unit_{rx}(unit_{tx}(cm_{tx}(x))))$$

Where  $cm_{tx}$ ,  $unit_{tx}$ ,  $unit_{rx}$  and  $cm_{rx}$  represent each step of the conversion as a linear function that maps the input value to an intermediate output value. Figure 5.2 illustrates a linear function.

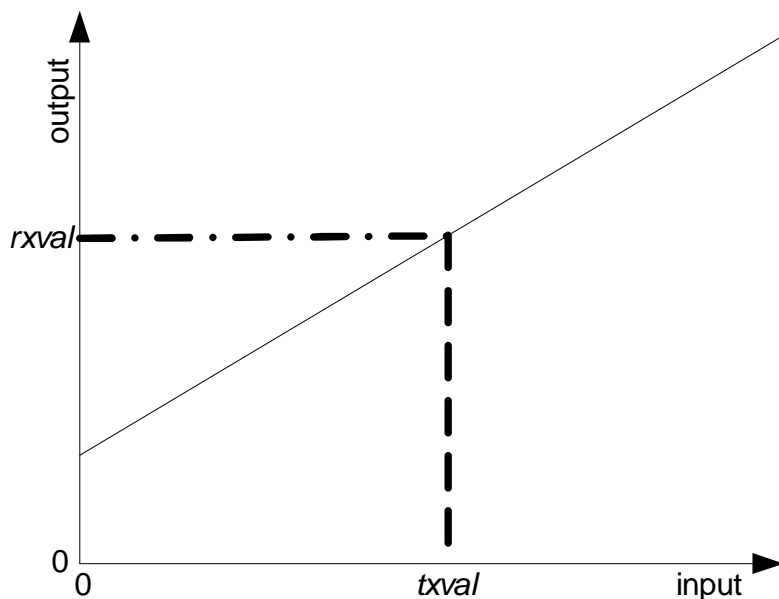


Figure 5.2: Linear data conversion

Since each function is linear it can be expressed as a line of the form  $y = (m \times x) + c$  where  $m$  is the gradient and  $c$  the offset. RTA-RTE derives the values for  $m$  and  $c$  from the transformations specified in the CompuMethods and in the Units:

$cm_{rx}$  is the function represented by the CompuMethod of the receiving type in the “physical-to-internal” direction.

The  $cm_{rx}$  function is “linear” and is derived from the <COMPU-PHYS-T0-INTERNAL> element’s numerators  $num_0$  and  $num_1$  within the <COMPU-NUMERATOR> element and the denominator  $denom$  from the <COMPU-DENOMINATOR> element. The  $cm_{rx}$  function can then be defined as:

$$cm_{rx}(x) = \frac{num_0 + (num_1 \times x)}{denom}$$

$unit_{rx}$  is the function represented by the FactorSIToUnit and OffsetSIToUnit of the receiving type in the “physical-to-internal” direction.

As with the  $cm_{rx}$  function, the  $unit_{rx}$  function is “linear” but the gradient,  $m$ , and offset,  $c$ , are derived from the Unit’s <FACTOR-SI-T0-UNIT> and <OFFSET-SI-T0-UNIT> elements. The  $unit_{rx}$  function can then be defined as:

$$unit_{rx}(x) = (m \times x) + c$$

$unit_{tx}$  is the function represented by the FactorSIToUnit and OffsetSIToUnit of the transmitting type in the “internal-to-physical” direction.

The  $unit_{tx}$  function is “linear” and the gradient,  $m$ , and offset,  $c$ , are derived from the Unit’s <FACTOR-SI-T0-UNIT> and <OFFSET-SI-T0-UNIT> elements. The  $unit_{tx}$  function can then be defined as:

$$unit_{tx}(x) = (m \times x) + c$$

The  $unit_{tx}$  function is the inverse of the  $unit_{rx}$  function.

$cm_{tx}$  is the function represented by the CompuMethod of the transmitting type in the “internal-to-physical” direction.

The  $cm_{tx}$  function is “linear” and is derived from the <COMPU-INTERNAL-T0-PHYS> element’s numerators  $num_0$  and  $num_1$  within the <COMPU-NUMERATOR> element and the denominator  $denom$  from the <COMPU-DENOMINATOR> element. The  $cm_{tx}$  function can then be defined as:

$$cm_{tx}(x) = \frac{num_0 + (num_1 \times x)}{denom}$$

Note that the above description of the conversion functions, e.g.  $cm_{rx}$  etc., is expressed in the same direction as data flow. This is not required: RTA-RTE will derive the inverse function as and when necessary.

## 5.5.2 Modelling Application Types

By correctly modelling Application Data Types with regard to PhysicalDimensions, RTA-RTE can automatically derive any data conversion code needed to transmit values between different numerical representations of the same physical dimension.

### PhysicalDimension

A PhysicalDimension represents the notion of length, or speed, or temperature, etc.

```
<PHYSICAL-DIMENSION>  
  <SHORT-NAME>length_in_metres</SHORT-NAME>  
  <LENGTH-EXP>1</LENGTH-EXP>  
</PHYSICAL-DIMENSION>
```

### Unit

Units express their relationship to a PhysicalDimension by means of FactorSIToUnit and OffsetSIToUnit. For example, a Unit representing the kilometre might have the FactorSIToUnit of 0.001.

FactorSIToUnit and OffsetSIToUnit are specified as decimal literals:

```
<UNIT>  
  <SHORT-NAME>centimetres</SHORT-NAME>  
  <FACTOR-SI-TO-UNIT>0.01</FACTOR-SI-TO-UNIT>  
  <OFFSET-SI-TO-UNIT>0</OFFSET-SI-TO-UNIT>  
  <PHYSICAL-DIMENSION-REF DEST='PHYSICAL-  
  DIMENSION'>/package/length_in_metres</PHYSICAL-DIMENSION-REF>  
</UNIT>
```

An ApplicationDataType may directly reference a Unit, or the Unit may instead be referenced by a CompuMethod (below). If the CompuMethod and ApplicationDataType reference conflicting Units, the configuration will be rejected.

### CompuMethod

The CompuMethod container in the input configuration has a ShortName, a Category, and optionally a reference to a Unit.

If the Category is IDENTICAL, there is nothing more to specify.

If the Category is LINEAR, you must supply the coefficients for a linear conversion function, either a CompuInternalToPhys for the “internal-to-physical” direction, a CompuPhysToInternal for the “physical-to-internal” direction, or both.



*If both are specified, CompuInternalToPhys and CompuPhysToInternal must be the inverse of one another.*

If only one of CompuInternalToPhys and CompuPhysToInternal is supplied, RTA-RTE calculates the inverse automatically. If both are provided, RTA-RTE honors them both but **does not** check that one is truly the inverse of the other.





When specified in a *CompuMethod* with category *LINEAR*, the *CompuInternalToPhys* and *CompuPhysToInternal* must contain exactly one *CompuScale* element containing exactly one *CompuRationalCoeffs* element.

The *CompuRationalCoeffs* element contains a *CompuNumerator* element that specifies two values,  $num_0$  and  $num_1$ , and a *CompuDenominator* with one value,  $denom$ . The meaning of these values is as follows:

$$f(x) = \frac{num_0 + (num_1 \times x)}{denom}$$

where  $num_0$  is the first value in the *CompuNumerator* element,  $num_1$  is the second value in the *CompuNumerator* element, and  $denom$  is the value in the *CompuDenominator*.

All numerators and denominators in *CompuRationalCoeffs* must be integers. For example, to represent  $(0.5 + x)$  use coefficients 1, 2, 2 which give the mathematically equivalent  $\frac{1+2x}{2}$ .

As an example, consider the conversion of centimetres to metres:

```
<COMPU-METHOD>
  <SHORT-NAME>compuMethod_centimetres</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR>
            <V>0</V>
            <V>1</V>
          </COMPU-NUMERATOR>
          <COMPU-DENOMINATOR>
            <V>100</V>
          </COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

The above XML expresses the relationship  $metres(cm) = \frac{0+(1 \times cm)}{100}$ , or more simply,  $metres(cm) = \frac{cm}{100}$ .

Alternatively, the relationship can be expressed in the “physical-to-internal” direction:

```
<COMPU-METHOD>
  <SHORT-NAME>compuMethod_centimetres</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
```

```
<COMPU-PHYS-TO-INTERNAL>
  <COMPU-SCALES>
    <COMPU-SCALE>
      <COMPU-RATIONAL-COEFFS>
        <COMPU-NUMERATOR>
          <V>0</V>
          <V>100</V>
        </COMPU-NUMERATOR>
        <COMPU-DENOMINATOR>
          <V>1</V>
        </COMPU-DENOMINATOR>
      </COMPU-RATIONAL-COEFFS>
    </COMPU-SCALE>
  </COMPU-SCALES>
</COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

In this case RTA-RTE will generate an “identical” conversion. That is, no conversion will take place.

## 5.6 Expressing Values in Physical Units or Numerical Representation

The initial values of `DataPrototypes` and the invalid values of `DataTypes` are expressed using various classes of `ValueSpecification`. Values for primitive types may be specified directly in the numerical representation required by the implementation (i.e. the value that will appear in the initializer in the generated C code) or, for `ApplicationPrimitiveDataTypes`, a physical value can be specified in a particular unit and the corresponding numerical representation will be computed.

A `NumericalValueSpecification` is used for a primitive value already in the necessary numerical representation. If used for an `ApplicationPrimitiveType`, the value is interpreted in the context of the mapped `ImplementationType`.

An `ApplicationValueSpecification` is used for a physical value in a particular unit and can be used only for `ApplicationPrimitiveTypes`. Data conversion occurs at generation time, from the given unit to the unit of the `ApplicationPrimitiveType`, then to the numerical representation in the mapped `ImplementationDataType`. The resulting value is used in the initializer in the generated C code.

For complex types, complex `ValueSpecifications` are used and aggregate further `ValueSpecifications` for the elements. The layout of the `ValueSpecifications` must match the layout of the `DataType`, with `RecordValueSpecifications` used for record data types and `ArrayValueSpecifications` used for array data types. The leaves, that are of primitive type, are given values using `NumericalValueSpecifications` or `ApplicationValueSpecifications` as described above.

Suppose that the `ApplicationPrimitiveDataType` `MyCMAArrayType` from the earlier example is mapped to an `ImplementationDataType` that is an array of four 16-bit unsigned integers. The following example shows an initializer for an array of four lengths, some specified in metres and some in centimetres, stored as 16-bit unsigned integers

giving the number of centimetres.

```
<ARRAY-VALUE-SPECIFICATION>
  <ELEMENTS>
    <NUMERICAL-VALUE-SPECIFICATION>
      <VALUE>17</VALUE>
    </NUMERICAL-VALUE-SPECIFICATION>
    <NUMERICAL-VALUE-SPECIFICATION>
      <VALUE>0</VALUE>
    </NUMERICAL-VALUE-SPECIFICATION>
    <APPLICATION-VALUE-SPECIFICATION>
      <CATEGORY>VALUE</CATEGORY>
      <SW-VALUE-CONT>
        <UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
        <SW-VALUES-PHYS>
          <V>5</V>
        </SW-VALUES-PHYS>
      </SW-VALUE-CONT>
    </APPLICATION-VALUE-SPECIFICATION>
    <APPLICATION-VALUE-SPECIFICATION>
      <CATEGORY>VALUE</CATEGORY>
      <SW-VALUE-CONT>
        <UNIT-REF DEST='UNIT'>/package/centimetres</UNIT-REF>
        <SW-VALUES-PHYS>
          <V>145</V>
        </SW-VALUES-PHYS>
      </SW-VALUE-CONT>
    </APPLICATION-VALUE-SPECIFICATION>
  </ELEMENTS>
</ARRAY-VALUE-SPECIFICATION>
```

The first two array elements are initialized with the numerical representation and the values given are used directly in the generated C code. The third element is given the value  $5\text{ m}$  which is converted to  $500\text{ cm}$  and the numerical representation is 500. The fourth element is given the value  $145\text{ cm}$  with numerical representation 145.

### 5.6.1 Reusing Values

As well as specifying initial and invalid values directly where they are needed, it is also possible to create free-standing ConstantSpecifications that can be referenced multiple times in the model, using a ConstantReference in place of a ValueSpecification.

The following example shows a reusable value that could be used as the initial value for several DataPrototypes of type MyPointType (from an earlier example) where the point at two metres by three is the desired initial point:

```
<CONSTANT-SPECIFICATION>
  <SHORT-NAME>MyInitialPoint</SHORT-NAME>
  <VALUE-SPEC>
    <RECORD-VALUE-SPECIFICATION>
      <FIELDS>
```

```
<APPLICATION-VALUE-SPECIFICATION>
  <CATEGORY>VALUE</CATEGORY>
  <SW-VALUE-CONT>
    <UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
    <SW-VALUES-PHYS>
      <V>2</V>
    </SW-VALUES-PHYS>
  </SW-VALUE-CONT>
</APPLICATION-VALUE-SPECIFICATION>
<APPLICATION-VALUE-SPECIFICATION>
  <CATEGORY>VALUE</CATEGORY>
  <SW-VALUE-CONT>
    <UNIT-REF DEST='UNIT'>/package/metres</UNIT-REF>
    <SW-VALUES-PHYS>
      <V>3</V>
    </SW-VALUES-PHYS>
  </SW-VALUE-CONT>
</APPLICATION-VALUE-SPECIFICATION>
</FIELDS>
</RECORD-VALUE-SPECIFICATION>
</VALUE-SPEC>
</CONSTANT-SPECIFICATION>
```

This can be referenced as follows, either for a DataPrototype in its entirety or for an element of a complex type by placing within a complex ValueSpecification:

```
<CONSTANT-REFERENCE>
  <CONSTANT-REF DEST='CONSTANT-SPECIFICATION'>/package/MyInitialPoint</
  CONSTANT-REF>
</CONSTANT-REFERENCE>
```

Since ValueSpecifications are not directly tied to data types it is even possible to reuse ConstantSpecifications for different types, provided they are compatible. For example, a numerical value of 7 can be used to initialize a signed 8-bit integer type and an unsigned 16-bit integer type, but a value of 1000 cannot be used to initialize a signed 8-bit integer type. Similarly a RecordValueSpecification containing the numerical values 1 and 2 can be used to initialize any record type with two elements of some integer type, regardless of the names of the elements, but cannot be used to initialize a record type with three elements, for example.

## 6 Interfaces

---

All ports of a software component (whether a provided or a required port) are typed by a specific interface. The definition of interfaces is considered in detail in this section.

Note that how the software component interacts with the interface is defined by the <INTERNAL-BEHAVIOR> element that references a software component. This is discussed in Chapter 8.

### 6.1 Sender-Receiver

---

The <SENDER-RECEIVER-INTERFACE> element is used to define a sender-receiver interface as follows:

```
<SENDER-RECEIVER-INTERFACE>
  <SHORT-NAME>SRInterface</SHORT-NAME>
  ...
  <DATA-ELEMENTS>
    ...
  </DATA-ELEMENTS>
</SENDER-RECEIVER-INTERFACE>
```

The name of the sender-receiver interface definition is given by the <SHORT-NAME>. The name is used within other elements that need to reference the interface type, for example a software component may specify that it uses sender-receiver interface SRInterface.

The short-name of a sender-receiver interface should be a valid C identifier.

A sender-receiver interface can be used to communicate data (using data element prototypes within the <DATA-ELEMENTS> element) or modes (using mode declaration group prototypes within the <MODE-GROUPS> element).

#### 6.1.1 Data Element Prototypes

---

Each sender-receiver interface can specify zero or more data elements that constitute the AUTOSAR signals communicated over the interface. Each data item defines a prototype of a specific type and can be a primitive data type, a RECORD or an ARRAY type. See Chapter 5 for details of defining data types.

The declaration of data elements within a sender-receiver interface definition has the following structure:

```
<VARIABLE-DATA-PROTOTYPE>
  <SHORT-NAME>Speed</SHORT-NAME>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-IMPL-POLICY>QUEUED</SW-IMPL-POLICY>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
  <TYPE-TREF>/types/SInt16</TYPE-TREF>
```

</VARIABLE-DATA-PROTOTYPE>

A data element is defined using either a <DATA-ELEMENT-PROTOTYPE> or <VARIABLE-DATA-PROTOTYPE> element and all elements must be defined within an encapsulating <DATA-ELEMENTS> element.

Each data element must specify:

- The <SHORT-NAME> that you will use to refer to the item.
- A <TYPE-TREF> reference to the type of the data item.
- The <SW-IMPL-POLICY> is specified for queued communication and omitted for un-queued.

Unqueued communication means that a newly received value over-writes the previous value of the data item. If a value is sent multiple times before it is received then the receiver can only access the last transmitted value.

Queued communication means that the sender-receiver interface queues arrivals of the data item on the receiver side.

## 6.2 Nv-Data

<NV-DATA-INTERFACE> elements are used to define an interface used by an Nv-block software component type (see Chapter 11). Each interface contains one or more Nv-Data elements as follows:

```
<NV-DATA-INTERFACE>  
  <SHORT-NAME>NVInterface</SHORT-NAME>  
  ...  
  <NV-DATAS>  
  ...  
</NV-DATAS>  
</NV-DATA-INTERFACE>
```

The name of the Nv-Data interface definition is given by the <SHORT-NAME>. The name is used within other elements that need to reference the interface type, for example an Nv-Block software component may specify that it uses Nv-Data interface NVInterface.

The short-name of a Nv-Data interface should be a valid C identifier.

### 6.2.1 Nv-Data Elements

Each Nv-Data interface can specify zero or more Nv-data elements that constitute the AUTOSAR signals communicated over the interface. Each Nv-data item defines a prototype of a specific type and can be a primitive data type, a RECORD or an ARRAY type.

The declaration of data elements within a Nv-Data interface definition has the following structure:

```
<VARIABLE-DATA-PROTOTYPE>  
  <SHORT-NAME>DTC</SHORT-NAME>  
  <TYPE-TREF>/types/SInt16</TYPE-TREF>  
</VARIABLE-DATA-PROTOTYPE>
```

An Nv-data element is defined using a <VARIABLE-DATA-PROTOTYPE> element and all elements must be defined within an encapsulating <NV-DATAS> element.

Each Nv-data element must specify:

- The <SHORT-NAME> that you will use to refer to the item.
- A <TYPE-TREF> reference to the type of the data item.

## 6.3 Mode-Switch

The <MODE-SWITCH-INTERFACE> element is used to define a mode-switch interface in a configuration as follows:

```
<MODE-SWITCH-INTERFACE>  
  <SHORT-NAME>SRInterface</SHORT-NAME>  
  ...  
  <MODE-GROUP>  
    ...  
  </MODE-GROUP>  
</MODE-SWITCH-INTERFACE>
```

The name of the mode-switch interface definition is given by the <SHORT-NAME>. The name is used within other elements that need to reference the interface type.

The short-name of a mode-switch interface should be a valid C identifier.



*A mode-switch interface can contain **exactly one** mode group.*

### 6.3.1 Mode Groups

Each mode-switch interface can specify **exactly one** mode group that defines application modes.

The declaration of each mode group prototype within a mode-switch interface definition has the following structure:

```
<MODE-GROUP>  
  <SHORT-NAME>Speed</SHORT-NAME>  
  <TYPE-TREF>/autosar/mg1</TYPE-TREF>  
</MODE-GROUP>
```

A mode group prototype is defined using the <MODE-GROUP> element. Each mode group prototype must specify:

- the <SHORT-NAME> that you will use to refer to the item.

- the <TYPE-TREF> reference to mode declaration group.

The declaration of mode group prototypes and the use of mode declaration prototypes within interfaces is considered in detail in Chapter 9.

## 6.4 Client-Server

---

The <CLIENT-SERVER-INTERFACE> element is used to define a client-server interface as follows:

```
<CLIENT-SERVER-INTERFACE>  
  <SHORT-NAME>CSInterface</SHORT-NAME>  
  <OPERATIONS>  
    ...  
  </OPERATIONS>  
</CLIENT-SERVER-INTERFACE>
```

A client-server interface is named using the <SHORT-NAME> element. The name is used within other elements that need to reference the interface type. The short-name of a client-server interface should be a valid C identifier.

A client-server interface consists of one or more operations defined using the <OPERATIONS> container element.

### 6.4.1 Operations

---

The <OPERATIONS> element encapsulates one or more <CLIENT-SERVER-OPERATION> elements each of which defines a single operation in the client-server interface.

```
<OPERATIONS>  
  <CLIENT-SERVER-OPERATION>  
    <SHORT-NAME>MaximumValue</SHORT-NAME>  
    <ARGUMENTS>  
      ...  
    </ARGUMENTS>  
  </CLIENT-SERVER-OPERATION>  
</OPERATIONS>
```

Each operation is named using the <SHORT-NAME> element. The name you specify here will form part of the name used by the RTE to refer to the operation in your code.

The <ARGUMENTS> element encapsulates one or more argument prototypes using <ARGUMENT-DATA-PROTOTYPE> elements that define each argument (parameter) of the operation.

Each argument prototype definition must define:

- the <SHORT-NAME> of the parameter.
- a <TYPE-TREF> reference to the type of the parameter. The referenced type must correspond to a defined type – see Chapter 5.



- the <DIRECTION> of the parameter as “IN” (read only), “OUT” (write only) or “IN-OUT” readable and writable by the component.

The following XML definition shows the arguments for our MaximumValue operation that we assume takes two parameters of type SInt16 and returns the maximum of the two parameters in an out parameter called result:

```
<ARGUMENTS>
  <ARGUMENT-DATA-PROTOTYPE>
    <SHORT-NAME>InputA</SHORT-NAME>
    <TYPE-TREF>/types/SInt16</TYPE-TREF>
    <DIRECTION>IN</DIRECTION>
  </ARGUMENT-DATA-PROTOTYPE>
  <ARGUMENT-DATA-PROTOTYPE>
    <SHORT-NAME>InputB</SHORT-NAME>
    <TYPE-TREF>/types/SInt16</TYPE-TREF>
    <DIRECTION>IN</DIRECTION>
  </ARGUMENT-DATA-PROTOTYPE>
  <ARGUMENT-DATA-PROTOTYPE>
    <SHORT-NAME>OutputMaximum</SHORT-NAME>
    <TYPE-TREF>/types/SInt16</TYPE-TREF>
    <DIRECTION>OUT</DIRECTION>
  </ARGUMENT-DATA-PROTOTYPE>
</ARGUMENTS>
```

## 6.5 Calibration

The <PARAMETER-INTERFACE> element is used to define a calibration interface as follows:

```
<PARAMETER-INTERFACE>
  <SHORT-NAME>if1</SHORT-NAME>
  <PARAMETERS>
    ...
  </PARAMETERS>
</PARAMETER-INTERFACE>
```

The name of the calibration interface definition is given by the <SHORT-NAME>. The name is used within other elements that need to reference the interface type.

The short-name of a calibration interface should be a valid C identifier.

### 6.5.1 Calibration Prototypes

Each calibration interface can specify zero or more calibration prototypes that constitute the AUTOSAR parameters communicated over the interface. Each calibration prototype defines a prototype of a specific type.

The declaration of calibration prototypes within a calibration interface definition has the following structure:

```
<PARAMETER-DATA-PROTOTYPE>
```

```
<SHORT-NAME>Speed</SHORT-NAME>  
<CATEGORY>VALUE</CATEGORY>  
<TYPE-TREF>/types/SInt16</TYPE-TREF>  
</PARAMETER-DATA-PROTOTYPE>
```

## 6.6 Trigger

---

A trigger interface is defined with the <TRIGGER-INTERFACE> element as follows:

```
<TRIGGER-INTERFACE>  
  <SHORT-NAME>ifSingle</SHORT-NAME>  
  <TRIGGERS>  
    ...  
  </TRIGGERS>  
</TRIGGER-INTERFACE>
```

Each trigger interface can specify one or more triggers. The declaration of triggers within a trigger interface definition has the following structure:

```
<TRIGGER>  
  <SHORT-NAME>Name</SHORT-NAME>  
</TRIGGER>
```

A trigger is defined using only its name; no other information is necessary.

## 7 Software Component Types

---

Each software component you want to use must have its component type declared in the RTE generator's configuration. The component type makes the component available for composition into a larger software system

An atomic software component type is defined using the <APPLICATION-SW-COMPONENT-TYPE> element:

```
<*-TYPE>
  <SHORT-NAME>AtomicSoftwareComponent</SHORT-NAME>
  <PORTS>
    ...
  </PORTS>
</*-TYPE>
```

Additional component types are also possible. For example, a sensor-actuator component types uses the <SENSOR-ACTUATOR-SW-COMPONENT-TYPE> element:

```
<SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
  <SHORT-NAME>SensorActuatorComponent</SHORT-NAME>
  <PORTS>
    ...
  </PORTS>
</SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
```

Whatever the component type it must be named using the <SHORT-NAME> element. The name must be system-wide unique and is used within other elements to reference the software component type.

The short-name of a software-component should be a valid C identifier.

### 7.0.1 Service Components

---

In order for a service to be used by application components, "service component" types must also be declared.

Service components are defined using the <SERVICE-SW-COMPONENT-TYPE> element:

```
<SERVICE-SW-COMPONENT-TYPE>
  <SHORT-NAME>ServiceComponent</SHORT-NAME>
  <PORTS>
    ...
  </PORTS>
</SERVICE-SW-COMPONENT-TYPE>
```

### 7.0.2 NVRAM

---

AUTOSAR makes NVRAM blocks visible to application components and to the AUTOSAR NVRAM manager through instances of "NV Block software component" types.

NV Block components are defined using the <NV-BLOCK-SW-COMPONENT-TYPE>:

```
<NV-BLOCK-SW-COMPONENT-TYPE>
```

```
<SHORT-NAME>nv1</SHORT-NAME>  
<PORTS>  
  ...  
</PORTS>  
</NV-BLOCK-SW-COMPONENT-TYPE>
```

Each Nv-Block software component declares one or more Nv-blocks. See Chapter 11 for more details.

## 7.1 Ports

The ports of a software component are defined within the `<PORTS>` element, for example:

```
<PORTS>  
  <*-PORT-PROTOTYPE>  
  ...  
  </*-PORT-PROTOTYPE>  
</PORTS>
```

Within the `<PORTS>` element, the `<P-PORT-PROTOTYPE>` and the `<R-PORT-PROTOTYPE>` elements are used to define provided and required ports respectively.

### 7.1.1 Provided Ports

A provided port within a software component type definition is named using the `<SHORT-NAME>` element. The name is used within other elements to reference the software component type.

The short-name of a provided port should be a valid C identifier.

Each provided port definition must specify the interface type over which it will communicate with other ports using the `<PROVIDED-INTERFACE-TREF>`:

```
<P-PORT-PROTOTYPE>  
  <SHORT-NAME>Sender</SHORT-NAME>  
  <PROVIDED-INTERFACE-TREF>/interfaces/SRInterface  
  </PROVIDED-INTERFACE-TREF>  
</P-PORT-PROTOTYPE>
```

This `<PROVIDED-INTERFACE-TREF>` element must identify the required interface – see Chapter 6.

### 7.1.2 Required Ports

The definition of a required port takes the same form as that of a provided port with the exception that the `<R-PORT-PROTOTYPE>` element is used.

A required port within a software component type definition must be named using the `<SHORT-NAME>` element. The name is used within other elements to reference the software component type.

The short-name of a required port should be a valid C identifier.

The required port definition must reference an interface definition defined using the <REQUIRED-INTERFACE-TREF> element:

```
<R-PORT-PROTOTYPE>
  <SHORT-NAME>Receiver</SHORT-NAME>
  <REQUIRED-INTERFACE-TREF>
    /interfaces/SRInterface
  </REQUIRED-INTERFACE-TREF>
</R-PORT-PROTOTYPE>
```

In the simplest scenario, the provided and required ports at the endpoints of a connection reference the same interface definition – this guarantees that they are compatible. However, it is also possible to connect a require port to a provide port whose Interface is a superset of the require port Interface, or to connect ports with Interfaces that are compatible except for differing names for the interface elements using PortInterfaceMappings (see section 13.3.4) to explicitly specify the connections and furthermore to connect data items that are not directly compatible via *Data Conversion* – see Section 13.5.

### 7.1.3 Service Components

All the ports of a service component must be characterized by interfaces with the service flag set:

```
<R-PORT-PROTOTYPE>
  <SHORT-NAME>ServiceReceiver</SHORT-NAME>
  <REQUIRED-INTERFACE-TREF>
    /interfaces/ServiceSRInterface
  </REQUIRED-INTERFACE-TREF>
</R-PORT-PROTOTYPE>
...
<SENDER-RECEIVER-INTERFACE>
  <SHORT-NAME>ServiceSRInterface</SHORT-NAME>
  ...
  <IS-SERVICE>true</IS-SERVICE>
</SENDER-RECEIVER-INTERFACE>
```

## 7.2 Communication Specifications

When a port prototype is created it can optionally define attributes that control how the port will behave. Provided port prototypes define attributes in <PROVIDED-COM-SPECS> element whereas required ports use a <REQUIRED-COM-SPECS> element.

```
<!-- Providing...-->
<P-PORT-PROTOTYPE>
  <SHORT-NAME>name</SHORT-NAME>
  <PROVIDED-COM-SPECS>
    ...
  </PROVIDED-COM-SPECS>
  <PROVIDED-INTERFACE-TREF ...
```

```

</P-PORT-PROTOTYPE>

<!-- Requiring...-->
<R-PORT-PROTOTYPE>
  <SHORT-NAME>name</SHORT-NAME>
  <REQUIRED-COM-SPECS>
    ...
  </REQUIRED-COM-SPECS>
  <REQUIRED-INTERFACE-TREF ...
</R-PORT-PROTOTYPE>

```

### 7.2.1 Sender-Receiver

#### Acknowledgment Request

A sender (the provider of a sender-receiver interface) can optionally request that receivers acknowledge the transmission for one or more of the data elements communicated on the interface. The configuration depends on whether the data element is queued or not.

```

<P-PORT-PROTOTYPE>
  <SHORT-NAME>pa</SHORT-NAME>
  <PROVIDED-COM-SPECS>
    <!-- Enable TxAck -->
    <UNQUEUED-SENDER-COM-SPEC>
      <DATA-ELEMENT-IREF>
        <P-PORT-PROTOTYPE-REF ...
        <DATA-ELEMENT-PROTOTYPE-REF ...
      </DATA-ELEMENT-IREF>
      <TRANSMISSION-ACKNOWLEDGE>
        <TIMEOUT>0.5</TIMEOUT>
      </TRANSMISSION-ACKNOWLEDGE>
    </UNQUEUED-SENDER-COM-SPEC>
  </PROVIDED-COM-SPECS>
  <PROVIDED-INTERFACE-TREF ...
</P-PORT-PROTOTYPE>

```

Acknowledgment is enabled if the timeout is greater than zero.

For a sender, the only acknowledgment that can be requested is on transmission. It is not possible to request acknowledgement on reception by the receiver since there may be multiple receivers of the data item and it would be impossible to determine what the semantics of successful reception were (would it be one receiver has received, all receivers have received, some receivers have received etc.)

When a software component sends data over a port and acknowledgment requests are defined then your application code can request transmission acknowledgment. This is done by requesting feedback from the RTE with the following API call:

```

Rte_StatusType
Rte_Feedback_<Port>_<Data/Event>(Rte_Instance Self)

```

The call can be made irrespective of whether the receiving software component is located locally on the same ECU as the sender or on a remote ECU.

The call will block for <TIMEOUT> seconds if a <WAIT-POINT> in the calling runnable entity references a <DATA-SEND-COMPLETED-EVENT>, otherwise it will not block.

The following code shows how to request feedback for a transmission assuming that a blocking variant is configured:

```
FUNC(void, RTE_APPL_CODE)
Runnable(Rte_Instance self)
{
    Rte_StatusType status;

    status = Rte_Send_Port_Value(self, 42);
    if (status != RTE_E_OK)
    {
        // Handle error
    }
    else
    {
        status = Rte_Feedback_Port_Value(self);
        // Call blocks for up to TIMEOUT seconds
        switch (status) {
            case RTE_E_TRANSMIT_ACK:
                // COM service transmitted data...
            case RTE_E_TIMEOUT:
                // Transmission request timedout
        }
    }
}
```



*Rte\_Feedback applies only to writes/sends that use the “explicit” form of the sender-receiver communication API. Furthermore, when multiple transmissions are made then it is not possible to determine which is acknowledged first. You should serialize calls and check each individually if you need this behavior.*

There is also an additional form of the feedback API, Rte\_IFeedback, that applies to the “implicit” form of sender-receiver communication. This API will never block.

#### Initial Values

Non-queued data elements can optionally specify whether they have an initial value that is used when a receiver attempts a receive operation on a software component that has not yet made its first send or when the sender has invalidated the signal.

```
<R-PORT-PROTOTYPE>
<SHORT-NAME>ra</SHORT-NAME>
```

```
<REQUIRED-COM-SPECS>
  <NONQUEUED-RECEIVER-COM-SPEC>
    <DATA-ELEMENT-REF>...</DATA-ELEMENT-REF>
    <INIT-VALUE>
      <CONSTANT-REFERENCE>
        <CONSTANT-REF>/pkg/constant_857</CONSTANT-REF>
      </CONSTANT-REFERENCE>
    </INIT-VALUE>
  </NONQUEUED-RECEIVER-COM-SPEC>
</REQUIRED-COM-SPECS>
<REQUIRED-INTERFACE-TREF ...
</R-PORT-PROTOTYPE>
```

## Invalidation

Non-queued data elements can optionally enable invalidation so that a sender can send a data item that has an invalidated value.

Invalidation is enabled through the interface's invalidation policy element. This ensures that all uses of the interface have the same invalidation state.

```
<INVALIDATION-POLICYS>
  <INVALIDATION-POLICY>
    <DATA-ELEMENT-REF>/pkg/if/element</DATA-ELEMENT-REF>
    <HANDLE-INVALID>KEEP</HANDLE-INVALID>
  </INVALIDATION-POLICY>
  ...
</INVALIDATION-POLICYS>
```

Invalidation is used to indicate whether a data item is invalid on transmission. When the attribute is set at a sender, RTE API calls generated for that can be used by the sender to set a signal value as invalid. There are two types of invalidation:

- Implicit communication invalidation using the Rte\_IInvalidate API.
- Explicit Communication Invalidation using the Rte\_Invalidate API.

Setting invalidity can only be made at the point of transmission – once the data item is sent, the invalidate attribute cannot be modified. The receiver of an invalidated value receives the specified initial value.

## Queued Data Buffering

Recall that the RTE allows data to be queued and this is specified on the interface. Data is always queued on the receiver software component. To avoid unnecessary use of buffer space, the number of elements queued can be configured.

```
<R-PORT-PROTOTYPE>
  <SHORT-NAME>ra</SHORT-NAME>
  <REQUIRED-COM-SPECS>
    <QUEUED-RECEIVER-COM-SPEC>
```



```
<DATA-ELEMENT-REF>...</DATA-ELEMENT-REF>  
<QUEUE-LENGTH>3</QUEUE-LENGTH>  
</QUEUED-RECEIVER-COM-SPEC>  
</REQUIRED-COM-SPECS>  
<REQUIRED-INTERFACE-TREF ...  
</R-PORT-PROTOTYPE>
```

Only data elements that are specified as <IS-QUEUED> can be buffered.

### Alive Timeout

---

When communication is between ECUs it can be useful to know whether a sender is still alive and sending data. This is configured using an alive timeout that specifies the maximum time between receives that the receiver considers to indicate the sender is not alive.

If the alive timeout is exceeded then the receiver is informed through a <DATA-RECEIVE-ERROR-EVENT>. Therefore, the internal behavior of a software component that wishes to use the alive timeout must ensure that a runnable entity that responds to a <DATA-RECEIVE-ERROR-EVENT> is configured.

The alive timeout is specified in seconds.

```
<REQUIRED-COM-SPECS>  
  <NONQUEUED-RECEIVER-COM-SPEC>  
    <DATA-ELEMENT-REF>...</DATA-ELEMENT-REF>  
    <ALIVE-TIMEOUT>1.5</ALIVE-TIMEOUT>  
  </NONQUEUED-RECEIVER-COM-SPEC>  
</REQUIRED-COM-SPECS>
```

### Filter

---

In some cases a receiver may only want to receive a value when the sent data has a specific property, for example the value has changed. This is especially useful for reducing the load in systems where runnable entities are triggered by <DATA-RECEIVED-EVENTS> since filtering can reduce the number of events that get passed to the receiver and therefore the amount of time consumed processing signals that are not deemed useful.

```
<NONQUEUED-RECEIVER-COM-SPEC>  
  <DATA-ELEMENT-REF>...</DATA-ELEMENT-REF>  
  <FILTER>...</FILTER>  
</NONQUEUED-RECEIVER-COM-SPEC>
```

The valid filter specifications are described in the *RTA-RTE Reference Manual*.

#### 7.2.2 Client-Server

---

Buffering for the server side of client-server communication is configured using the <SERVER-COM-SPEC> element.

```
<P-PORT-PROTOTYPE>
```

```
<SHORT-NAME>Server</SHORT-NAME>
<PROVIDED-COM-SPECS>
  <SERVER-COM-SPEC>
    <OPERATION-REF>...</OPERATION-REF>
    <QUEUE-LENGTH>2</QUEUE-LENGTH>
  </SERVER-COM-SPEC>
</PROVIDED-COM-SPECS>
<PROVIDED-INTERFACE-TREF ...
</P-PORT-PROTOTYPE>
```

### 7.2.3 Client Component Instance Attributes

---

This release of RTA-RTE does not use the <CLIENT-COM-SPEC>.

## 8 Internal Behavior

---

In Chapter 6 you saw how to define the external interface of a software component. In this chapter you will see how to configure the internal behavior of the component.

Internal behavior elements are used to define how the software component will interact with the RTE at runtime.

The internal behavior of a software component specifies many aspects of how it behaves, including:

- The runnable entities that belong to the software component and how they interact (if at all) with the ports of the software component
- The events that cause runnable entities to be activated at runtime.
- The exclusive areas that exist so runnable entities can execute all or part of their code in mutual exclusion from other runnable entities.
- Variables that belong to the software component that are accessed by multiple runnable entities.
- Port argument lists for accessing AUTOSAR Basic Software Modules.

The following example illustrates the internal behavior description for software component “SoftwareComponent”:

```
<INTERNAL-BEHAVIOR>
  <SHORT-NAME>behavior</SHORT-NAME>
  <COMPONENT-REF>SoftwareComponent</COMPONENT-REF>

  <!-- Events that cause runnable activation -->
  <EVENTS>
    ...
  </EVENTS>

  <!-- Specification of mutual exclusion areas -->
  <EXCLUSIVE-AREAS>
    ...
  </EXCLUSIVE-AREAS>

  <!-- Variables shared by multiple runnable entities -->
  <INTER-RUNNABLE-VARIABLES>
    ...
  </INTER-RUNNABLE-VARIABLES>

  <!-- Variables unique to SW-C instance -->
  <PER-INSTANCE-MEMORYS>
    ...
  </PER-INSTANCE-MEMORYS>

  <!-- Arguments for accessing BSW modules -->
```

```
<PORT-ARGUMENT-LISTS>
...
</PORT-ARGUMENT-LISTS>

<!-- Runnable entities -->
<runnable entities>
...
</runnable entities>

<!-- Flag; if true SW-C can exist more than once -->
<SUPPORTS-MULTIPLE-INSTANTIATION>
...
</SUPPORTS-MULTIPLE-INSTANTIATION>
</INTERNAL-BEHAVIOR>
```

An internal behavior must be named using the <SHORT-NAME> element. The name is used within other elements to reference the behavior.

The short-name of an internal behavior does not need to be a valid C identifier (but it must pass the syntactic checks enforced by the XML Schema).

The way in which the configuration of RTE events within the <EVENTS> element and runnable entities defines the receive mode communication attributes is shown in Section 8.5. The following sections first outline the basic framework for RTE events and runnable entities before showing how to configure the RTE to achieve different types of runnable entity/interface interaction.

## 8.1 RTE Events

Events control how runnable entities are triggered by the generated RTE at runtime. These events are termed *RTE Events* to distinguish them from *OS events*.

RTA-RTE supports the following RTE Event types:

- **TIMING-EVENT** – activate a runnable entity periodically. The **TIMING-EVENT** allows you to execute a runnable entity to poll an RPort to check if data has been received, periodically call a server (ie be a client), periodically send data on a PPort or simply to execute some internal software component functionality. Runnable entities that are activated in response to a timing event are said to be “time-triggered”.
- **DATA-RECEIVED-EVENT** – activate a runnable entity when data is received on a RPort/data element. Runnable entities that are activated in response to a data received event are “event-triggered”.
- **DATA-RECEIVE-ERROR-EVENT** – activate a runnable entity when either invalidated data is received or an “alive timeout” has failed.
- **ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT** – activate a runnable entity when the result from an earlier call to a server is available.

- OPERATION-INVOKED-EVENT – activate a runnable entity to handle a server call for an operation on a provided port characterized by a client-server interface.
- DATA-SEND-COMPLETED-EVENT – activate a runnable entity when a transmission acknowledgement is available (explicit communication).
- DATA-WRITE-COMPLETED-EVENT – activate a runnable entity when a transmission acknowledgement is available (implicit communication).
- MODE-SWITCH-EVENT – activate a runnable entity on either entry to, or exit from an application mode. AUTOSAR also permits mode switch events to be triggered on the transition between two application modes.
- MODE-SWITCHED-ACK-EVENT – activate a runnable when a mode switch transition is complete.
- BACKGROUND-EVENT – activate a runnable within the “idle” task.
- INTERNAL-TRIGGER-OCCURRED-EVENT – activate a runnable within the SW-C, possibly by direct function call, without passing data.
- EXTERNAL-TRIGGER-OCCURRED-EVENT – activate a runnable within the SW-C by an external entity without passing data.

A <TIMING-EVENT> is used to indicate that a runnable entity will be activated periodically by the Operating System. The RTE generator will use this information to generate an appropriate schedule table that must be ticked from application code.

The <DATA-RECEIVED-EVENT> and <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT> elements implicitly provide the XML configuration of the RECEIVE\_MODE VFB communication attribute.

The structure for specifying RTE events is:

```
<EVENTS>
  <*-EVENT>
    <!-- Elements for event/runnable relationship -->
  </*-EVENT>
</EVENTS>
```

The <EVENTS> element can encapsulate any number of RTE events. RTE events can be listed in any order.

An RTE event can be used to activate a runnable entity when the event occurs. The <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>, <DATA-RECEIVED-EVENT> and <DATA-SEND-COMPLETED-EVENT> can also be waited upon by a runnable entity.

An RTE event references the runnable entity that is to be activated when the event occurs. Similarly, when a runnable entity waits on one of the RTE events, the runnable entity must reference the event(s) upon which it waits. Each of the RTE events can be used in both ways and by more than one runnable entity but it is not possible for the same event to simultaneously activate a runnable entity and to be waited upon.

## 8.2 Runnable Entities

Each software component will require at least one runnable entity. All runnable entities must be defined within the runnable entities definition in a <SWC-INTERNAL-BEHAVIOR> element.

```
<runnable entities>
  <RUNNABLE-ENTITY>
    <SHORT-NAME>RunnableEntity</SHORT-NAME>
    <!-- Elements defining behavior of entity -->
    <SYMBOL>CFunctionName</SYMBOL>
  </RUNNABLE-ENTITY>
</runnable entities>
```

A <RUNNABLE-ENTITY> must be named using the <SHORT-NAME> element. The name is used within other elements to reference the runnable entity.

The <SHORT-NAME> denotes the name of the runnable entity in the XML namespace, but it does not tell the RTE what the associated function body you will provide in your code is called. This information is provided by the <SYMBOL> declaration that links the runnable entity to the C function name you will use in your implementation. The <SYMBOL> element defines the name of the function that implements the runnable entity. The <SYMBOL> name must be a valid C identifier.

This declaration is sufficient if your runnable entity does not need to interact with the software component's ports. However, if a runnable entity needs to interact then you will need to specify additional information that allows the RTE generator to generate APIs to allow interaction to take place, for example:

1. What data items the runnable entity can send.
2. What data items the runnable entity can receive.
3. What mode instances the runnable entity can switch.
4. What RTE events the runnable entity waits upon.
5. Which servers the runnable entity calls and how it expects the result to be returned.

You can use the same runnable entity to receive data on one interface and send data on another interface, or to receive data on a port and then call a server port to process the received data. For example, you may create a runnable entity that reads an integer value from an RPort, multiplies it by two and sends it out on a PPort.

### 8.2.1 Specifying Maximum Arrival Rate

AUTOSAR permits the specification of the minimum time between executions by specifying a minimum start interval to place a limit on the rate at which activations can occur. RTA-RTE manages runnable activation to ensure that the minimum start interval is not violated.

A minimum start interval cannot be specified for a runnable entity that is invoked by an `OperationInvokedEvent`.



The RTE API function `Rte_MainFunction` must be invoked when minimum start intervals are used. The period at which the function will be invoked is specified by a command-line option.

## 8.3 Responding to Periodic Events

If your software component includes runnable entities that are not triggered by communication through the ports of the component you need to make sure that the runnable entities are executed at runtime. This is achieved by associating them with a `<TIMING-EVENT>` or `<BACKGROUND-EVENT>` that specifies how the runnable entity should execute.

### 8.3.1 Timing events

Each `<TIMING-EVENT>` element specifies the `<PERIOD>` of occurrence in seconds and must reference a runnable entity defined in the component's internal behavior using a `<START-ON-EVENT-REF>` element. A period of zero is illegal.

The runnable entity declaration itself needs only to declare a name for the runnable entity.

The following example shows how to configure the RTE to activate a runnable entity every second.

```
<EVENTS>
  <TIMING-EVENT>
    <SHORT-NAME>EverySecond</SHORT-NAME>
    <!-- Runnable to activate -->
    <START-ON-EVENT-REF>
      /pkg/ib/RunnableEntity
    </START-ON-EVENT-REF>
    <PERIOD>1.0</PERIOD> <!-- Every second -->
  </TIMING-EVENT>
</EVENTS>

<RUNNABLE-ENTITY>
  <SHORT-NAME>RunnableEntity</SHORT-NAME>
  <SYMBOL>MyFunctionName</SYMBOL>
</RUNNABLE-ENTITY>
```

A timing event must be named using the `<SHORT-NAME>` element. The name is used within other elements to reference the timing event. The short-name of a timing event does not need to be a valid C identifier.

### 8.3.2 Background events

RTA-RTE also supports *Background* events. These events are similar to `TimingEvents` in that they are recurring but unlike timing events have no fixed period.

RTA-RTE supports two mechanisms for activating background events:

- Via invocation from an “idle” task.
- Via an external OsAlarm or schedule table (see Section 24.1).

When all events mapped to a task are background events and no event has an externally declared OsAlarm or OsScheduleTable reference then RTA-RTE generates a task body that uses ChainTask instead of TerminateTask.



*In accordance with AUTOSAR requirements, RTA-RTE rejects configurations where a background triggered event is not triggered by an external OsAlarm or schedule table and is not mapped to the lowest priority task in the system (the “idle” task).*

## 8.4 Sending to a Port

If your software component provides a sender-receiver interface then you should define at least one runnable entity that sends data over the interface.

The runnable can send data in two ways:

**Explicitly** – The RTE generates an explicit API call (that may be optimized to a macro). The sent data item may be either queued or unqueued by the receiver.

**Implicitly** – The RTE generates an implicit API call that will be optimized to a macro. The sent data item must not be queued.

### 8.4.1 Explicit Communication

Runnable entities that send data must define a <DATA-SEND-POINT> that each of which specifies the data item that will be sent for a given interface.

The <DATA-SEND-POINT> specifies the port and data element using <P-PORT-PROTOTYPE-REF> and <DATA-ELEMENT-PROTOTYPE-REF> elements:

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>Sender</SHORT-NAME>
  <DATA-SEND-POINTS>
    <DATA-SEND-POINT>
      <SHORT-NAME>SendPoint</SHORT-NAME>
      <DATA-ELEMENT-IREF>
        <P-PORT-PROTOTYPE-REF ...
        <DATA-ELEMENT-PROTOTYPE-REF ...
      </DATA-ELEMENT-IREF>
    </DATA-SEND-POINT>
  </DATA-SEND-POINTS>
</RUNNABLE-ENTITY>
```



The <DATA-SEND-POINT> specifies access to an AUTOSAR variable – the access declaration includes the port and data element using <PORT-PROTOTYPE-REF> and <TARGET-DATA-PROTOTYPE-REF> elements:

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>Sender</SHORT-NAME>
  <DATA-SEND-POINTS>
    <VARIABLE-ACCESS>
      <SHORT-NAME>dsp_p1_dil</SHORT-NAME>
      <ACCESSED-VARIABLE>
        <AUTOSAR-VARIABLE-IREF>
          <PORT-PROTOTYPE-REF ...
          <TARGET-DATA-PROTOTYPE-REF ...
        </AUTOSAR-VARIABLE-IREF>
      </ACCESSED-VARIABLE>
    </VARIABLE-ACCESS>
  </DATA-SEND-POINTS>
</RUNNABLE-ENTITY>
```

For senders, it does not matter how the runnable entity is triggered, so any EVENT can be used to activate the runnable entity.

A <DATA-SEND-POINT> must be named using the <SHORT-NAME> element. The name is used within other elements to reference the data send point. The short-name does not need to be a valid C identifier.

#### 8.4.2 Implicit Communication

Runnable entities can also communicate using implicit data read/write access. Such configuration is guaranteed to be implemented as a simple macro that accesses global storage defined in the RTE rather than through a C function call. Configuration is almost identical to the <DATA-SEND-POINT>.

The <DATA-SEND-POINT> specifies access to an AUTOSAR variable – the access declaration includes the port and data element using <PORT-PROTOTYPE-REF> and <TARGET-DATA-PROTOTYPE-REF> elements:

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>Sender</SHORT-NAME>
  <DATA-WRITE-ACCESS>
    <VARIABLE-ACCESS>
      <SHORT-NAME>dwa_p1_dil</SHORT-NAME>
      <ACCESSED-VARIABLE>
        <AUTOSAR-VARIABLE-IREF>
          <PORT-PROTOTYPE-REF ...
          <TARGET-DATA-PROTOTYPE-REF ...
        </AUTOSAR-VARIABLE-IREF>
      </ACCESSED-VARIABLE>
    </VARIABLE-ACCESS>
  </DATA-WRITE-ACCESS>
</RUNNABLE-ENTITY>
```

For senders, it does not matter how the runnable entity is triggered, so any EVENT can be used to activate the runnable entity.

## 8.5 Receiving from a Port

---

Similarly, if your software component requires a sender-receiver interface then you must define at least one runnable entity that receives data over the interface. Data can be received in three ways:

**Implicit data read access** — your runnable is activated by some RTE event, e.g. a TimingEvent, and makes an RTE API call to read data

**Explicit Data read access** — your runnable entity is activated by an event and makes an RTE API call to read/receive the data. The receiver uses a non-blocking API to poll for the data.

Prior to R4.0 data was read via an OUT parameter of the generated API call. R4.0 introduced an additional form that performs the read using the generated APIs return value.

**Activation of Runnable Entity** — your runnable entity is activated when data is received on the software component's port and makes a non-blocking RTE API call to read/receive the data.

**Wake up of Wake Point** — your runnable entity makes a call to the RTE API to receive data. If the data has not yet been received the call will block and your runnable entity will continue to execute only after the event occurs or a configured timeout occurs, whichever happens first. When using this receive mode the invoking runnable entity may block and therefore must be a Category 2 runnable entity.

These receive modes are configured by defining specific combinations of runnable entity elements and RTE events. The required configurations for each receive mode are shown in the sections below.

### 8.5.1 Explicit Data Read Access

---

Runnable entities that are required to receive data with the "data read access" receive mode must define a <DATA-RECEIVE-POINT-BY-VALUES> element that specifies the data item they will receive.

The <DATA-RECEIVE-POINT-BY-ARGUMENTS> element specifies access to an AUTOSAR variable – the access declaration includes the port and data element using <PORT-PROTOTYPE-REF> and <TARGET-DATA-PROTOTYPE-REF> elements:

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>Receiver_DataReadAccess</SHORT-NAME>
  <DATA-RECEIVE-POINT-BY-ARGUMENTS>
    <VARIABLE-ACCESS>
      <SHORT-NAME>drp_r1_dil</SHORT-NAME>
      <ACCESSED-VARIABLE>
```

```

    <AUTOSAR-VARIABLE-IREF>
      <PORT-PROTOTYPE-REF ...
      <TARGET-DATA-PROTOTYPE-REF ...
    </AUTOSAR-VARIABLE-IREF>
  </ACCESSED-VARIABLE>
</VARIABLE-ACCESS>
</DATA-RECEIVE-POINT-BY-ARGUMENTS>
</RUNNABLE-ENTITY>

```

Using data read access implies that the runnable entity is polling the RPort for the specified data item. It is common, therefore, that a runnable entity which defines a data receive point will be activated by a <TIMING-EVENT> that specifies the required polling period.

A data receive point element must be named using the <SHORT-NAME> element. The name is used within other elements to reference the data receive point. The short-name does not need to be a valid C identifier.

#### Direct Read

AUTOSAR also supports reading data via the generated API's return value rather than via arguments. In this case an <DATA-RECEIVE-POINT-BY-VALUES> is used instead of the <DATA-RECEIVE-POINT-BY-ARGUMENTS> element.

As with the <DATA-RECEIVE-POINT-BY-ARGUMENTS> element the <DATA-RECEIVE-POINT-BY-VALUES> element specifies access to an AUTOSAR variables – the access declaration includes the port and data element using <PORT-PROTOTYPE-REF> and <TARGET-DATA-PROTOTYPE-REF> elements:

```

<DATA-RECEIVE-POINT-BY-VALUES>
  <VARIABLE-ACCESS>
    <SHORT-NAME>drv_R1_di2</SHORT-NAME>
    <ACCESSED-VARIABLE>
      <AUTOSAR-VARIABLE-IREF>
        <PORT-PROTOTYPE-REF ...
        <TARGET-DATA-PROTOTYPE-REF ...
      </AUTOSAR-VARIABLE-IREF>
    </ACCESSED-VARIABLE>
  </VARIABLE-ACCESS>
</DATA-RECEIVE-POINT-BY-VALUES>

```

#### 8.5.2 Implicit Data Read Access

Receivers may also use “implicit” data read access to access a receive port using implicit communication. The APIs generated for “implicit” access are guaranteed to be resolved to macros.

The <DATA-READ-ACCESS> element specifies access to one or more AUTOSAR variables – the access declaration includes the port and data element using <PORT-PROTOTYPE-REF> and <TARGET-DATA-PROTOTYPE-REF> elements:

```

<RUNNABLE-ENTITY>

```

```

<SHORT-NAME>Receiver_DataReadAccess</SHORT-NAME>
<DATA-READ-ACCESS>
  <VARIABLE-ACCESS>
    <SHORT-NAME>dra_r1_dil</SHORT-NAME>
    <ACCESSED-VARIABLE>
      <AUTOSAR-VARIABLE-IREF>
        <PORT-PROTOTYPE-REF ...
        <TARGET-DATA-PROTOTYPE-REF ...
      </AUTOSAR-VARIABLE-IREF>
    </ACCESSED-VARIABLE>
  </VARIABLE-ACCESS>
</DATA-READ-ACCESS>
</RUNNABLE-ENTITY>

```

### 8.5.3 Activation of Runnable Entity

If you require the receive mode to be “activation of runnable entity” then you must define a <DATA-RECEIVED-EVENT> as the RTE event that will activate the runnable entity when the data is received.

The following XML fragment shows how the “data read access” example above is converted to “activation of runnable entity”:

```

<EVENTS>
  <DATA-RECEIVED-EVENT>
    <SHORT-NAME>ActivationOfRunnableEntity</SHORT-NAME>
    <!-- Runnable to activate -->
    <START-ON-EVENT-REF>
      /pkg/internal/ReceivePort_DataItem
    </START-ON-EVENT-REF>
    <DATA-IREF>
      ...
    </DATA-IREF>
  </DATA-RECEIVED-EVENT>
</EVENTS>

<RUNNABLE-ENTITY>
  <SHORT-NAME>ReceivePort_DataItem</SHORT-NAME>
  <DATA-RECEIVE-POINTS>
    <!-- as above -->
  </DATA-RECEIVE-POINTS>
</RUNNABLE-ENTITY>

```

With this receive mode, the runnable entity still needs to define a data receive point (as above) to ensure that an API is created to read the data once the runnable entity has been activated by the RTE.

No <WAIT-POINT> should be specified since the combination of activated runnable entity and blocking API is forbidden by AUTOSAR.

#### 8.5.4 Wake up of Wait Point

If you need to receive with receive mode of “wake up of wait point” then your runnable entity must define the <WAIT-POINT> at configuration time. Any runnable entity that defines one or more wait points automatically becomes a Category2 runnable entity.

Before the runnable entity can wait you will need to make sure that the runnable entity is triggered initially. This can be done by defining another RTE event.

For example, if you want the runnable entity to run periodically but to wait during execution for data then you would use a <TIMING-EVENT> to trigger the periodic execution. Alternatively, you might want the runnable entity to be triggered by one data item and then wait for another data item to arrive. In this case you would use a <DATA-RECEIVED-EVENT> that includes a reference to the runnable entity to trigger the initial execution.

A <WAIT-POINT> must be named using the <SHORT-NAME> element. The name is used within other elements to reference the wait point. The short-name does not need to be a valid C identifier.

Each <WAIT-POINT> must specify the RTE events by which it can be triggered. This is done with the <TRIGGER-REFS> element. For a receiver, this reference must be to the EVENT called <DATA-RECEIVED-EVENT>.

Each <WAIT-POINT> must also specify a <TIMEOUT> which is the maximum time that the receiver will wait for wake-up.

The following XML fragment shows how the “activation of runnable entity” example above is converted to “wake up of wait point”:

```
<EVENTS>
  <DATA-RECEIVED-EVENT>
    <SHORT-NAME>WakeUpOfWaitPoint</SHORT-NAME>
    <!-- Note the runnable reference is removed -->
    <DATA-IREF>
      <!-- as above -->
    </DATA-IREF>
  </DATA-RECEIVED-EVENT>
</EVENTS>

<RUNNABLE-ENTITY>
  <SHORT-NAME>ReceivePort_DataItem</SHORT-NAME>
  <DATA-RECEIVE-POINTS>
    ... as above ...
  </DATA-RECEIVE-POINTS>
  <WAIT-POINTS>
    <WAIT-POINT>
      <SHORT-NAME>WaitPoint</SHORT-NAME>
      <TRIGGER-REFS>
        <TRIGGER-REF>
          /pkg/internal/WakeUpOfWaitPoint
        </TRIGGER-REF>
      </TRIGGER-REFS>
    </WAIT-POINT>
  </WAIT-POINTS>
</RUNNABLE-ENTITY>
```

```

        </TRIGGER-REFS>
        <TIMEOUT>0.5</TIMEOUT> <!-- 500ms -->
    </WAIT-POINT>
    ...
</WAIT-POINTS>
</RUNNABLE-ENTITY>

```

### 8.5.5 Accessing Calibration Parameters

When an RPort is categorized by a calibration interface (see Section 6.5) RTA-RTE can create APIs to access the calibration prototypes.

Prior to R4.0 an `Rte_CallPrm` API was created for each parameter in the RPort. This could lead to many unused, and undesired, APIs and therefore AUTOSAR R4.0 introduced the concept of a `ParameterAccess` point to control when and how runnable entities access a mode using the `Rte_Prm` API.

```

<RUNNABLE-ENTITY>
...
<PARAMETER-ACCESS>
  <PARAMETER-ACCESS>
    <SHORT-NAME>pap_rCall_cp_value1</SHORT-NAME>
    <ACCESSED-PARAMETER>
      <AUTOSAR-PARAMETER-IREF>
        <PORT-PROTOTYPE-REF ...
        <TARGET-DATA-PROTOTYPE-REF ...
      </AUTOSAR-PARAMETER-IREF>
    </ACCESSED-PARAMETER>
  </PARAMETER-ACCESS>
...

```

The `<PARAMETER-ACCESS>` element references the required calibration prototype using the `<PORT-PROTOTYPE-REF>` and `<TARGET-DATA-PROTOTYPE-REF>` elements. The referenced port should be declared within the context of the runnable entity's SWC-type.

A `ParameterAccess` point must be declared for each required `Rte_Prm` API.

### 8.6 Responding to a Server Request on a Port

If your software component provides a client-server interface then you must define a runnable entity for each operation in the interface. These runnable entities are the servers for the client-server PPorts on your software component.

Each PPort that is characterized by a client-server interface must provide a runnable entity for each operation defined by the interface.

For the runnable entity to be regarded by the RTE as a server it must be tied to an `<OPERATION-INVOKED-EVENT>`. This RTE event allows the RTE to call the runnable entity at runtime in response to client requests. The `<OPERATION-INVOKED-EVENT>` must specify what operation request on the server interface will result in the runnable entity being activated. The following example shows how to configure runnable "ServerOper-

ation”:

```

<EVENTS>
  <OPERATION-INVOKED-EVENT>
    <SHORT-NAME>ActivateServerRunnable</SHORT-NAME>
    <START-ON-EVENT-REF>
      /pkg/internal/ServerOperation
    </START-ON-EVENT-REF>
    <OPERATION-IREF>
      ...
    </OPERATION-IREF>
  </OPERATION-INVOKED-EVENT>
</EVENTS>

<RUNNABLE-ENTITY>
  <SHORT-NAME>ServerOperation</SHORT-NAME>
  <SYMBOL>...</SYMBOL>
</RUNNABLE-ENTITY>

```

An `<OPERATION-INVOKED-EVENT>` must be named using the `<SHORT-NAME>` element. The name is used within other elements to reference the event. The short-name does not need to be a valid C identifier.

### 8.6.1 Concurrent Invocation of Servers

When a runnable acting as a server is written to be re-entrant then the RTE can optimize invocation by clients on the same ECU to a direct function call. This means that no queuing is required (or possible) and therefore multiple invocations of the server can occur concurrently.

The RTE generator needs to know which runnable entities can be called in this way. Concurrent invocation is defined within the server’s runnable entity definition as follows:

```

<CAN-BE-INVOKED-CONCURRENTLY>
true
</CAN-BE-INVOKED-CONCURRENTLY>

```



*The runnable must be written to be re-entrant. If this is not the case then data consistency is not guaranteed when there is more than one client simultaneously requesting the server.*

### 8.7 Making a Client Request on a Port

Similarly, if your software component requires a client-server interface then you must define at least one runnable entity that acts as the client.

Clients can access servers either synchronously or asynchronously.

**Synchronous** —the client will be blocked while the server processes the request. When the server has processed the request the result is passed back to the client and the client continues to execute.

A synchronous client is specified using a `SynchronousServerCallPoint`.

**Asynchronous** —the client makes the request then continues to execute. When the server has serviced the request, the client can collect the result. The mechanism by which the result is collected by the client is configured statically at configuration time by configuring a VFB `RECEIVE_MODE`. When using asynchronous communication, a client may have at most one outstanding request of any one server (but may make multiple simultaneous requests to different servers).

An asynchronous client is specified using an `AsynchronousServerCallPoint`.

In the case of synchronous access, the client will block until the server completes and the result of the operation will be returned by the API call. In the case of asynchronous access, the client does not block and simply continues to execute. The client can then receive the server result in one of three receive modes:

**Data read access** —the client uses a non-blocking read API to poll for the server's result. If the receive mode attribute is not specified, RTA-RTE assumes data read access by default. This mode is only valid for asynchronous client-server communication.

If only an `AsynchronousServerCallPoint` is specified then RTA-RTE assumes data read access by default.

**Activation of runnable entity** —the client runnable entity is activated by RTA-RTE whenever new data is available. RTA-RTE creates a non-blocking read API to access the result. This mode is only valid for asynchronous client-server communication.

This receive mode is specified using a combination of an `AsynchronousServerCallPoint` an `AsynchronousServerCallReturnsEvent`.

**Wake up of wait point** —the client suspends execution via a blocking read API call until either a timeout occurs or the server's result is available. The client uses a blocking read API to read data. When using this receive mode the runnable entity may block and therefore must be a Category 2 runnable entity.

A client that communicates synchronously implies that the receive mode is "wake up of wake up point".

This receive mode is specified using a combination of an `AsynchronousServerCallPoint`, an `AsynchronousServerCallReturnsEvent` and a `WaitPoint`.

These receive modes are configured by defining specific combinations of runnable entity elements and `EVENTS`. The required configurations for each receive mode are shown in the sections below.

You can use the same runnable entity as a server on one interface and client on another interface. For example, you may create a runnable entity that handles a server request for sorting on a `PPort` and uses an auxiliary operation on an `RPort`.



### 8.7.1 Synchronous Clients

Runnable entities that need to call a server synchronously (i.e. block when handling an RPort characterized by a client-server interface) must define a `<SYNCHRONOUS-SERVER-CALL-POINT>`.

The `<SYNCHRONOUS-SERVER-CALL-POINT>` defines which operations the client can call and specifies a global `<TIMEOUT>` value for all the called operations. The `<TIMEOUT>` specifies the maximum time that the client will wait for any of the servers providing an operation.

A `<SYNCHRONOUS-SERVER-CALL-POINT>` must be named using the `SHORT-NAME` element. The name is used within other elements to reference the call point. The short-name does not need to be a valid C identifier but must pass the syntactic checks imposed by the AUTOSAR schema.

```

<RUNNABLE-ENTITY>
  <SHORT-NAME>SynchronousClient</SHORT-NAME>
  <SERVER-CALL-POINTS>
    <SYNCHRONOUS-SERVER-CALL-POINT>
      <SHORT-NAME>SyncCallPoint</SHORT-NAME>
      <OPERATION-IREFS>
        <OPERATION-IREF>
          ...
        </OPERATION-IREF>
      </OPERATION-IREFS>
    </SYNCHRONOUS-SERVER-CALL-POINT>
    <TIMEOUT>0.500</TIMEOUT> <!-- 500ms -->
    ...
  </SERVER-CALL-POINTS>
  <SYMBOL>...</SYMBOL>
</RUNNABLE-ENTITY>

```

When a client has a client mode of synchronous, this means that the client will be blocked until the server call returns. Implicitly this means the client is using a receive mode of “wake up of wait point”.

You will have to ensure that the client is triggered by an RTE event.

### 8.7.2 Asynchronous Clients

Runnable entities that need to call a server asynchronously (i.e. handle an RPort characterized by a client-server interface without blocking) must define an `<ASYNCHRONOUS-SERVER-CALL-POINT>`.

When a client is synchronous, it will be blocked while the server executes. However, when a client is asynchronous the client is not blocked while the server executes. Once the server has completed it is the client’s responsibility to get the result of the operation back from the server.

Server results are collected by clients using one of the same three receive modes used

for receivers in sender-receiver communication:

- data read access
- activation of runnable entity
- wake up of wait point

In all cases, the framework for the client request is the same and is shown in the following example where a runnable entity called Client is configured to make an asynchronous call on “CSInterface” to “Operation”:

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>AsynchronousClient</SHORT-NAME>
  <SERVER-CALL-POINTS>
    <ASYNCHRONOUS-SERVER-CALL-POINT>
      <SHORT-NAME>AsyncCallPoint<SHORT-NAME>
      <OPERATION-IREFS>
        <OPERATION-IREF>
          ...
        </OPERATION-IREF>
      </OPERATION-IREFS>
    </ASYNCHRONOUS-SERVER-CALL-POINT>
    ...
  </SERVER-CALL-POINTS>
  <SYMBOL>...</SYMBOL>
</RUNNABLE-ENTITY>
```

An `<ASYNCHRONOUS-SERVER-CALL-POINT>` must be named using the `<SHORT-NAME>` element. The name is used within other elements to reference the call point. The short-name does not need to be a valid C identifier but must pass the syntactic checks imposed by the AUTOSAR schema.

#### Collecting the Result by Data Read Access

If the client does not specify any `<WAIT-POINTS>` that reference an `<ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>` then the receive mode for the server results defaults to “data read access” and the RTE generator will provide a non-blocking `Rte_Result` API call for accessing the server result.

#### Collecting the Result by Activation of Runnable Entity

When you want the server result to be communicated back to the client software component through receive mode “activation of runnable entity” then, in addition to defining an `<ASYNCHRONOUS-SERVER-CALL-POINT>` in the client runnable entity, the following elements are required:

- a runnable entity to capture the result (this may or may not be the same runnable entity that makes the client request);
- an RTE event to trigger this runnable entity when the server completes.

The trigger is configured using the RTE event called <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT> that references both the client's server call point and the runnable entity to activate when the call returns:

```
<EVENTS>
  <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
    <SHORT-NAME>ServerCallReturnsEvent</SHORT-NAME>
    <START-ON-EVENT-REF>
      /pkg/internal/CollectResult
    </START-ON-EVENT-REF>
    <EVENT-SOURCE-REF>
      /pkg/internal/RE1/AsyncCallPoint
    </EVENT-SOURCE-REF>
  </ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
</EVENTS>

<RUNNABLE-ENTITY>
  <SHORT-NAME>CollectResult</SHORT-NAME>
  ...
</RUNNABLE-ENTITY>
```

Like data read access, a non-blocking Rte\_Result API call is generated to enable the activated runnable entity to collect the server result.

#### Collecting the Result by Wake up of Wake Point

When you want the server result to be communicated through receive mode "wake up of wait point" then, in addition to defining an ASYNCHRONOUS-SERVER-CALL-POINT in the client runnable entity the following additional configuration is needed:

- declaration of an <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT> that specifies for which call the RTE event will occur.
- a <WAIT-POINT> in the client runnable entity that waits on the <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>.

The following example shows the declaration of the RTE event and the additional configuration required by the client:


```
<EVENTS>
  <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
    <SHORT-NAME>ServerCallReturns</SHORT-NAME>
    <EVENT-SOURCE-REF>
      /pkg/internal/Client/AsyncCallPoint
    </EVENT-SOURCE-REF>
  </ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
</EVENTS>

<RUNNABLE-ENTITY>
  <SHORT-NAME>Client</SHORT-NAME>
  <!-- Server call point as before -->
```

```
<SERVER-CALL-POINTS>
  <ASYNCHRONOUS-SERVER-CALL-POINT>
    <SHORT-NAME>AsyncCallPoint</SHORT-NAME>
    <OPERATION-IREFS>
      ...
    </OPERATION-IREFS>
  </ASYNCHRONOUS-SERVER-CALL-POINT>
</SERVER-CALL-POINTS>

<!-- Additional WAIT-POINT configuration -->
<WAIT-POINTS>
  <WAIT-POINT>
    <SHORT-NAME>WaitForServerResult</SHORT-NAME>
    <TRIGGER-REFS>
      <TRIGGER-REF>
        /pkg/internal/ServerCallReturns
      </TRIGGER-REF>
    </TRIGGER-REFS>
    <TIMEOUT>0.5</TIMEOUT> <!-- 500ms -->
  </WAIT-POINT>
</WAIT-POINTS>
</RUNNABLE-ENTITY>
```

As with waiting on events in sender-receiver interface interaction, the `<WAIT-POINT>` must specify a `<TIMEOUT>` with the maximum allowed amount of time in seconds for which wait can occur.

 A timeout of 0 will cause an infinite wait.

A `<WAIT-POINT>` must be named using the `<SHORT-NAME>` element. The name is used within other elements to reference the wait point. The short-name does not need to be a valid C identifier.

## 8.8 Direct Trigger of a Runnable Entity

As we have seen, port based communication provides an effective mechanism for transmitting an “event” that combines both data and the notification that the event has occurred. However such a mechanism may be more than is required; a software component may want to transmit just the fact that the event has occurred and have no associated data. AUTOSAR R4.0 introduced the concept of *Trigger Events* to handle this situation.

### Internal Trigger

An *Internal Trigger Occurred Event* is used within a SW-C instance to activate one or more runnables when the event occurs. Since the event is internal to a component no ports are required and hence no connections are necessary during integration.

An internal trigger requires a *Triggering Point* to be declared within the runnable that will make the `Rte_IrTrigger` API call.

```
<RUNNABLE-ENTITY>
```

```
<SHORT-NAME>re_tel</SHORT-NAME>
...
<INTERNAL-TRIGGERING-POINTS>
  <INTERNAL-TRIGGERING-POINT>
    <SHORT-NAME>itp3</SHORT-NAME>
  </INTERNAL-TRIGGERING-POINT>
  ...
```

Then RTA-RTE will create Rte\_IrTrigger\_re1\_itp3 API based on the short name of the triggering point. The short name is also used to identify the internal trigger point as the event source within an *Internal Trigger Occurred Event* element:

```
<INTERNAL-TRIGGER-OCCURRED-EVENT>
  <SHORT-NAME>itp3</SHORT-NAME>
  <START-ON-EVENT-REF ...
  <EVENT-SOURCE-REF>/pkg/swc/ib/re_tel/itp3</EVENT-SOURCE-REF>
</INTERNAL-TRIGGER-OCCURRED-EVENT>
```

Multiple Internal Trigger Occurred events can reference the same triggering point and thus one internal trigger API can start multiple runnables. The runnables to be started are specified by in response to an internal triggering event are specified using one or more *Internal Trigger Occurred Event* elements.

When a SW-C indicates the event has occurred by invoking the Rte\_IrTrigger API created for the internal triggering point RTA-RTE activates all runnables specified by Internal Trigger Occurred events that reference the triggering point. For runnables **without** a minimum start interval and **without** a task referenced in the event's RteEventMapping this activation is by a direct function call.

## External Trigger

In contrast to internal triggering, *External Trigger Occurred Event* is used by entities external to a SW-C instance to activate runnables. Since the event is external to a component ports categorized by a trigger interface are used hence must be connected during integration.

An external trigger requires a port categorized by a *Trigger Interface* (see Section 6.6) to be declared within the SW-C. For SW-Cs that indicate the event has occurred by invoking the Rte\_Trigger API call a provided port is necessary whereas SW-Cs that respond to the event use a required port.

```
<R-PORT-PROTOTYPE>
  <SHORT-NAME>r2</SHORT-NAME>
  <REQUIRED-INTERFACE-TREF>/pkg/itf</REQUIRED-INTERFACE-TREF>
</R-PORT-PROTOTYPE>
```

Where itf is a trigger interface, for example:

```
<TRIGGER-INTERFACE>
  <SHORT-NAME>itf</SHORT-NAME>
  <TRIGGERS>
```

```
<TRIGGER>
  <SHORT-NAME>trig1</SHORT-NAME>
</TRIGGER>
<TRIGGER>
  <SHORT-NAME>trig2</SHORT-NAME>
</TRIGGER>
</TRIGGERS>
</TRIGGER-INTERFACE>
```

The short name of the port and the short name of the trigger within the interface are used to form the name of the Rte\_Trigger API. Within a SW-C, multiple *External Trigger Occurred Events* can reference the same port/trigger and thus one external trigger API can start multiple runnables. The provided port can be connected to multiple ports in different SW-Cs.

```
<EXTERNAL-TRIGGER-OCCURRED-EVENT>
  <SHORT-NAME>etp2</SHORT-NAME>
  <START-ON-EVENT-REF>/pkg/swc/IB/re</START-ON-EVENT-REF>
  <TRIGGER-IREF>
    <CONTEXT-R-PORT-REF>/pkg/swc/rPort</CONTEXT-R-PORT-REF>
    <TARGET-TRIGGER-REF>/pkg/itf/trig1</TARGET-TRIGGER-REF>
  </TRIGGER-IREF>
</EXTERNAL-TRIGGER-OCCURRED-EVENT>
```

When a SW-C with a provide trigger port indicates the event has occurred by invoking the Rte\_Trigger API created for the external trigger RTA-RTE activates all runnables specified by connected External Trigger Occurred events that reference the triggering port/trigger. For runnables **without** a minimum start interval and **without** a task referenced in the event's RteEventMapping this activation is by a direct function call.

## 8.9 Exclusive Areas

Software components that need to provide mutual exclusion over data shared by two (or more) of their runnable entities do so by configuring exclusive areas

The RTE generator uses exclusive area configuration to create operating system configuration files and to optimize exclusive areas. For example, if the only components that access a region are mapped to the same task then the entire region can be elided.

Exclusive areas are defined in the XML configuration and are associated with the runnable entities that use them.

### 8.9.1 Configuration

Each <SWC-INTERNAL-BEHAVIOR> element must name the <EXCLUSIVE-AREAS> it uses. Note that this means that the scope of any exclusive areas that you define is the software component instance. It is not possible to define exclusive areas that cross software component boundaries<sup>1</sup>.

<sup>1</sup>Data that is shared between multiple software-component instances which can potentially be accessed concurrently should be encapsulated in its own component and then normal Sender-Receiver or Client-

Each exclusive-area defined within an internal behavior definition must be named using the <SHORT-NAME> element. The name is used within other elements to reference the software component type and to form the “handle” by which the exclusive area is accessed at run-time. The short-name of an exclusive area should be a valid C identifier.

```
<INTERNAL-BEHAVIOR>
  <EXCLUSIVE-AREAS>
    <EXCLUSIVE-AREA>
      <SHORT-NAME>ExclusiveArea</SHORT-NAME>
    </EXCLUSIVE-AREA>
    ...
  </EXCLUSIVE-AREAS>
</INTERNAL-BEHAVIOR>
```

RTA-RTE can be informed how to implement the exclusive area with an RteExclusiveAreaImplementation element within the ECUC description.



*You should create an exclusive area implementation element for each exclusive area. If omitted RTA-RTE defaults to “OS resource” implementation strategy.*

A different exclusive area implementation method can be set for each exclusive area and SWC instance.



*The InterruptBlocking method will cause all OS interrupts to be blocked in the worst case for the longest execution time of the protected critical section.*

## 8.9.2 Usage

Each runnable entity in your <INTERNAL-BEHAVIOR> section can declare if it uses one of your named exclusive areas and how it uses the area at runtime with the following elements:

- **Explicit access**—a <RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREA> element defines exclusive areas accessed using an explicit API. The area’s name forms part of the generated API (explicit access is similar to a standard resource in OSEK OS).
- **Implicit access**—a <RUNNABLE-ENTITY-RUNS-IN-EXCLUSIVE-AREA> element defines an exclusive area that is locked automatically before the runnable entity executes and is released when the runnable entity terminates or waits for an RTE event (similar to an internal resource in OSEK OS). In this case there is no user accessible way of explicitly entering and leaving the defined exclusive area.

The configuration of both types of the exclusive area is shown below:

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>NeedsExclusiveAreas</SHORT-NAME>
  <RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREAS>
```

Server communication used to access the data.

```
<RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREA>
  <EXCLUSIVE-AREA-REF>
    /pkg/internal/ExclusiveArea
  </EXCLUSIVE-AREA-REF>
</RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREA>
</RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREAS>
<RUNNABLE-ENTITY-RUNS-IN-EXCLUSIVE-AREAS>
  <RUNNABLE-ENTITY-RUNS-IN-EXCLUSIVE-AREA>
    <EXCLUSIVE-AREA-REF>
      /pkg/internal/AnotherExclusiveArea
    </EXCLUSIVE-AREA-REF>
  </RUNNABLE-ENTITY-RUNS-IN-EXCLUSIVE-AREA>
</RUNNABLE-ENTITY-RUNS-IN-EXCLUSIVE-AREAS>
<RUNNABLE-ENTITY>
```

### 8.9.3 Implementation Methods

---

RTA-RTE supports the following implementation mechanisms for exclusive areas:

- OS resource locking.
- OS interrupt blocking.
- All interrupt blocking.
- Cooperative Runnable Placement.

#### OS Resource

---

When the *OS resource* implementation method is selected for an exclusive area instance RTA-RTE will attempt to use an OS resource is used to implement the exclusive area.

If an *exclusive area implementation* element explicitly references an *OsResource* in the input then RTA-RTE will use that resource unless access can be optimized. However if no *OsResource* is selected then RTA-RTE will **create** a suitable standard/internal resource. The naming of this resource is described in the *RTA-RTE Reference Manual*.

When using OS resources RTA-RTE will attempt to optimize the implementation of *Rte\_Enter* and *Rte\_Exit* APIs to try and eliminate use of the resource entirely. This optimization is possible when RTA-RTE observes that data consistency is guaranteed by other means, namely:

- All accessors are located in the same task, or
- All accessing tasks are non-preemptive, or
- All accessing tasks have the same priority, or
- All accessing tasks are declared in the input as accessing the same internal resource.



When an accessing runnable is declared as having implicit (“run inside”) access to the exclusive area RTA-RTE will insert resource access around runnable calls in the generated task body or use an internal resource if all accessors use implicit access and no other runnables are mapped to the relevant tasks.

### OS Interrupt Blocking

---

When the *OS interrupt* implementation method is selected for an exclusive area instance RTA-RTE will use suspend/resume of **OS interrupts** to implement the exclusive area.

When using OS interrupt blocking RTA-RTE optimizes the APIs to null access if data inconsistency cannot occur.



*Avoid using any RTE APIs when OS interrupts are blocked. Failure to adhere to this restriction may result in context switches not being recognized by the OS when interrupt handling is resumed.*

### All Interrupt Blocking

---

AUTOSAR allows an exclusive area to block all interrupts, not just OS interrupts, when the *All interrupt* implementation method is selected for an exclusive area instance.

When using all interrupt blocking RTA-RTE will optimize API calls to null access if data inconsistency cannot occur.



*Avoid using any RTE APIs when OS interrupts are blocked. Failure to adhere to this restriction may result in context switches not being recognized by the OS when interrupt handling is resumed.*

### Non-preemptive Tasks

---

When the *non-preemptive task* implementation method is selected for an exclusive area instance RTA-RTE assumes that data consistency is guaranteed by external means and that therefore all access to the exclusive area can be optimized to null access.

### Cooperative Runnable Placement

---

When the *cooperative runnable placement* implementation method is selected for an exclusive area instance RTA-RTE ensures that tasks containing runnables accessing an exclusive area with implementation method cooperative runnable placement cannot preempt other tasks with runnables accessing the same exclusive area instance when a runnable is active. Unlike the *non-preemptive tasks* implementation method preemption is allowed when runnables are not running, e.g. between runnables.

If required to implement data consistency and no OsResource is explicit selected within an EA implementation element in the ECUC then RTA-RTE will **create** a suitable standard/internal resource. The naming of this resource is described in the *RTA-RTE Reference Manual*.

When using cooperative runnable placement RTA-RTE will optimize the implementation, e.g. when all accessors are located in the same task or all accessing tasks are non-preemptive, to null access since data consistency is then assured by external means. The decision on whether to optimize is the same as for the *OS resource* implementation method.

The cooperative runnable placement is best used when the accessing runnables are declared as having implicit (“run inside”) access the exclusive area since RTA-RTE will then insert resource access around runnable calls in the generated task body or use internal resources as appropriate. If accessing runnables are declared as explicit access (“can enter”) the exclusive area, i.e. will use the `Rte_Enter` API, then RTA-RTE cannot use an internal resource to implement cooperative runnable placement since this would block access for the entire runnable.

## 8.10 Inter-Runnable Variables

---

The RTE allows the configuration of inter-runnable variables that provide a way for runnable entities to communicate between themselves. While this is possible through user code and shared variable access (protected by exclusive areas - see Section 8.9) this can be inefficient when handling primitive data types since the exclusive area API calls are typically mapped onto the underlying Operating System’s resource control mechanism.

Therefore, the RTE provides lightweight mechanisms for inter-runnable communication that is equivalent in semantics to implicit/explicit sender-receiver communication but within the scope of the software component instance.

### 8.10.1 Defining Variables

---

Each software component can define a number of variables at the scope of the component and how access is provided to the variable at runtime, either using explicit or implicit communication:

### 8.10.2 Declaring Read and Write Access

---

Each runnable must explicitly specify whether it reads or writes a variable at runtime. Access is declared using `<READ-LOCAL-VARIABLES>` and `<WRITTEN-LOCAL-VARIABLES>` elements that reference an inter-runnable variable defined within the SWC-type, for example:

```
<WRITTEN-LOCAL-VARIABLES>  
  <VARIABLE-ACCESS>  
    <SHORT-NAME>xx</SHORT-NAME>  
    <ACCESSED-VARIABLE>  
      <LOCAL-VARIABLE-REF>/pkg/swc/IB/irvex1</LOCAL-VARIABLE-REF>  
    </ACCESSED-VARIABLE>  
  </VARIABLE-ACCESS>  
</WRITTEN-LOCAL-VARIABLES>
```

## 8.11 Accessing Modes

---

A mode user/manager can query the current application mode for a mode instance using the Rte\_Mode API as well as have runnable entities activated as a result of a mode switch event.

Prior to R4.0 a Rte\_Mode API was created by RTA-RTE for all possible mode accesses. This could lead to many unused, and undesired, APIs and therefore in R4.0 AUTOSAR introduced the concept of a ModeAccessPoint to control when and how runnable entities access a mode using the Rte\_Mode API.

A ModeAccessPoint is declared using the <MODE-ACCESS-POINT> element within the context of the runnable that will invoke the API:

```
<MODE-ACCESS-POINTS>
  <MODE-ACCESS-POINT>
    <MODE-GROUP-IREF>
      <R-MODE-GROUP-IN-ATOMIC-SWC-INSTANCE-REF>
        <CONTEXT-R-PORT-REF>/pkg/swc/r1</CONTEXT-R-PORT-REF>
        <TARGET-MODE-GROUP-REF>/pkg/if/mdgp1</TARGET-MODE-GROUP-REF>
      </R-MODE-GROUP-IN-ATOMIC-SWC-INSTANCE-REF>
    </MODE-GROUP-IREF>
  </MODE-ACCESS-POINT>
</MODE-ACCESS-POINTS>
```

A ModeAccessPoint must be declared for each required Rte\_Mode API. The access point can reference either required or provided mode groups.

## 8.12 Per-instance Memory

---

Software component instances may have state that has a lifetime that is longer than the execution of any of its constituent runnable entities. Global variables cannot be used for maintaining state because more than one instance of a software component may be integrated onto the same ECU and there is no concurrency control mechanism between instances of software components.

Per-instance memory allows software components to define one or more typed regions of memory that can be accessed by the runnable entities at runtime.

The configuration of a per-instance memory section is shown below:

```
<INTERNAL-BEHAVIOR>
  ...
  <PER-INSTANCE-MEMORYS>
    <PER-INSTANCE-MEMORY>
      <SHORT-NAME>VAL</SHORT-NAME>
      <TYPE>dataType</TYPE>
      <TYPE-DEFINITION>uint8</TYPE-DEFINITION>
    </PER-INSTANCE-MEMORY>
  </PER-INSTANCE-MEMORYS>
  ...
</INTERNAL-BEHAVIOR>
```

Each per-instance memory must be named using the <SHORT-NAME> element. The name allocated at configuration is the same name you will use to access the per-instance memory in your code. The short-name of a per-instance memory should be a valid C identifier.

The <TYPE> and <TYPE-DEFINITION> elements are used to, respectively, name and define the data type created by RTA-RTE for the per-instance memory. The short-name of a per-instance memory should be a valid C identifier. The type definition must be a valid C type definition suitable for direct inclusion in a type definition as follows:

```
typedef ...TYPE-DEFINITION... ...TYPE...;
```

The example per-instance memory definition results on the following type definition:

```
typedef uint8 dataType;
```

Within the generated RTE, RTA-RTE allocates RAM for each per-instance memory. If the ECUC specifies an NVRamAllocation container that references a per-instance memory then the name is the RamBlockLocationSymbol. Otherwise, RTA-RTE constructs a unique name.

## 8.13 Port Options

---

The port options element permits:

- Enable/disable indirect API for a port.
- Port-defined arguments to be specified for client-server communication.
- Enabling application code to take the “address” of RTE API functions generated for a port.

### 8.13.1 Indirect API

---

If the indirect API generated is not required for a port it can be disabled through “port options” within the internal behavior:

```
<PORT-API-OPTIONS>  
  <PORT-API-OPTION>  
    <INDIRECT-API>false</INDIRECT-API>  
    <PORT-REF DEST="...">/pkg/component/port</PORT-REF>  
  </PORT-API-OPTION>  
</PORT-API-OPTIONS>
```

Disabling the indirect API can reduce the size of the component data structure (and hence save ROM) when a SWC does not support multiple instances.

### 8.13.2 Port-defined Arguments

---

When the same runnable is used as the server for multiple ports it is necessary to be able to identify which is the port that has caused the server invocation. Port-defined

arguments permit a server to be supplied with default arguments based on the port through which it is invoked.

```
<PORT-API-OPTIONS>
  <PORT-API-OPTION>
    <PORT-ARG-VALUES>
      <INTEGER-LITERAL>
        <SHORT-NAME>pdav_1</SHORT-NAME>
        <TYPE-TREF DEST="...">/pkg/SInt16</TYPE-TREF>
        <VALUE>1</VALUE>
      </INTEGER-LITERAL>
      ...
    </PORT-ARG-VALUES>
  <PORT-REF DEST="...">/pkg/component/pport</PORT-REF>
</PORT-API-OPTION>
...
```

### 8.13.3 Enable "Take Address"

A port option can be used to ensure that RTE APIs are generated in a way that their address can be taken at run-time:

```
<PORT-API-OPTIONS>
  <PORT-API-OPTION>
    <ENABLE-TAKE-ADDRESS>true</ENABLE-TAKE-ADDRESS>
    <PORT-REF DEST="...">/pkg/component/port</PORT-REF>
  </PORT-API-OPTION>
</PORT-API-OPTIONS>
```

If not specified, RTA-RTE assumes that <ENABLE-TAKE-ADDRESS> is false.

## 8.14 Supporting Multiple Instantiation

By default, software components will only support the creation of a single software component instance. If you want to support multiple instantiation then this must be declared in the <INTERNAL-BEHAVIOR> clause:

```
<SUPPORTS-MULTIPLE-INSTANTIATION>
  true
</SUPPORTS-MULTIPLE-INSTANTIATION>
```

The element takes the values of "true" indicating that multiple instantiation is supported or "false" indicating that multiple instantiation is not supported.

If the <SUPPORTS-MULTIPLE-INSTANTIATION> element is omitted, RTA-RTE assumes that the software-component **does not** support multiple instantiation.



*The AUTOSAR 4.x feature of STATIC-MEMORYS is only available in Internal-Behaviors that are single-instantiated. See errors 2469 and 2470 for more information.*

## 8.15 Memory Allocation

RTA-RTE uses the AUTOSAR memory abstraction scheme when declaring RunnableEntities. A Memory Allocation Keyword is chosen according to the SwAddrMethod for the RunnableEntity. If the RunnableEntity on the input references a swAddrMethod then its shortName is used for the Memory Allocation Keyword; otherwise RTA-RTE uses the AUTOSAR specified SwAddrMethod CODE.

For example:

```
<RUNNABLE-ENTITY>  
  <SHORT-NAME>re2</SHORT-NAME>  
  <SW-ADDR-METHOD-REF>/pkg/METHOD1</SW-ADDR-METHOD-REF>  
  <SYMBOL>swcA_re1</SYMBOL>  
  ...
```

The resultant definition of the runnable generated by RTA-RTE in the SWC type's application header file then uses the appropriate section start/stop definitions:

```
#define swcA_START_SEC_METHOD1  
#include "swcA_MemMap.h"  
FUNC(void, swcA_METHOD1) swcA_re1(void);  
#define swcA_STOP_SEC_METHOD1  
#include "swcA_MemMap.h"
```



*This applies to ASW only; this behaviour is not specified for BSW.*

## 9 Modes

---

The previous chapters have explored how an AUTOSAR software-component type can be defined and configured. In this chapter you will learn how to define application modes that can be used by software-components to control the execution of runnable entities.

### 9.1 Defining Modes

---

Modes are declared within a `<MODE-DECLARATION-GROUP>` contained within an `<AR-PACKAGE>` element.

```
<AR-PACKAGE>
  <SHORT-NAME>MyModes</SHORT-NAME>
  <ELEMENTS>
    <MODE-DECLARATION-GROUP>
      ...
    </MODE-DECLARATION-GROUP>
  </ELEMENTS>
</AR-PACKAGE>
```

The `<MODE-DECLARATION-GROUP>` element is used to declare one or more modes that are subsequently used by interface declarations.

```
<MODE-DECLARATION-GROUP>
  <SHORT-NAME>Modes</SHORT-NAME>
  <INITIAL-MODE-REF>/pkg/Modes/iMode</INITIAL-MODE-REF>
  <MODE-DECLARATIONS>
    <MODE-DECLARATION>
      <SHORT-NAME>iMode</SHORT-NAME>
    </MODE-DECLARATION>
    <MODE-DECLARATION>
      <SHORT-NAME>runMode</SHORT-NAME>
    </MODE-DECLARATION>
  </MODE-DECLARATIONS>
</MODE-DECLARATION-GROUP>
```

One mode within a `<MODE-DECLARATION-GROUP>` element is marked as the group's initial mode through the `<INITIAL-MODE-REF>`. Mode switch events that are attached to the ENTRY of an initial mode are triggered by RTA-RTE when the RTE is started using `Rte_Start`.

A `<MODE-DECLARATION-GROUP>` can be used (referenced) by multiple sender-receiver interfaces and therefore inherently used by multiple software-components.

### 9.2 Mode Communication

---

Modes are communicated using mode-switch interfaces. Each mode-switch interface can specify exactly one mode declaration group prototype that defines the AUTOSAR modes communicated over the interface.

```
<MODE-SWITCH-INTERFACE>
  <SHORT-NAME>ModeInterface</SHORT-NAME>
```

```
<MODE-GROUP>  
  <SHORT-NAME>mdg1</SHORT-NAME>  
  <TYPE-TREF DEST=". . .">/MyModes/Modes</TYPE-TREF>  
</MODE-GROUP>  
...  
<MODE-SWITCH-INTERFACE>
```

A mode-switch interface can contain exactly one mode declaration group prototype defined using the <MODE\_GROUP> element.

Each <MODE-DECLARATION-GROUP-PROTOTYPE> or <MODE\_GROUP> element must specify:

- the <SHORT-NAME> that you will use to refer to the item within references from other elements.
- a <TYPE-TREF> reference to the mode declaration group.

### 9.2.1 Mode Managers and Mode Users

A software-component can be either<sup>1</sup> a mode manager responsible for requesting a switch to a new mode or a mode user activated in response to a mode switch.


A software-component declares itself as a mode manager by providing a port categorized by a sender-receiver interface containing a mode declaration group prototype and declaring a mode switch point. The software-component can then use the `Rte_Switch` API to request that a mode switch occurs as well as have runnable entities activated as a result of a mode switch event.

A software-component declares itself as a mode user by requiring a port categorized by a sender-receiver interface containing a mode declaration group prototype.

A mode manager is connected, using assembly connectors, to the mode users.

### 9.2.2 Mode Queues

When a mode manager initiates a mode switch RTA-RTE will be processing the request immediately as long as no mode switch is currently in progress for the mode instance. However if the switch cannot begin then it is placed in a queue of mode switch requests and handled when the current switch is complete.

 *RTA-RTE eliminates queues when they are not required; for example if no `ModeSwitchEvents` are defined for the mode instance.*

The length of the queue can be defined for each mode manager. For SWCs the length is defined by the provided port's com-spec element, for example to specify a queue length of three entries:

```
<PROVIDED-COM-SPECS>  
  <MODE-SWITCH-SENDER-COM-SPEC>
```

<sup>1</sup>It is possible for the same software-component to be simultaneously a mode manager and a mode user if it contains both provided and required ports.



```
<MODE - GROUP - REF>/pkg/if/mdgp1</MODE - GROUP - REF>  
<QUEUE - LENGTH>3</QUEUE - LENGTH>  
</MODE - SWITCH - SENDER - COM - SPEC>  
</PROVIDED - COM - SPECS>
```

The queue length can be specified for a BSW module by a <BSW-MODE-SENDER-POLICY> element within the module's <MODE-SENDER-POLICYS> element.

```
<BSW - MODE - SENDER - POLICY>  
<PROVIDED - MODE - GROUP - REF>/pkg/module/group</PROVIDED - MODE - GROUP - REF>  
<QUEUE - LENGTH>3</QUEUE - LENGTH>  
</BSW - MODE - SENDER - POLICY>
```

The Rte\_Switch API returns RTE\_E\_LIMIT error if the mode queue overflows. The equivalent BSW API returns SCHM\_E\_LIMIT in the same circumstances.

### 9.2.3 Accessing Modes

A mode user/manager can query the current mode using the Rte\_Mode API as well as have runnable entities activated as a result of a mode switch event.

ModeAccessPoints are used to control when and how runnable entities access a mode using the Rte\_Mode API. A ModeAccessPoint is declared using the <MODE-ACCESS-POINT> element within the context of the runnable that will invoke the API:

```
<MODE - ACCESS - POINTS>  
<MODE - ACCESS - POINT>  
<MODE - GROUP - IREF>  
<R - MODE - GROUP - IN - ATOMIC - SWC - INSTANCE - REF>  
<CONTEXT - R - PORT - REF>/pkg/swc/r1</CONTEXT - R - PORT - REF>  
<TARGET - MODE - GROUP - REF>/pkg/if/mdgp1</TARGET - MODE - GROUP - REF>  
</R - MODE - GROUP - IN - ATOMIC - SWC - INSTANCE - REF>  
</MODE - GROUP - IREF>  
</MODE - ACCESS - POINT>  
</MODE - ACCESS - POINTS>
```

A ModeAccessPoint must be declared for each required Rte\_Mode API. The access point can reference either required or provided mode groups.

### 9.2.4 Interface Compatibility

For a provider to be successfully connected to a receiver the interfaces categorizing the ports must have compatible interfaces – for modes this means that each Mode Declaration Group Prototype in the required interface must have the same name<sup>2</sup> as a Mode Declaration Group Prototype in the provided reference (this is analogous to the requirement for matching name and data-type for connecting interfaces containing Data Element Prototypes). The Mode Declaration Group Prototypes must have compatible modes.

<sup>2</sup>Port interface mapping can be used to map mode declaration prototypes with dissimilar names.

It is permitted for a provider to provide more Mode Declaration Group Prototypes than are required.

### 9.2.5 Synchronous and Asynchronous Mode Switch

The `Rte_Switch` API supports both *synchronous* and *asynchronous* mode switches. RTA-RTE uses the synchronous mode unless all mode users explicitly enable support for asynchronous mode switches.

#### Synchronous Mode Switch

When synchronous mode switches are used the `Rte_Switch` API activates the task containing triggered mode switch runnables (all such runnables for a given mode instance must be mapped to the same task). The execution of mode switch runnables is synchronized with the termination of any executing runnables with mode disabling dependencies and thus is guaranteed that such runnables will have terminated when the mode switch runnables are invoked.

#### Asynchronous Mode Switch

Support for asynchronous mode switches is enabled or disabled using a `ModeSwitchReceiverComSpec` element within the requiring port prototype or a `BswModeReceiverPolicy` within the BSW's internal behaviour.

```
<MODE-SWITCH-RECEIVER-COM-SPEC>
  <SUPPORTS-ASYNCHRONOUS-MODE-SWITCH>true</SUPPORTS-ASYNCHRONOUS-MODE-
    SWITCH>
</MODE-SWITCH-RECEIVER-COM-SPEC>

<BSW-MODE-RECEIVER-POLICY>
  <REQUIRED-MODE-GROUP-REF DEST="...">
    ...
  </REQUIRED-MODE-GROUP-REF>
  <SUPPORTS-ASYNCHRONOUS-MODE-SWITCH>true</SUPPORTS-ASYNCHRONOUS-MODE-
    SWITCH>
</BSW-MODE-RECEIVER-POLICY>
```



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

When asynchronous mode switches are used the `Rte_Switch` or `SchM_Switch` API can directly invokes the mode switch runnables (unless the runnable has a minimum start interval or is in a different OS Application) and thus these run in the context of the caller. The execution of mode switch runnables is thus not synchronized with the termination of any executing runnables with mode disabling dependencies and such runnables may not have terminated when the mode switch runnables are invoked.

RTA-RTE supports direct invocation of the mode switch runnable if the relevant RTE event mappings does not specify a task. If the mode switch runnable entry point function is to be directly invoked then the input configuration must still provided an RTE

event mapping for the associated event. The mapping must specify the position in the task (for ordering purposes within the Switch API) but must not specify a task.

### 9.2.6 Mode Switch Management

RTA-RTE provides APIs for requesting a mode switch within a mode instance graph and for reading the current mode. All mode switch management occurs either within the generated switch API itself or within the task containing the mode switch event triggered runnable entities.

## 9.3 Using Modes

### 9.3.1 Software Component Initialization & Finalization

AUTOSAR modes can be used to execute code when the RTE is started, e.g. to initialize internal data structures etc. Similarly, when a system is shut down your software component may need to store data, log operational details etc.

Each mode declaration group describes an initial mode—to activate a runnable when the system is started created by a <MODE-SWITCH-EVENT> or <SWC-MODE-SWITCH-EVENT> element for entry to the initial mode.

A runnable entity within a software-component can be started when the RTE is started by declaring a mode-switch event for entry to an initial mode.

Within the mode switch event definition the particular mode instance is selected using <R-PORT-PROTOTYPE-REF>, <MODE-DECLARATION-GROUP-PROTOTYPE-REF>, <CONTEXT-PORT-REF> and <TARGET-MODE-DECLARATION-REF> elements:

```
<INTERNAL-BEHAVIOR>
  <SHORT-NAME>Behaviour</SHORT-NAME>
  <EVENTS>
    <SWC-MODE-SWITCH-EVENT>
      <SHORT-NAME>mse1</SHORT-NAME>
      <START-ON-EVENT-REF>
        /pkg/internal/init
      </START-ON-EVENT-REF>
      <ACTIVATION>ON-ENTRY</ACTIVATION>
      <MODE-IREFS>
        <MODE-IREF>
          <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF ...
          <CONTEXT-PORT-REF ...
          <TARGET-MODE-DECLARATION-REF ...
        </MODE-IREF>
      </MODE-IREFS>
    </SWC-MODE-SWITCH-EVENT>
  </EVENTS>
  ...
  <RUNNABLES>
    <RUNNABLE-ENTITY>
      <SHORT-NAME>init</SHORT-NAME>
      <SYMBOL>InitializationFunction</SYMBOL>
```

```
</RUNNABLE-ENTITY>  
</RUNNABLES>  
</INTERNAL-BEHAVIOR>
```

Once the mode switch event has been configured an initialization runnable entity must be written that has the following structure:

```
FUNC(void, RTE_APPL_CODE)  
InitializeFunction(Rte_Instance Self)  
{  
    /* user code */  
}
```

The initialization runnable entity must be a Category 1 runnable entity and hence must not make any (blocking) RTE calls nor access other application components.

Similarly, when a system is shut down your software component may need to store data, log termination etc. The principle is the same as initialization, except that finalization is simply a transition to a new mode that is simply associated with shutdown.

### 9.3.2 Triggering a Runnable Entity on a Mode Switch

A runnable entity can be activated on either entry, transition or exit from a mode using a Mode Switch Event configured, like all other events, in the <INTERNAL-BEHAVIOR> element of a software-component.

The <MODE-SWITCH-EVENT> selects the mode by the <R-PORT-PROTOTYPE-REF>, <MODE-DECLARATION-GROUP-PROTOTYPE-REF> and <MODE-DECLARATION-REF> elements:

```
<MODE-SWITCH-EVENT>  
  <SHORT-NAME>mse1</SHORT-NAME>  
  <START-ON-EVENT-REF>/pkg/IB/initialize</START-ON-EVENT-REF>  
  <ACTIVATION>ENTRY</ACTIVATION>  
  <MODE-IREF>  
    <R-PORT-PROTOTYPE-REF ...  
    <MODE-DECLARATION-GROUP-PROTOTYPE-REF ...  
    <MODE-DECLARATION-REF ...  
  </MODE-IREF>  
</MODE-SWITCH-EVENT>
```

Finally, the declaration of the <SWC-MODE-SWITCH-EVENT> can reference one or two modes within its <MODE-IREFS> element. One referenced mode is required for enter/exit mode switch events and two referenced modes for transition events.

```
<SWC-MODE-SWITCH-EVENT>  
  <SHORT-NAME>mse1</SHORT-NAME>  
  <START-ON-EVENT-REF>  
    /pkg/internal/init  
  </START-ON-EVENT-REF>  
  <ACTIVATION>ON-ENTRY</ACTIVATION>
```

```
<MODE-IREFS>
  <MODE-IREF>
    <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF ...
    <CONTEXT-PORT-REF ...
    <TARGET-MODE-DECLARATION-REF ...
  </MODE-IREF>
</MODE-IREFS>
</SWC-MODE-SWITCH-EVENT>
```

A mode-switch event element defines three things:

1. The <START-ON-EVENT-REF> element defines the runnable entity to activate. The reference must be to a runnable entity within the same software-component type.
2. The <ACTIVATION> element defines the activation for the mode switch event, i.e. when the runnable entity is triggered. A Mode Switch Event can apply to one activation condition. If multiple activation conditions are required for a runnable then more than one Mode Switch Event should be defined.


RTA-RTE supports “ENTRY”, “EXIT” or “ON-TRANSITION” activation conditions. A mode switch event with one of these activation conditions is triggered whenever the referenced mode is entered or exited.

Mode switch events using “ON-TRANSITION” activation are triggered by RTA-RTE when the mode instance moves from the first referenced mode to the second referenced mode.

3. The <MODE-IREF> defines the mode associated with the mode switch event. The <MODE-IREF> element must contain three references (the port prototype, the mode declaration group prototype and the mode declaration group that types the declaration group prototype).

A mode switch event with an “ON-TRANSITION” activation condition requires two modes to be referenced from the mode switch event’s <MODE-IREFS> element. Both referenced modes must be within the same mode instance.

One mode within a <MODE-DECLARATION-GROUP> element is marked as the group’s initial mode. Any mode switch events that are attached to the “ENTRY” of an initial mode within any group are triggered by RTA-RTE when the RTE is started using Rte\_Start.

 *The order of execution of runnable entities when more than one runnable entity is triggered by the same mode entry (or exit) is not defined by the SWC-type. For portability, therefore, a system should not rely on a particular execution order since it may be redefined in the runnable to task mapping (see Section 18.4.4).*

### 9.3.3 Disabling RTE Events

A <MODE-DEPENDENCY> permits the behavior of an RTE event to be different in different modes. This allows such use-cases as the activation of a runnable entity to be suppressed when a certain mode is active.

A mode dependency declaration can be added to any RTE event (other than OperationInvokedEvents). A single RTE event can contain multiple mode dependency declarations in which case it is disabled in all the specified modes.

The <DISABLED-MODE-IREF> element selects the mode using the <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF>, <CONTEXT-PORT-REF> and <TARGET-MODE-DECLARATION-REF> elements:

```
<DISABLED-MODE-IREF>
  <CONTEXT-MODE-DECLARATION-GROUP-PROTOTYPE-REF ...
  <CONTEXT-PORT-REF ...
  <TARGET-MODE-DECLARATION-REF ...
</DISABLED-MODE-IREF>
```

When the mode specified within the dependency is active, RTA-RTE will **not activate** the runnable (the activation is discarded — it is not queued).



*When a runnable activation triggered by, for example a DataReceivedEvent with queued communication, is suppressed, RTA-RTE also suppresses the write to the runnable entity's queue – this prevents the queue filling up without corresponding runnable activations to process the events.*

## 9.4 Understanding Mode Instances

A unique 'current mode' is maintained by RTA-RTE for each connected graph of compatible Mode Declaration Group Prototypes. If an interface contains multiple prototypes that reference different Mode Declaration Groups then the RTE defines a unique 'current mode' value for each prototype. If the Mode Declaration Group Prototypes reference the same Mode Declaration Group then each 'current mode' can take one of the same set of modes but their state remains independent.

As an example, assume we have SW-C swcA with provided port P1 and swcB with required port R1 both categorized by the same interface. An Assembly Connector that connects instances of swcA and swcB (e.g. A1.P1 to B1.R1) will result in a unique 'current mode' maintained by RTA-RTE.

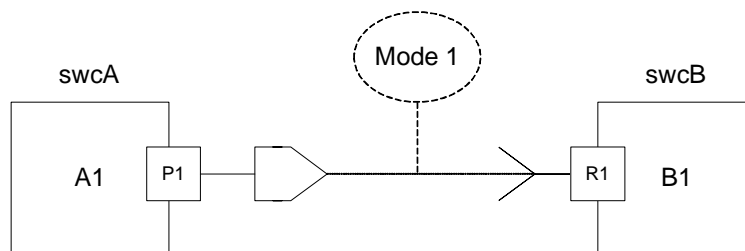


Figure 9.1: Single mode manager and mode user

Furthermore, another Assembly Connector that connects different instances of swcA and swcB (e.g. A2.P1 to B2.R1) will result in RTA-RTE maintaining two independent 'current modes' since the graphs formed by the connections do not intersect.

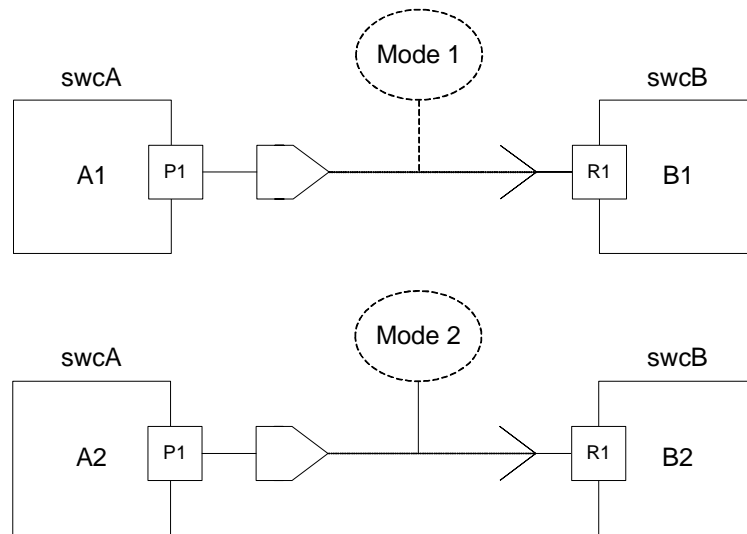


Figure 9.2: Multiple distinct mode managers and mode users

However if the additional Assembly Connector connects the same instance of swcA to a new instance of swcB (e.g. A1.P1 to B2.R1) then a new 'current mode' is not created by RTA-RTE but instead an additional mode user can access the same mode.

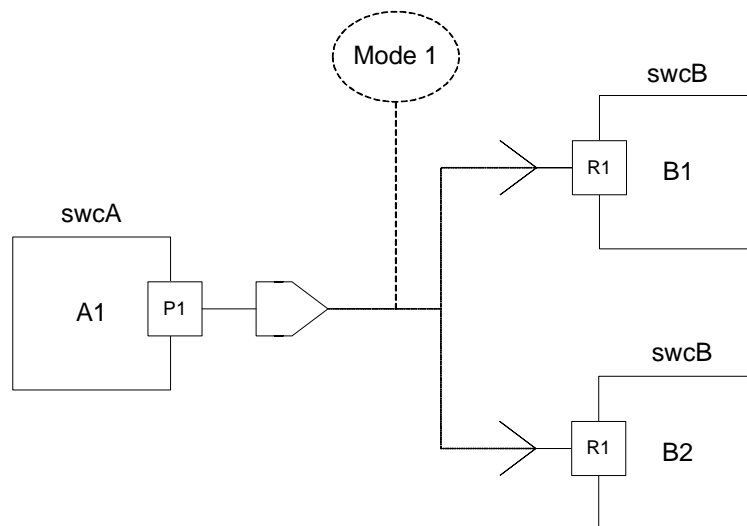


Figure 9.3: Single mode manager and multiple mode users



*Multiple mode managers for the same mode instance are not supported by AUTOSAR.*

## 9.5 Fast Init

RTA-RTE supports a “fast init” activation method for Mode Switch Events that bypasses the normal AUTOSAR activation method for events that are triggered only once at system startup. This mechanism is useful for a runnable that only runs once at system start where the infrastructure code created by RTA-RTE to support the mode changes

at run-time is superfluous.

**ETAS** *The “fast init” activation method is unique to RTA-RTE and therefore non-portable.*

An individual ModeSwitchEvent is made subject to “fast init” activation through a command-line option (see *RTA-RTE Reference Manual*). It is also possible to make all ModeSwitchEvents for a SWC subject to “fast init” through the same command-line option.

When a ModeSwitchEvent is “fast init” then the implementation created by RTA-RTE within the task is a simple function call — the use of a *FastInit* task both simplifies the activation code necessary for the runnable associated with the event as well as removing the normal infrastructure code created by RTA-RTE to support the AUTOSAR mode mechanism.

It is the user’s responsibility to ensure that all *FastInit* tasks are activated at the appropriate time.



*An event that is specified as FastInit will not be activated by RTA-RTE during a user-triggered mode switch. Instead the FastInit task must be activated by user code at the appropriate time.*

## 9.6 Synchronizing Modes

---

RTA-RTE supports the synchronization of modes between SWC components and BSW modules. Synchronizing mode groups permits both events in both AUTOSAR software components and basic software modules to be triggered by a single mode manager. RTA-RTE supports synchronization where the referenced BSW *provides* or *releases* (requires) the mode group.

The synchronization of mode groups is covered in detail in Chapter 15.

## 9.7 Non-AUTOSAR Functionality

---

### 9.7.1 Shared Mode Queues

---

When enabled with the `--deviate-shared-mode-queue` command-line option a shared mode queue means that all mode instances mapped to a task share a single (common) queue.

**ETAS** *Shared mode queues do not support timeouts for ModeSwitchAck events nor can they be mixed with distributed mode users.*

A Shared Mode Queue results in a more predictable mode transition than is possible with AUTOSAR compliant mechanisms since for mode instances assigned to the task it is guaranteed that the transition for one mode instance will be complete before another is started.



## 9.7.2 Distributed Mode Users

When enabled with the `--mode-policy` command-line option different mode switch events for a mode instance can be mapped to different tasks.

### Dequeue Synchronization

When multiple mode switch tasks exist for a single mode instance then any task can be responsible for performing the dequeue. RTA-RTE provides a callback, `Rte_ModeDequeueCbk_<index>`, for each mode instance that controls within which task the dequeue occurs.

The callback is called from each mode switch task after the mode switch has completed for the calling task. When it returns `TRUE` the generated code dequeues the mode switch.



*The callback is only invoked when a mode switch instance is in a transition.*

During each transition the callback must return `TRUE` for exactly one of the mode switch tasks but which task is selected may vary between transitions. For example the callback might return `TRUE` for the last task to finish the mode switch and enter the callback, which could be timing-dependent, especially if the tasks run on different cores.

The name of the callback includes the mode instance index “<index>”. RTA-RTE makes this index available within the `Rte_Const.h` generated header file for each provide mode port (i.e. mode manager) using the `RTE_DMU` macro:

```
#define RTE_DMU_<swci>_<port>_<modegroup> ...
```

Where “<swci>” is the SWC instance name (which can be set using a flatmap, see Section 13.4), “<port>” is the name of the provided mode port and “<modegroup>” is the name of the mode declaration prototype within the mode interface categorizing the port.

The “`RTE_DMU_...`” macro can be used with the C-preprocessor to construct the name of the callback function.

## 10 Implementing Software Components

---

This section shows how to write software components so that the objects required by the RTE are declared and how to use the RTE API generated by the RTE generator.

### 10.1 Basic Concepts

---

#### 10.1.1 Namespace

---

All RTE symbols (e.g. function names, global variables etc.) that are visible in the global namespace use either the prefix `Rte_` or the prefix `RTE_`.



*You must not create symbols that use either the prefix `Rte_` or the prefix `RTE_` to remove the possibility of namespace clashes.*

#### 10.1.2 Runnable Naming Convention

---

RTE generator generates code that activates your runnable entities. To do this, the RTE's internal mechanisms need to be able to access your code through defined interfaces.

Each of the named runnable entities defined in your runnable entity `<SYMBOL>` declarations must be implemented. Failure to define all runnable entities will be detected at compile time when your application is linked to form the ECU's executable image. The linker error message will reference the missing runnable entity entry point.

Runnable entities are executed by RTE generated code when required. The function providing an entry point for a runnable entity should not be invoked directly by an application software-component.

#### 10.1.3 API Naming Convention

---

The RTE API calls are generated for each software component using names derived from the the RTE generator's input. The RTE API provides a consistent interface to each software component but allows the RTE generator to provide different implementations of the API functionality.

Each API call name is formed from:

- An AUTOSAR defined prefix. For RTE API functions this is "`Rte_`" and, for BSW functions, "`SchM_`".
- The call functionality (Read, Write, etc.).
- Either
  - The port name and data item name (sender-receiver) or operation name (client-server) through which the API operates.
  - The name of the object (e.g. trigger, exclusive area, per-instance memory etc) upon which the API operates.

Thus RTE API calls involving communication through ports have the format:

```
Rte_StatusType  
Rte_<API call name>_<port>_<dataitem/operation>
```

Whereas other RTE APIs have the format:

```
Rte_StatusType  
Rte_<API call name>_<object name>
```

#### 10.1.4 API Parameter Passing Mechanisms

RTE API calls and runnable entities may be invoked from multiple component prototypes. In both cases, the RTE expects that the first parameter to an API call or to a runnable entity to indicate the prototype upon which the function operates.



*Every RTE API call or runnable entity that can operate on different prototypes of a component must pass in the instance handle as the first formal parameter.*

The second and subsequent API parameters (if any) fall into one of three classes:

**“In” Parameters** —All “in” parameters that are AUTOSAR primitive data types (with the exception of a string) are passed by value. Strings and other “in” parameters that are a complex data type (i.e. a record or an array) are passed by reference.

Note that while AUTOSAR defines a string as a primitive data type, its inherent size makes it inefficient to pass by value and is therefore treated the same as a complex data type.

“In” parameters are strictly read-only.

**“Out” Parameters** —All “out” parameters are passed to RTE API functions by reference. This is required to ensure that the API functions can modify the parameter.

“Out” parameters are strictly write-only.

**“In/Out” Parameters** —All “in/out” parameters are passed to the RTE API functions by reference except for an asynchronous client-server call when primitive data types (other than strings) are passed by value to `Rte_Call` and by reference to `Rte_Result`.

“In/out” parameters can be read and written by the API function being called.

#### 10.1.5 API Forms

RTA-RTE supports two forms of API; direct and indirect. The direct API form includes the instance handle as the first parameter of all API calls whereas the indirect form operates through a port handle. The direct form is subject to a higher potential level of optimization than the indirect form but the latter supports iteration over ports typed by the same interface.



Both direct and indirect forms of the RTE API are equivalent and result in the same generated RTA-RTE function being invoked. The direct and indirect forms of the RTE API can be mixed within a program – even within the same software-component.

#### Direct API

---

The direct API form is the most efficient mechanism for invoking RTA-RTE generated functions since, depending on the configuration, RTA-RTE is capable of optimizing the API form to bypass the component data structure to produce direct invocation of generated functions. Furthermore, for source-code components the direct API form can potentially be further optimized to elide the RTA-RTE generated function completely.

#### Indirect API

---

The indirect API form of API invocation that uses indirection through a port handle to invoke RTA-RTE API functions.

The indirect form is less efficient than the direct form since the indirection cannot be optimized away but supports iteration over ports typed by a similar interface. 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.

If the indirect API generated is not required for a port it can be disabled through “port options” within the internal behaviour:

```
<PORT-API-OPTIONS>
  <PORT-API-OPTION>
    <INDIRECT-API>false</INDIRECT-API>
    <PORT-REF DEST=". . .">/pkg/component/port</PORT-REF>
  </PORT-API-OPTION>
</PORT-API-OPTIONS>
```

## 10.2 Application Source Code

---

### 10.2.1 Supported Programming Languages

---

RTA-RTE supports components written in C and C++. The same application header file can be used for C and C++.

The generated RTE is independent of the programming language used to create the components since an individual RTE may be required to simultaneously support components written in different languages. RTA-RTE always generates the RTE in C.

### 10.2.2 Application Header Files

---

Each component you write must reside in its own C or C++ source code file(s) and include the relevant application header file created during RTE configuration.

```
#include "Rte_ComponentName.h"

/* Component Implementation for "ComponentName" */
```



*The RTE API is specific to each software component type and therefore you must include only the component's application header file for each source code file defines a component (whether completely or partially).*

A single source module must not include multiple application header files as the API mappings they contain may be different for different software-components. The header files generated by the RTE generator protect against such multiple file inclusion.

The component type specific header file defines the component's RTE API and provides access to the type definition through a component instance handle that is used to identify particular prototypes of a component.

### 10.3 Single and Multiple Instances

When a software component is configured to support a single instance it is known at integration time that there will only be one instance of each software component type.

However, it is possible to configure a software component to support multiple instances. When this is configured, the RTE provides an instance handle for each component instance.

The instance handle is passed to a software component as the first parameter of the component's runnable entity (or entities).

Each time an API is made from the runnable entity then you must pass the same instance handle back to the RTE as the first parameter of API calls.

If a software component type supports only a single instance then an instance handle is not passed to runnable entities and therefore cannot be passed back to API calls.

The following code example assumes a single instance:

```
#include Rte_Component.h''

FUNC(void, RTE_APPL_CODE)
RunnableEntity(void)
{
    UInt16 Result = 0;

    // ...
    Rte_Send_FromPort_MyData(42);
    // ...
    Result = SomeOtherFunction(42);
    // ...
}
```

Now, if the same example is re-written for a software component that supports multiple instances we get:

```
#include Rte_Component.h''
```

```
FUNC(void, RTE_APPL_CODE)
RunnableEntity(Rte_Instance Self)
{
    UInt16 Result = 0;

    ...
    Rte_Send_FromPort_MyData(Self, 42};
    ...
    Result = SomeOtherFunction(42);
    ...
}
```

Note that if runnable entities need take different actions for different instances then per-instance calibration parameters can be used to identify each instance—the instance name/handle is not visible to the component (i.e. there is information hiding) and hence is not suitable. Access to the state of an instance is only possible through the `Rte_Pim` API call (see Section 10.9).

The following sections all show examples with multiple instance support configured, however the conversion to single instance is trivial (as shown above).

## 10.4 Runnable Entities

You are responsible for providing the implementations of all the runnable entities required to make a software component work at runtime.

You must provide an entry point (i.e. a C function) for each <RUNNABLE-ENTITY> you have declared in the component description. It is possible for multiple runnable entities to share the same entry point function.

If the software component supports multiple instantiation, all its runnable entities are passed the instance handle as a function parameter when it is activated by the RTE so the code you write must expect an instance handle as its first parameter. If the runnable entity is triggered by an <OPERATION-INVOKED-EVENT> then additional parameters corresponding to the configuration of the client-server operation may be required.

RTA-RTE supports components written in C++. However, the generated RTE, including the generated task bodies, is created in C and therefore all runnable entity entry point functions must be exported with C linkage:

```
extern "C"
FUNC(void, RTE_APPL_CODE)
re_entry_point(Rte_Instance self)
{
    ...
}
```

#### 10.4.1 Entry Point Signature

---

With the exception of a runnable entity invoked as a result of an “Operation Invoked” RTE event, the signature of a runnable entity entry point function must follow the following implementation rules:

- There is a single formal parameter (the instance handle) only if the internal behaviour of the software component supports multiple instantiation.
- There are no user-defined parameters.
- There is no return value (i.e. a return type of void must be specified) unless a server specifies application errors in which case Std\_ReturnType is used.

The signature of a runnable entity that responds to an “Operation Invoked” RTE event must conform to the following additional rules:

- Formal parameters in addition to the instance handle are the operations IN, IN/OUT and OUT parameters. These parameters are passed by value or reference depending on the type.

#### 10.4.2 Sample Runnable Implementations

---

RTA-RTE can generate sample application source files using the `--samples=swc` command-line option. The name of the generated sample file is `Rte_<swc>.c` where `swc` is the software component’s short name.

The generated file contains an empty function for each runnable entity.

The `--samples=swc` option is most useful in contract phase before starting implementation of a software component.

#### 10.4.3 Examples

---

The following examples present the configurations and resultant obligations on software-component implementation of different RTE events triggering runnable entities.

##### Basic Configuration

---

All RTE events other than “Operation Invoked” RTE events use the same basic signature for runnable entity entry points irrespective of the event that actually triggers the runnable entity.

For example, given the following declaration of two runnable entities in the XML configuration file for software-component Component:

```
<SWC-INTERNAL-BEHAVIOR>  
  <SHORT-NAME>Behaviour</SHORT-NAME>  
  <RUNNABLES>
```

```
<RUNNABLE-ENTITY>
  <SHORT-NAME>A</SHORT-NAME>
  <SYMBOL>Runnable_A</SYMBOL>
</RUNNABLE-ENTITY>
<RUNNABLE-ENTITY>
  <SHORT-NAME>B</SHORT-NAME>
  <SYMBOL>Runnable_B</SYMBOL>
</RUNNABLE-ENTITY>
</RUNNABLES>
</SWC-INTERNAL-BEHAVIOR>
```

Then, following the rules defined above, the following runnable entry point functions would be required:

```
#include Rte_Component.h''

FUNC(void, RTE_APPL_CODE)
Runnable_A(Rte_Instance Self)
{
  /* Implementation of Runnable A */
}

FUNC(void, RTE_APPL_CODE)
Runnable_B(Rte_Instance Self)
{
  /* Implementation of Runnable B */
}
```

### Data Receiver

---

A sender-receiver receiver runnable entity will be invoked by RTA-RTE when a port receives data (or events) and a <DATA-RECEIVED-EVENT> is specified that references a runnable entity.

The signature of the runnable entity must conform to the rules defined above, i.e.:

- There must be no return value.
- The first and only parameter is the instance handle
- The memory class must be RTE\_APPL\_CODE (this can be modified, see Section 8.15).

### Server Operation

---

A runnable entity will be invoked by RTA-RTE each time a request is made for an operation on the server's port. Each component that defines a port that uses an interface with the type SERVER must implement this type of runnable entity.

The signature of the runnable entity must conform to the rules defined above, i.e.:



- There must be no return value (unless application errors are used).
- The first parameter is the instance handle.
- Other parameters are the operation's IN, OUT and INOUT parameters.
- The memory class must be RTE\_APPL\_CODE (this can be modified, see Section 8.15).

Assuming the following interface declaration in the configuration:

```
<CLIENT-SERVER-INTERFACE>
  <SHORT-NAME>CSInterface</SHORT-NAME>
  <OPERATIONS>
    <OPERATION-PROTOTYPE>
      <SHORT-NAME>Mod8</SHORT-NAME>
      <ARGUMENTS>
        <ARGUMENT-PROTOTYPE>
          <SHORT-NAME>InputVal</SHORT-NAME>
          <TYPE-TREF>/types/UInt16</TYPE-TREF>
          <DIRECTION>in</DIRECTION>
        </ARGUMENT-PROTOTYPE>
        <ARGUMENT-PROTOTYPE>
          <SHORT-NAME>OutputVal</SHORT-NAME>
          <TYPE-TREF>/types/UInt16</TYPE-TREF>
          <DIRECTION>out</DIRECTION>
        </ARGUMENT-PROTOTYPE>
      </ARGUMENTS>
    </OPERATION-PROTOTYPE>
  </OPERATIONS>
</CLIENT-SERVER-INTERFACE>
```

And a software component Component that includes a provide port Calculate categorized by interface CSInterface then the internal behavior can define the runnable (and hence C function) implementing the server:

```
<INTERNAL-BEHAVIOR>
  <SHORT-NAME>Behaviour</SHORT-NAME>
  <COMPONENT-REF>/pkg/Component</COMPONENT-REF>
  <EVENTS>
    <OPERATION-INVOKED-EVENT>
      <SHORT-NAME>ActivateServer</SHORT-NAME>
      <START-ON-EVENT-REF>
        /pkg/Behaviour/CalculateMod8
      </START-ON-EVENT-REF>
      <OPERATION-IREF>
        ...
      </OPERATION-IREF>
    </OPERATION-INVOKED-EVENT>
  </EVENTS>
  <RUNNABLES>
    <RUNNABLE-ENTITY>
      <SHORT-NAME>CalculateMod8</SHORT-NAME>
      <SYMBOL>Mod8Function</SYMBOL>
```

```
</RUNNABLE-ENTITY>  
</RUNNABLES>  
</INTERNAL-BEHAVIOR>
```

The component source must then define a runnable entity to be invoked by the RTE when a request is received for operation “Mod8” on port “Calculate”:

```
#include Rte_Component.h''  
  
FUNC(void, RTE_APPL_CODE)  
Mod8Function(Rte_Instance Self,  
             UInt16      InputVal,  
             UInt16*     OutputVal,  
{  
    /* Implementation of Mod8 */  
    *OutputVal = InputVal % 8;  
}
```

Servers may be invoked from multiple sources, for example through a request from a client received via the communication service or directly via intra-task communication. Unless marked as concurrently executable within the runnable’s configuration the RTE will serialize access to the server, queuing requests on a first-in/first-out basis.

### Asynchronous Server Result

When an <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT> references a runnable entity then the runnable entity will be invoked by RTA-RTE each time the corresponding server runnable returns. Each port that uses a client interface with the CLIENT\_MODE attribute set to asynchronous requires this type of runnable entity to be implemented by the user.

The server’s result is itself collected using the Rte\_Result API call; however the signature of the runnable entity must conform to the rules defined above, i.e.:

- There must be no return value.
- The first and only parameter is the instance handle
- The memory class must be RTE\_APPL\_CODE (this can be modified, see Section 8.15).

Assuming a software component MyComponent that includes a require port Client categorized by client-server interface CSInterface then the component can asynchronously access a server and collect the result as follows:

```
<INTERNAL-BEHAVIOR>  
  <SHORT-NAME>Behaviour</SHORT-NAME>  
  <EVENTS>  
    <ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>  
      <SHORT-NAME>ServerCallReturns</SHORT-NAME>
```

```

    <START-ON-EVENT-REF>
      /pkg/Behaviour/CollectResult
    </START-ON-EVENT-REF>
    <ASYNCHRONOUS-SERVER-CALL-POINT-REF>
      /pkg/Behaviour/CallServer/AsyncCallPoint
    </ASYNCHRONOUS-SERVER-CALL-POINT-REF>
  </ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>
</EVENTS>
<RUNNABLE-ENTITY>
  <SHORT-NAME>CallServer</SHORT-NAME>
  <SERVER-CALL-POINTS>
    <ASYNCHRONOUS-SERVER-CALL-POINT>
      <SHORT-NAME>AsyncCallPoint</SHORT-NAME>
      <OPERATION-IREFS>
        <OPERATION-IREF>
          ...
        </OPERATION-IREF>
      </OPERATION-IREFS>
    </ASYNCHRONOUS-SERVER-CALL-POINT>
  </SERVER-CALL-POINTS>
  <SYMBOL>...</SYMBOL>
</RUNNABLE-ENTITY>
<RUNNABLE-ENTITY>
  <SHORT-NAME>CollectResult</SHORT-NAME>
  <SYMBOL> CalledWhenServerReturns </SYMBOL>
</RUNNABLE-ENTITY>
</INTERNAL-BEHAVIOR>

```

Where the interface MyCSInterface is defined as:

```

<CLIENT-SERVER-INTERFACE>
  <SHORT-NAME>CSInterface</SHORT-NAME>
  <OPERATIONS>
    <OPERATION-PROTOTYPE>
      <SHORT-NAME>Function</SHORT-NAME>
      <ARGUMENTS>
        <ARGUMENT-PROTOTYPE>
          <SHORT-NAME>Param1</SHORT-NAME>
          <TYPE-TREF>/types/MonthType</TYPE-TREF>
          <DIRECTION>IN</DIRECTION>
        </ARGUMENT-PROTOTYPE>
        <ARGUMENT-PROTOTYPE>
          <SHORT-NAME>Result</SHORT-NAME>
          <TYPE-TREF>/types/UInt8</TYPE-TREF>
          <DIRECTION>OUT</DIRECTION>
        </ARGUMENT-PROTOTYPE>
      </ARGUMENTS>
    </OPERATION-PROTOTYPE>
  </OPERATIONS>
</CLIENT-SERVER-INTERFACE>

```

The runnable to handle the results coming back from the server would have the follow-

ing structure:

```
#include Rte_MyComponent.h''

boolean February;

FUNC(void, RTE_APPL_CODE)
CalledWhenServerReturns(Rte_Instance Self)
{
    UInt8      Result;
    /* Collect and use the result from the server */
    Rte_StatusType err;

    err = Rte_Result_Client_MyFunction(Self, &Result);

    if (err == RTE_E_OK) {
        if (Result == 28 || Result == 29 ) {
            February = True;
        } else {
            February = False;
        };
    };
    return;
}
```

## 10.5 Sender-Receiver Communication

The RTE API calls for handling sender-receiver communication differ for the type of information that is being communicated:

- Non-queued communication
  - Send with `Rte_Write`
  - Receive with `Rte_Read`
- Queued communication
  - Send with `Rte_Send`
  - Receive with `Rte_Receive`

Furthermore, for non-queued data a secondary pair of API calls can be used to access the data called the “implicit” API. The implicit API should be used when you need to guarantee that every access to a data item in a runnable entity will provide the same result irrespective of how many times it is accessed during an invocation of the runnable entity.

- Non-queued data (Implicit)

- Send with Rte\_IWrite
- Read with Rte\_IRead

The following sections show how to use these calls in your application.

### 10.5.1 Non-Queued Communication

---

#### Explicit API

---

Components communicate data to other components using the Rte\_Write call. The call is defined per port and interface data item for each component and therefore has the following signature:

```
Rte_StatusType  
Rte_Write_<Port>_<DataItem>(Rte_Instance Self,  
                             DataItemType Data)
```

Similarly, components receive communicated data items from other components using the Rte\_Read call:

```
Rte_StatusType  
Rte_Read_<Port>_<DataItem>(Rte_Instance Self,  
                             DataItemType* Data)
```

The runtime behaviour of Rte\_Read depends on how you have configured the internal behaviour:

- **Data Read Access:** Rte\_Read is non-blocking even if no data is present to read. If no data is present, the return value from the call is RTE\_E\_NO\_DATA.
- **Activation of runnable entity:** The RTE activates the specified runnable entity and generates a non-blocking Rte\_Read.
- **Wake up of wait point:** This combination is not permitted.

#### Implicit API

---

The implicit API uses a locally cached copy of data to preserve consistency over a calling runnable entity invocation. Data is read into a global cache before the runnable entity starts executing and is written from the global cache after the runnable entity terminates. Data writes are done once, no matter how many times it is written.

The RTE guarantees cached data does not change during execution of the runnable entity.

The implicit API includes a reference to the runnable entity that is declared as accessing the data in the API name. Care should be taken when writing a runnable entity to invoke the correct API. The Rte\_IRead API reads data:

```
DataItemType  
Rte_IRead_<Runnable>_<Port>_<DataItem>(Rte_Instance Self)
```

The Rte\_IWrite API writes data

```
Rte_StatusType  
Rte_IWrite_<Runnable>_<Port>_<DataItem>(Rte_Instance Self,  
                                           DataItemType Data)
```

As explained above, the implicit API calls access a cached copy of the data. The cache is updated before the runnable entity starts and therefore within a single execution of a runnable entity the value returned by Rte\_IRead is guaranteed not to change. Likewise, Rte\_IWrite writes data to a cached copy and changes are only made visible after the runnable entity terminates irrespective of the number of times the data is written.

## 10.5.2 Queued Communication

Components communicate events to other components using the Rte\_Send call. The call is defined per port and interface data item for each component type and therefore has the following signature:

```
Rte_StatusType  
Rte_Send_<Port>_<DataItem>(Rte_Instance Self,  
                           EventType      Event)
```

Similarly, component prototypes receive events that are communicated from other components using the Rte\_Receive call:

```
Rte_StatusType  
Rte_Receive_<Port>_<DataItem>(Rte_Instance Self,  
                              EventType      Event)
```

The behaviour of the Rte\_Receive API depends on how you configured the RECEIVE\_MODE interface attributes:

- **Data Read Access:** Rte\_Receive is non-blocking even if no data is present to read. If no data is present, the return value from the call is RTE\_E\_NO\_DATA.
- **Activation of runnable entity** The RTE activates the specified runnable entity and generates a non-blocking Rte\_Receive.
- **Wake up of wait point:** Rte\_Receive blocks if no data is available. If no data is received within the specified timeout value, the return value from the call is RTE\_E\_TIMEOUT.

## 10.6 Client-Server Communication

Client-server communication is initiated using the `Rte_Call` API call. There are two forms that the call can take depending on the configuration of the `CLIENT_MODE` attribute.

If `CLIENT_MODE` is set to synchronous then `Rte_Call` returns after the operation has been completed by the server. This means that your code will not continue to execute until the server returns the result. Once the result has been computed it is passed back to the component by the return value of the `Rte_Call`.

```
Rte_StatusType
Rte_Call_<Port>_<Operation>(Rte_Instance Self,
                             InParam1Type InParam1,
                             InParam2Type InParam1,
                             ...
                             InParamNType InParamN,
                             OutParam1Type OutParam1,
                             OutParam2Type OutParam2,
                             ...
                             OutParamNType OutParamN)
```

If `CLIENT_MODE` is asynchronous then `Rte_Call` needs only specify the in and inout parameters.

```
Rte_StatusType
Rte_Call_<Port>_<Operation>(Rte_Instance Self,
                             InParam1Type InParam1,
                             InParam2Type InParam1,
                             ...
                             InParamNType InParamN)
```

The result(s) of the server operation are accessed using the asynchronous client result runnable entity that you define (see Section [8.7.2](#)) or the `Rte_Result_<port>_<operation>` API call.

The behaviour of the call returns depends on whether the `<ASYNCHRONOUS-SERVER-CALL-RETURNS-EVENT>` is referenced by a `<WAIT-POINT>` or not. If there is no wait point then the call returns immediately, otherwise the call will block until the server returns

```
Rte_StatusType
Rte_Result_<Port>_<Operation>(Rte_Instance Self,
                              OutParam1Type OutParam1,
                              OutParam2Type OutParam1,
                              ...
                              OutParamNType OutParamN)
```

### 10.6.1 Client-Server Serialization

---

The targets of client-server communications (i.e. the runnable entities that are invoked to produce the required result) are serialized. A serialized server only accepts and processes requests atomically and thus avoids the potential for conflicting concurrent access.

## 10.7 Using Inter-Runnable Variables

---

Runnable entities in software components that use inter-runnable variables access the variable through an API call. Since the API calls are generated in the software component's header file each API call must include the name of the runnable itself.

The type of API call generated depends on whether the variable is declared as implicit or explicit. Implicit variables are accessed through an API of the form:

```
Rte_IrvI[Read|Write]_<Runnable>_<Variable>(...);
```

Explicit variables are accessed with an API call of the form:

```
Rte_Irv[Read|Write]_<Runnable>_<Variable>(...);
```

The following code example illustrates usage:

```
FUNC(void, RTE_APPL_CODE)
RunnableA(Rte_Instance self)
{
    Unit16 ExplicitlyRead = 0;
    Unit32 ImplicitlyRead = 0;
    Sint8 ExplicitlyWritten = 42;
    Boolean ImplicitlyWritten = True;

    /* Read inter-runnable variables */
    ExplicitlyRead = Rte_IrvRead_RunnableA_VarP(self);
    ImplicitlyRead = Rte_IrvIRead_RunnableA_VarQ(self);

    /* Write inter-runnable variables */
    Rte_IrvWrite_RunnableA_VarR(self, ExplicitlyWritten);
    Rte_IrvIWrite_RunnableA_VarS(self, ImplicitlyWritten);
}
```

## 10.8 Accessing Parameters

---

A software component declares calibration parameters (whether shared or per-instance) then each parameter is accessed at runtime using the `Rte_CData_<calprm_name>` API call. The call returns either the calibration data (primitive types) or a pointer to the data (complex types).

As the data is specific to each software component instance, the instance handle must be supplied to each call if multiple instantiation is configured.



```
FUNC(void, RTE_APPL_CODE)
YourRunnable(Rte_Instance Self)
{
    uint8* config
    config = Rte_CData_TrimValue(Self);
    /* user code */
}
```

## 10.9 Accessing Per Instance Memory

---

Component types are standard programs. The RTE only interacts with components through ports at runtime by the activation of associated runnable entities. Ports are handled by defined runnable entities, so as a minimum an implementation of a software component must include an implementation of all runnable entities for the software component.

All instances of an application software component share the same code, but each instance may have private state that is not shared with other instances. Consequently, each software component instance has a “copy” of the private state to which it has exclusive access.

The `Rte_Pim` API is used to access the component state for a particular instance and, like all RTE API calls, takes the component instance handle as its first parameter.

The function returns a handle to a per-instance memory section. The type of the return value depends on the per-instance memory declaration.

The `Rte_Pim` call has the signature:

```
Rte_Pim_<name>(Rte_Instance Self)
```

The state information returned by `Rte_Pim` can be used by a component instance to store information with a lifetime greater than that of any one runnable entity. You should not use “traditional” global variables since these are shared by all prototypes of the component mapped to the ECU.

The `Rte_Pim` API does not impose concurrency control on a component’s prototype state and therefore concurrent write access to prototype state may result in inconsistent state information.

## 10.10 Concurrency Control with Exclusive Areas

---

Where a component has multiple runnable entities that require concurrent write access to the same prototype state then the `Rte_Enter` and `Rte_Exit` API calls must be used to ensure that data consistency is maintained.

A component includes multiple runnable entities each of which can be active simultaneously. The potential exists for concurrent access to private global data (e.g. elements in the data memory sections) and/or non-reentrant functions.

Operating system concurrency control mechanisms are hidden from components. However the RTE API implements explicit access to exclusive areas by exposing an appropriate OS mechanism to components:

- `Rte_Enter_<exclusive area name>` enters an exclusive area.
- `Rte_Exit_<exclusive area name>` exits an exclusive area.

Where components declare exclusive areas, the generated RTE API for the component includes these API calls to allow you to control concurrent access to shared data.

A component can use the `Rte_Enter` and `Rte_Exit` API calls for any exclusive area ID you define at configuration time.

For example, assuming the following configuration:

```
<INTERNAL-BEHAVIOR>  
  <EXCLUSIVE-AREAS>  
    <EXCLUSIVE-AREA>MyExclusiveArea</EXCLUSIVE-AREA>  
  </EXCLUSIVE-AREAS>  
</INTERNAL-BEHAVIOR>
```

The following code shows how the calls are used:

```
Rte_Enter_MyExclusiveArea();  
/* Code protected from concurrent execution */  
Rte_Exit_MyExclusiveArea();
```

It is not permitted to place RTE API calls between a call to `Rte_Enter` and its corresponding call to `Rte_Exit`.



*Attempting RTE API calls while an exclusive area is locked may result in delayed response to task activations by the OS.*

Software components must use the `Rte_Enter` and `Rte_Exit` calls in a strictly nested manner. Exclusive areas must be exited in the reverse order in which they were entered as shown below:

```
Rte_Enter_Outer();  
/* Code protected by Outer */  
  Rte_Enter_Inner();  
  /* Code protected by Inner & Outer */  
  Rte_Exit_Inner();  
  /* Code protected by Outer */  
Rte_Exit_Outer();
```



*The scope of an exclusive area is the software component prototype and not the software component type or system wide and hence exclusive areas only provide concurrency control within a software component. Wider scope can be achieved using an AUTOSAR component to control access to shared data.*

## 10.11 Starting and Stopping the RTE

The RTE itself is started using the `Rte_Start` API call. You must not make this call from a software component.

`Rte_Start` is typically called from the idle task since this ensures that runnable entities triggered on entry to the initial mode are activated and dispatched (based on the task priority to which they are mapped) before the API returns. For example:

```
OS_MAIN()  
{  
    /* Initialize hardware */  
    StartOS(OSDEFAULTAPPMODE);  
  
    /* RTA-OSEK: OS_MAIN is now the idle task */  
    Rte_Start();  
    for (;;) {  
        {  
            /* Do background processing */  
        }  
    }  
}
```

The `Rte_Start` call returns to the callee once the RTE is initialized.



*The RTE must not be started before the operating system. If not using the ECU state manager, the recommended start order is to initialize/start the lowest elements of the AUTOSAR architecture (e.g. OS) first and then proceed “upwards” (i.e. communication drivers, PDU Router, COM and finally RTE).*

The RTE is stopped by calling the `Rte_Stop` API call. This API call stops any operating system schedules that release time-triggered runnable entities and then calls any software component finalization runnable entities that you have defined.



*The RTE must be stopped before the operating system. The recommended shutdown order is to stop the higher elements of the AUTOSAR architecture (e.g. RTE) first and then proceed “downwards” (i.e. RTE, COM, PDU Router, communication drivers and finally OS).*

## 11 NVRAM

---

### 11.1 NV-Block Software Component Types

---

In Chapter 7 we discussed how AUTOSAR application software is partitioned using AUTOSAR software components. The use of NVRAM within AUTOSAR adds another software component type to the list of supported types, an *NvBlock software component type*, that can declare ports used for NVRAM communication.

An Nv-Block software component type is described using a `<NV-BLOCK-SW-COMPONENT-TYPE>` element. This element describes the component type name, its ports and Nv-blocks, for example:

```
<NV-BLOCK-SW-COMPONENT-TYPE>
  <SHORT-NAME>nv1</SHORT-NAME>
  <PORTS>
    ...
  </PORTS>
  <NV-BLOCK-DESCRIPTORS>
    ...
  </NV-BLOCK-DESCRIPTORS>
  <INTERNAL-BEHAVIORS>
    ...
  </INTERNAL-BEHAVIORS>
</NV-BLOCK-SW-COMPONENT-TYPE>
```

#### 11.1.1 Ports

---

Like an application software component, an NvBlock software component type declares ports for communication with other component instances. However, unlike a general application SWC, ports within an NvBlock SWC can only be categorized by the following interface types:

**Nv-Data** – permits read and/or write access by application software components to a ram-block within a Nv-block. A block mapping (see below) is used to associate the port and the ram-block.

**Client-Server** – permits notifications from the NVRAM manager to application SWCs and for the invocation of NVRAM manager API functions by application SWCs.

RTA-RTE will raise an error if any other interface types is used to categorize a port declared by an Nv-Block software component type.

#### 11.1.2 Nv-Blocks

---

A Nv-Block software component type declares one or more Nv-Blocks using the `<NV-BLOCK-DESCRIPTOR>` element. Each block declares a RAM-based mirror of non-volatile data.


```
<NV-BLOCK-DESCRIPTOR>
  <SHORT-NAME>block1</SHORT-NAME>
```

```
<CLIENT-SERVER-PORTS>
...
</CLIENT-SERVER-PORTS>
<NV-BLOCK-DATA-MAPPINGS>
...
</NV-BLOCK-DATA-MAPPINGS>
<RAM-BLOCK>
  <SHORT-NAME>myRamBlock2</SHORT-NAME>
...
</RAM-BLOCK>
</NV-BLOCK-DESCRIPTOR>
```

Each NvBlock descriptor must declare:

1. Exactly one `ram-block` element. The `ram-block` is instantiated by RTA-RTE and acts as the mirror of the NVRAM data that is read and/or written by application SWC.
2. Zero or more *NvBlock Data Mappings* that connect ports categorized by Nv-Data interfaces with the `ram-block`.
3. Zero or more *Client-Server Ports* that connect ports categorized by client-server interfaces, the `ram-block` and the NVRAM manager.

In addition, an Nv-block descriptor can optionally describe a ROM initializer for the `ram-block`.

 *ROM initializers are partially supported by this release of RTA-RTE. The `rom-block` is optionally used by RTA-RTE to initialize the `ram-block` during `Rte_Start`.*

### 11.1.3 NVRAM Mirrors

Each Nv-block descriptor within an Nv-Block SWC declares exactly one *ram-block*. The `ram-block` forms the data that is read and/or written by application SWC through ports declared by the Nv-Block SWC.

RTA-RTE provides APIs for the NVRAM manager to initially set the mirror data before the RTE is started and to get the final mirror state after the RTE is terminated.

The `ram-block` instance is created by the RTE generator. The `ram-block` instance name is based on the `ram-block` name and therefore each block must have a unique name within the context of the NvBlock software component type.'

### 11.1.4 Port Mappings

A port within an Nv-Block SWC categorized by an Nv-Data interface (see Section 6.2) can be connected to a `ram-block` so that it can be read/written through the port. This connection is achieved through *Nv-Data mappings* declared within the Nv-block descriptor:

```

<NV-BLOCK-DATA-MAPPING>
  <NV-RAM-BLOCK-ELEMENT>
    <LOCAL-VARIABLE-REF ...
  </NV-RAM-BLOCK-ELEMENT>
  <READ-NV-DATA>
  ...
</READ-NV-DATA>
  <WRITTEN-NV-DATA>
  ...
</WRITTEN-NV-DATA>
</NV-BLOCK-DATA-MAPPING>

```

The mapping's <NV-RAM-BLOCK-ELEMENT> references the ram-block. RTA-RTE supports a reference made using, for example, the <LOCAL-VARIABLE-REF> element.

The connection with the port and Nv-data element is through either the <WRITTEN-NV-DATA> element for rPorts and the <READ-NV-DATA> element for pPorts. For example:

```

<WRITTEN-NV-DATA>
  <AUTOSAR-VARIABLE-IREF>
    <PORT-PROTOTYPE-REF ...
    <TARGET-DATA-PROTOTYPE-REF ...
  </AUTOSAR-VARIABLE-IREF>
</WRITTEN-NV-DATA>

```

A data mapping element can declare both read and write mappings for the same ram-block. Since read mappings reference pPorts and write mappings reference rPorts then, clearly, different ports must be specified for the mappings.

### 11.1.5 Fan-out and Fan-in

RTA-RTE does **not support** the use of data mappings to specify data fan-out from one rPort to multiple ram-blocks – it is not possible to write the same data to multiple Nv-Block mirrors from a single rPort.

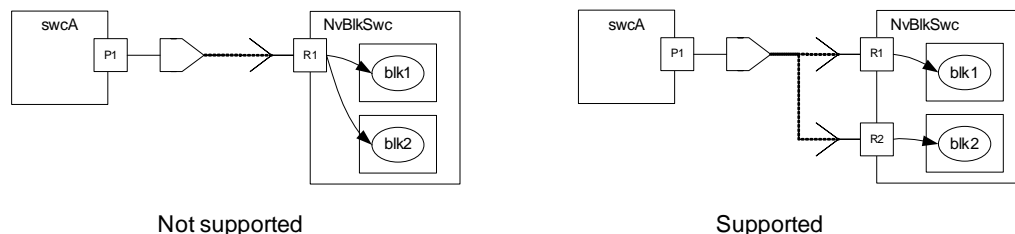


Figure 11.1: Supported and unsupported fan-out of NVRAM communication

Likewise the fan-in from multiple ram-blocks to a single pPort is **not supported** since it is not possible to collapse the contents of multiple disparate ram-blocks onto a single data read.

However RTA-RTE does support multiple rPorts being mapped to the same ram-block

(sub-element) and hence being written simultaneously by multiple sources. This is required to support the mapping of a port to an element of a ram-block when the latter uses a complex (record or array) type.

## 11.2 Interaction with Application SWC

---

Having created an Nv-Block software component type with a declared Nv-block and data mappings it needs to be accessed by application SWC. This access occurs between ports within the SWCs:

- An application SWC writes data to an Nv-block by declaring a Sender-receiver or Nv-data pPort (with an appropriate interface) and connecting it to an rPort on the Nv-Block SWC categorized by an Nv-Data interface.
- An application SWC reads data from an Nv-block by declaring a Sender-receiver or Nv-data rPort (with an appropriate interface) and connecting it to a pPort on the Nv-Block SWC categorized by an Nv-Data interface.

Accessing NVRAM from application SWC is considered in detail in Chapter 16.

## 11.3 Interaction with the NVRAM Manager

---

The NVRAM manager can interact with an NvBlock SWC, and hence with application SWCs, via API functions and runnable entity bodies generated by RTA-RTE.

Two API functions are generated by RTA-RTE for each NvBlock descriptor:

**Rte\_GetMirror** – Generated for each NvBlock descriptor and enables the NVRAM manager to read the contents of the ram-block.

**Rte\_SetMirror** – Generated for each NvBlock descriptor and enables the NVRAM manager to write the contents of the ram-block.

In addition to the Rte\_GetMirror and Rte\_SetMirror APIs, RTA-RTE can also generate additional APIs and runnable entity bodies based on declared *role based port assignments* depending on whether the assignment references a provided or required client-server port:

**Require port** – RTA-RTE generates an API to be invoked by the NVRAM manager. The name of the API is based on the declared *role*, the NvBlock SWC component prototype name and the NvBlock descriptor name.

The generated API invokes the server through the referenced require port. If multiple role based assignments have the same “role” then the generated API aggregates invocation of all referenced servers.

The generated API invokes servers synchronously. To avoid use of OsEvents in the generated code it is strongly recommended to use direct function invocation by omitting the task reference the relevant RTE event mapping.

**Provide port** – RTA-RTE generates the runnable entity body for all *connected* provide ports that have a configured `OperationInvokedEvent`.

The generated runnable entity body uses the symbol name from the runnable entity declaration – this must therefore be globally unique. The formal argument list consists of three elements:

1. An optional instance handle (included only when the `NvBlock SWC` is declared as supporting multiple instances, see Section 8.14).
2. The port-defined arguments (for the referenced server port, see Section 8.13.2).
3. The operation arguments from the operation in the server’s ports interface with the **same name** as the “role”.

The generated body for the runnable invokes the C-based API of the NVRAM manager based on the declared “role”. All runnable parameters, apart from the instance handle, are passed directly to the invoked NVRAM API.



## **Part III**

# **Developing Basic Software**

## 12 Basic Software

---

### 12.1 Basic Software Modules

---

In Chapter 7 we discussed how application software is partitioned using AUTOSAR software components. Similarly, AUTOSAR basic software (BSW) is partitioned into *BSW modules*.

A BSW module is closely related to a software component, and, just like a software component, has both an internal behaviour and an implementation. The module is described using a <BSW-MODULE-DESCRIPTION> element. This element describes the module name and its internal behaviour, for example:

```
<BSW-MODULE-DESCRIPTION>
  <SHORT-NAME>Can</SHORT-NAME>
  <MODULE-ID>23</MODULE-ID>
  <INTERNAL-BEHAVIORS>
    <BSW-INTERNAL-BEHAVIOR>
      ...
    </BSW-INTERNAL-BEHAVIOR>
  </INTERNAL-BEHAVIORS>
</BSW-MODULE-DESCRIPTION>
```

The module's name and, optionally the BSW implementation's <VENDOR-ID> and <VENDOR-API-INFOX>, are used when RTA-RTE generates API names for the module. The combination ensures that each module instance receives unique generated APIs.

*AUTOSAR configurations can specify multiple internal behaviors for a BSW module and then select the appropriate behavior using pre-build variability. This release of RTA-RTE does not support pre-build variability and therefore **at most one** internal behavior can be specified.*

#### 12.1.1 Internal Behaviour

---

A basic software module's internal behaviour is similar to the SWC's internal behavior described in Chapter 8 however reflecting the differing requirements different XML tags are used so specifying conceptually similar elements.

The internal behavior describes one or more <BSW-SCHEDULABLE-ENTITY>s which are analogous to runnable entities in a SWC and one or more BSW events that are analogous to RTE events. RTA-RTE supports the following BSW events:

**BSW-TIMING-EVENT** – Specifies a periodic activity within a basic software module.

**BSW-BACKGROUND-EVENT**

**BSW-EXTERNAL-TRIGGER-OCCURRED-EVENT** – permits triggering of a schedulable entity by an external SWC or BSW instance. RTA-RTE supports synchronization of BSW and SWC trigger instances enabling either SWCs to activate schedulable entities or BSWs to activate runnable entities in response to a external trigger occurred event.

**BSW-INTERNAL-TRIGGER-OCCURRED-EVENT** – permits triggering of a schedulable entity by the same BSW instance.

**BSW-MODE-SWITCH-EVENT** – enables BSW schedule entities to be activated in response to a mode switch. RTA-RTE supports synchronization of BSW and SWC mode instances enabling either SWCs to activate schedulable entities or BSWs to activate runnable entities in response to a mode switch event.

**BSW-MODE-SWITCHED-ACK-EVENT** – enables a BSW schedule entity to be activated when a mode switch is complete.

These events and the interaction of basic software modules with the RTE are discussed in this chapter.

## 12.2 Interaction with the RTE

The mechanisms by which BSW modules interact with the RTE and through it with other application elements are closely related to the facilities that are available for software components. In fact, apart from not possessing ports, many of the RTE features for software components have direct analogues in BSW modules.

This section explains the facilities RTA-RTE provides for BSW modules.

### 12.2.1 Schedulable Entity

A BSW schedulable entity is closely related to a software component's runnable entity since they are both representations of an entry point within the application software that can be started executing by the RTE.

```
<BSW-SCHEDULABLE-ENTITY>  
  <SHORT-NAME>BSWTicked</SHORT-NAME>  
  <IMPLEMENTED-ENTRY-REF DEST='...'>/pkg/Ticked</IMPLEMENTED-ENTRY-REF>  
</BSW-SCHEDULABLE-ENTITY>
```

The <IMPLEMENTED-ENTRY-REF> references a <BSW-MODULE-ENTRY> that defines the entry point for the BSW schedulable entity. For example:

```
<BSW-MODULE-ENTRY>  
  <SHORT-NAME>Ticked</SHORT-NAME>  
  <IS-REENTRANT>>true</IS-REENTRANT>  
  <IS-SYNCHRONOUS>>true</IS-SYNCHRONOUS>  
  <CALL-TYPE>SCHEDULED</CALL-TYPE>  
  <EXECUTION-CONTEXT>TASK</EXECUTION-CONTEXT>  
  <SW-SERVICE-IMPL-POLICY>STANDARD</SW-SERVICE-IMPL-POLICY>  
</BSW-MODULE-ENTRY>
```

The BSW module entry defines the execution context of the schedulable entity which should be TASK. BSW module entries are not defined within the context of the BSW module description and thus can be in a different AUTOSAR package if necessary.

Multiple BSW modules can reference the **same** BSW module entry, e.g. multiple modules can each reference the same MainFunction module entry. RTA-RTE will use the

defined BSW module name, *vendor API infix* and *module ID* to ensure that each module implementation receives a unique API.

A BSW schedulable entity is started by the RTE triggering a BSW event. RTA-RTE supports multiple event types for BSW schedulable entities including time and mode switches.

### 12.2.2 Periodic Events

---

A BSW schedulable entity is started periodically using a <BSW-TIMING-EVENT>. Similarly to an SW-C's <TIMING-EVENT> the BSW event defines a period (in seconds) and an optional minimum start interval (also in seconds).

```
<BSW-TIMING-EVENT>  
  <SHORT-NAME>BSWTick</SHORT-NAME>  
  <STARTS-ON-EVENT-REF DEST='...'>/pkg/mod/ib/entity</STARTS-ON-EVENT-REF>  
  <PERIOD>1.0</PERIOD>  
</BSW-TIMING-EVENT>
```

The <STARTS-ON-EVENT-REF> references the schedule entity to be started when the trigger occurs. It must therefore reference an entity within the basic software module description. For clarity the DEST attribute has been omitted in the example above but would normally have the value BSW-SCHEDULABLE-ENTITY.

### 12.2.3 Background Events

---

RTA-RTE supports BSW background events. As with <BSW-TIMING-EVENT>s these events are directly analogous to an SW-C's <BACKGROUND-EVENT>.

### 12.2.4 External Trigger Events

---

Section 8.8 covered the configuration of internal and external trigger events for SW-Cs. BSW modules can trigger such events and also define responses that are triggered by either SW-Cs or by other BSW modules.

A basic software module declares an external trigger event using the <BSW-EXTERNAL-TRIGGER-OCCURRED-EVENT> event:

```
<BSW-EXTERNAL-TRIGGER-OCCURRED-EVENT>  
  <SHORT-NAME>BSWTrigger</SHORT-NAME>  
  <STARTS-ON-EVENT-REF DEST='...'>/pkg/bsw/ib/entity</STARTS-ON-EVENT-REF>  
  <TRIGGER-REF DEST='...'>/pkg/bsw/trigger</TRIGGER-REF>  
</BSW-EXTERNAL-TRIGGER-OCCURRED-EVENT>
```

The short name of the external trigger event is only significant for referencing and does not form part of the generated API.

The <STARTS-ON-EVENT-REF> references the schedule entity to be started when the trigger occurs. It must therefore reference an entity within the basic software module description. For clarity the DEST attribute has been omitted in the example above but would normally have the value BSW-SCHEDULABLE-ENTITY.

The <TRIGGER-REF> references a required <TRIGGER> declared within the <BSW-MODULE-DESCRIPTION>. For clarity the DEST attribute has been omitted in the example above but would normally have the value TRIGGER.

```
<REQUIRED-TRIGGERS>
  <TRIGGER>
    <SHORT-NAME>reqTrigger</SHORT-NAME>
  </TRIGGER>
</REQUIRED-TRIGGERS>
```

Multiple external trigger events can reference the same required <TRIGGER> and RTA-RTE will ensure that all are referenced schedulable entities are started when the trigger fires.

The source for an external trigger is a schedulable entity that references a release <TRIGGER>.

```
<BSW-SCHEDULABLE-ENTITY>
  ...
  <ISSUED-TRIGGERS>
    <TRIGGER-REF-CONDITIONAL>
      <TRIGGER-REF DEST='TRIGGER'>/pkg/mod/relTrigger</TRIGGER-REF>
    </TRIGGER-REF-CONDITIONAL>
  </ISSUED-TRIGGERS>
</BSW-SCHEDULABLE-ENTITY>
```

The short name of the referenced release <TRIGGER> is used to form the SchM\_Trigger API name.

The connection between the release and require triggers is made within the ECUC, see Section 14.3.1 for details. In addition, both released and required BSW external triggers can be synchronized with SWC external triggers, see Section 15.3.

### 12.2.5 Internal Trigger Events

Section 8.8 covered the configuration of internal and external trigger events for SW-Cs. BSW modules can trigger such events and also define responses that are triggered by either SW-Cs or by other BSW modules.

A basic software module declares an internal trigger event using the <BSW-INTERNAL-TRIGGER-OCCURRED-EVENT> event:

```
<BSW-INTERNAL-TRIGGER-OCCURRED-EVENT>
  <SHORT-NAME>BSWTrigger</SHORT-NAME>
  <STARTS-ON-EVENT-REF DEST='...'>/pkg/mod/ib/entity</STARTS-ON-EVENT-REF>
  <EVENT-SOURCE-REF DEST='...'>/pkg/mod/ib/trigPoint</EVENT-SOURCE-REF>
</BSW-INTERNAL-TRIGGER-OCCURRED-EVENT>
```

The short name of the internal trigger event is only significant for referencing and does not form part of the generated API.

The `<STARTS-ON-EVENT-REF>` references the schedule entity to be started when the trigger occurs. It must therefore reference an entity within the basic software module description. For clarity the `DEST` attribute has been omitted in the example above but would normally have the value `BSW-SCHEDULABLE-ENTITY`.

The `<EVENT-SOURCE-REF>` references a `<BSW-INTERNAL-TRIGGERING-POINT>` declared within the `<BSW-MODULE-DESCRIPTION>`. For clarity the `DEST` attribute has been omitted in the example above but would normally have the value `BSW-INTERNAL-TRIGGERING-POINT`.

```
<INTERNAL-TRIGGERING-POINTS>
  <BSW-INTERNAL-TRIGGERING-POINT>
    <SHORT-NAME>trigPoint</SHORT-NAME>
  </BSW-INTERNAL-TRIGGERING-POINT>
</INTERNAL-TRIGGERING-POINTS>
```

Multiple internal trigger events can reference the same required `<BSW-INTERNAL-TRIGGERING-POINT>` and RTA-RTE will ensure that all are referenced schedulable entities are started when the trigger fires.

Internal triggers must have a source within the BSW module. The source schedule entity references the `<BSW-INTERNAL-TRIGGERING-POINT>` using a `<BSW-INTERNAL-TRIGGERING-POINT-REF-CONDITIONAL>` element:

```
<ACTIVATION-POINTS>
  <BSW-INTERNAL-TRIGGERING-POINT-REF-CONDITIONAL>
    <BSW-INTERNAL-TRIGGERING-POINT-REF DEST='...'>/pkg/mod/ib/trigPoint</
      BSW-INTERNAL-TRIGGERING-POINT-REF>
  </BSW-INTERNAL-TRIGGERING-POINT-REF-CONDITIONAL>
</ACTIVATION-POINTS>
```

The `<BSW-INTERNAL-TRIGGERING-POINT-REF>` references a `<BSW-INTERNAL-TRIGGERING-POINT>` declared within the `<BSW-MODULE-DESCRIPTION>`. For clarity the `DEST` attribute has been omitted in the example above but would normally have the value `BSW-INTERNAL-TRIGGERING-POINT`.

## 12.2.6 Mode Switch Events

Chapter 9 described how a SW-C accessed modes through a provided or required categorized by an interface containing mode declaration group prototypes. Basic software does not have ports however accessed modes are still described using mode declaration group prototypes but for basic software these are contained within the BSW module description.

```
<BSW-MODULE-DESCRIPTION>
  <SHORT-NAME>Can</SHORT-NAME>
  ...
  <PROVIDED-MODE-GROUPS>
    <MODE-DECLARATION-GROUP-PROTOTYPE>
      <SHORT-NAME>bmdgp</SHORT-NAME>
      <TYPE-TREF DEST='...'>/pkg/mdg1</TYPE-TREF>
```

```
</MODE-DECLARATION-GROUP-PROTOTYPE>  
</PROVIDED-MODE-GROUPS>
```

The <TYPE-TREF> references a previously declared <MODE-DECLARATION-GROUP>. For clarity the DEST attribute has been omitted in the example above but would normally have the value MODE-DECLARATION-GROUP.

A schedulable entity declares itself as the mode manager using a reference within <MANAGED-MODE-GROUPS> to the provided mode group prototype.

```
<MANAGED-MODE-GROUPS>  
  <MODE-DECLARATION-GROUP-PROTOTYPE-REF-CONDITIONAL>  
    <MODE-DECLARATION-GROUP-PROTOTYPE-REF DEST='...'>/pkg/mod/bmdgp</MODE-  
      DECLARATION-GROUP-PROTOTYPE-REF>  
  </MODE-DECLARATION-GROUP-PROTOTYPE-REF-CONDITIONAL>  
</MANAGED-MODE-GROUPS>
```

The <MODE-DECLARATION-GROUP-PROTOTYPE-REF> references the previously declared <MODE-DECLARATION-GROUP-PROTOTYPE>. For clarity the DEST attribute has been omitted in the example above but would normally have the value MODE-DECLARATION-GROUP-PROTOTYPE.

A BSW mode manager can optionally specify the length of the mode switch queue, see Section 9.2.2 for details. If omitted, RTA-RTE assumes a queue length of one.

A mode user requires access to a mode and this is configured through the <REQUIRED-MODE-GROUPS> within a basic software module description.

```
<REQUIRED-MODE-GROUPS>  
  <MODE-DECLARATION-GROUP-PROTOTYPE>  
    <SHORT-NAME>bmdgr2</SHORT-NAME>  
    <TYPE-TREF DEST='...'>/pkg/mdg1</TYPE-TREF>  
  </MODE-DECLARATION-GROUP-PROTOTYPE>  
</REQUIRED-MODE-GROUPS>
```

The <TYPE-TREF> references a previously declared <MODE-DECLARATION-GROUP>. For clarity the DEST attribute has been omitted in the example above but would normally have the value MODE-DECLARATION-GROUP.

A schedulable entity can be triggered as a result of a mode switch using a <BSW-MODE-SWITCH-EVENT> that references a required mode group prototype using a <CONTEXT-MODE-DECLARATION-GROUP-REF>.

```
<BSW-MODE-SWITCH-EVENT>  
  <SHORT-NAME>BSWmodeSwitchEvent</SHORT-NAME>  
  <STARTS-ON-EVENT-REF DEST='...'>/pkg/mod/ib/BSWmse</STARTS-ON-EVENT-REF>  
  <MODE-IREFS>  
    <MODE-IREF>  
      <CONTEXT-MODE-DECLARATION-GROUP-REF DEST='...'>/pkg/mod/bmdgr2</  
        CONTEXT-MODE-DECLARATION-GROUP-REF>  
      <TARGET-MODE-REF DEST='MODE-DECLARATION'>/pkg/mdg1/implicit</TARGET-  
        MODE-REF>
```

```

    </MODE-IREF>
  </MODE-IREFS>
</BSW-MODE-SWITCH-EVENT>

```

The <STARTS-ON-EVENT-REF> references the schedule entity to be started when the mode switch occurs. It must therefore reference an entity within the basic software module description. For clarity the DEST attribute has been omitted in the example above but would normally have the value BSW-SCHEDULABLE-ENTITY.

The connection between the provided and requires mode group prototypes is made within the ECUC, see Section 14.3.2 for details. In addition, both provided and required BSW mode declaration groups can be synchronized with provided SWC mode declaration groups, see Section 15.2.

### 12.2.7 Mode Switch Ack Events

The configuration of mode switch acknowledge events within basic software is similar to the process described above for mode switched events. The only significant difference is how schedule entities are triggered using a <BSW-MODE-SWITCHED-ACK-EVENT> element rather than a <BSW-MODE-SWITCH-EVENT> element.

```

<BSW-MODE-SWITCHED-ACK-EVENT>
  <SHORT-NAME>BSWmodeSwitchAckEvent</SHORT-NAME>
  <STARTS-ON-EVENT-REF DEST='...'>/pkg/mod/ib/BSWmsae</STARTS-ON-EVENT-REF>
  <MODE-GROUP-REF DEST='...'>/pkg/mod/bmdgp</MODE-GROUP-REF>
</BSW-MODE-SWITCHED-ACK-EVENT>

```

The <STARTS-ON-EVENT-REF> references the schedule entity to be started when the mode switch is acknowledged. It must therefore reference an entity within the basic software module description. For clarity the DEST attribute has been omitted in the example above but would normally have the value BSW-SCHEDULABLE-ENTITY.

The connection between the provided and requires mode group prototypes is made within the ECUC, see Section 14.3.2 for details.

## 12.3 API

RTA-RTE forms the names of generated BSW API functions using a the module's name which is optionally combined with each BSW implementation's <VENDOR-ID> and <VENDOR-API-INFIX>. Each generated name, therefore, as the following form:

```
SchM_<root>_<mp>[_<vi>_<ai>]_<name>
```

Where <mp> is the the name of the BSW module (AUTOSAR terms this the "module prefix"), <vi> the <VENDOR-ID> and <ai> the <VENDOR-API-INFIX>. The <root> and <name> components of the API name vary depending on the API.

For example, assuming the following BSW module description and BSW implementation:

```
<BSW-MODULE-DESCRIPTION>
```



```
<SHORT-NAME>Can</SHORT-NAME>
...
</BSW-MODULE-DESCRIPTION>

<BSW-IMPLEMENTATION>
  <SHORT-NAME>MyCanDrv</SHORT-NAME>
  <VENDOR-ID>25</VENDOR-ID>
  ...
  <VENDOR-API-INFIX>Dev0815</VENDOR-API-INFIX>
</BSW-IMPLEMENTATION>
```

RTA-RTE would generate API names of the form SchM\_<root>\_Can\_25\_Dev0815\_<name>.

The specification of the <VENDOR-ID> and <VENDOR-API-INFIX> elements are optional. AUTOSAR defines that if the <VENDOR-API-INFIX> is omitted then both the <VENDOR-ID> and <VENDOR-API-INFIX> elements are omitted from the generated names.

# **Part IV**

# **Composition**

## 13 Composing Software Components

---

In Part II you saw how to define software components, their internal behaviour and their implementation characteristics. In this chapter you will see how to assemble, or compose, applications from the components you have defined (or that have been defined for you).

AUTOSAR uses a <COMPOSITION-TYPE> to define which prototypes, of which software component types, you will have in your system and how their ports are connected at the abstract level. Each composition therefore represents a logical assembly of interacting software components.

Each composition creates a logical software system for deployment on one or more ECUs. Every system that you need to deploy will have at least one “master” composition that defines how all software component prototypes are connected as shown below.

However, it may be the case you need to define one or more compositions to logically partition your vehicle system. For example, you might define compositions for body, chassis and powertrain. This will allow you to define how these groups of component prototypes are deployed if you need to allocate components from some applications onto specific ECUs.

### 13.1 Composition Type Definition

---

All composition information is contained in an <AR-PACKAGE> element.

```
<AR-PACKAGE>
  <SHORT-NAME>MyComposition</SHORT-NAME>
  <ELEMENTS>
    <COMPOSITION-TYPE>
      ...
    </COMPOSITION-TYPE>
  </ELEMENTS>
</AR-PACKAGE>
```

The <COMPOSITION-TYPE> element is used to instantiate one or more software component types and the connections between the component prototypes.

```
<COMPOSITION-TYPE>
  <SHORT-NAME>MyComposition</SHORT-NAME>
  <PORTS>
    ...
  </PORTS>
  <COMPONENTS>
    ...
  </COMPONENTS>
  <CONNECTORS>
    ...
  </CONNECTORS>
</COMPOSITION-TYPE>
```

The <COMPONENTS> element is used to instantiate one or more SWC-type definitions. The <CONNECTORS> element is used to create connections between ports defined on component prototypes. The <PORTS> element is used to create ports for use with delegation connectors.

A composition defines a group of connected software component prototypes and provides a mechanism for hierarchical composition of systems. However, the individual component prototypes within the composition can be allocated onto different ECU instances at system definition time (see Section 17.1).

## 13.2 Component Instances

---

The <COMPONENTS> element within a <COMPOSITION-TYPE> is used to instantiate one or more SWC-type definitions.

Each component type can be instantiated multiple times, but each instance must be given a unique name.

The <SW-COMPONENT-PROTOTYPE> element provides a name for the component instance and provides a reference to the type description. The following description:

```
<COMPONENTS>
  <SW-COMPONENT-PROTOTYPE>
    <SHORT-NAME>John</SHORT-NAME>
    <TYPE-TREF>/pkg/PersonType</TYPE-TREF>
  </SW-COMPONENT-PROTOTYPE>

  <SW-COMPONENT-PROTOTYPE>
    <SHORT-NAME>Jane</SHORT-NAME>
    <TYPE-TREF>/pkg/PersonType</TYPE-TREF>
  </SW-COMPONENT-PROTOTYPE>

  <SW-COMPONENT-PROTOTYPE>
    <SHORT-NAME>Cafe</SHORT-NAME>
    <TYPE-TREF>/pkg/BusinessType</TYPE-TREF>
  </SW-COMPONENT-PROTOTYPE>
</COMPONENTS>
```

Creates the instances shown in Figure 13.1.

## 13.3 Connector Prototypes

---

As well as defining component prototypes, a <COMPOSITION-TYPE> element includes the <ASSEMBLY-CONNECTOR> definitions that connect ports between component prototypes and the <DELEGATION-CONNECTOR> elements that connect ports within a composition to elements outside the composition.

### 13.3.1 Assembly Connectors

---

An assembly connector is used to connect provided and required ports within a composition.

```
<CONNECTORS>
```

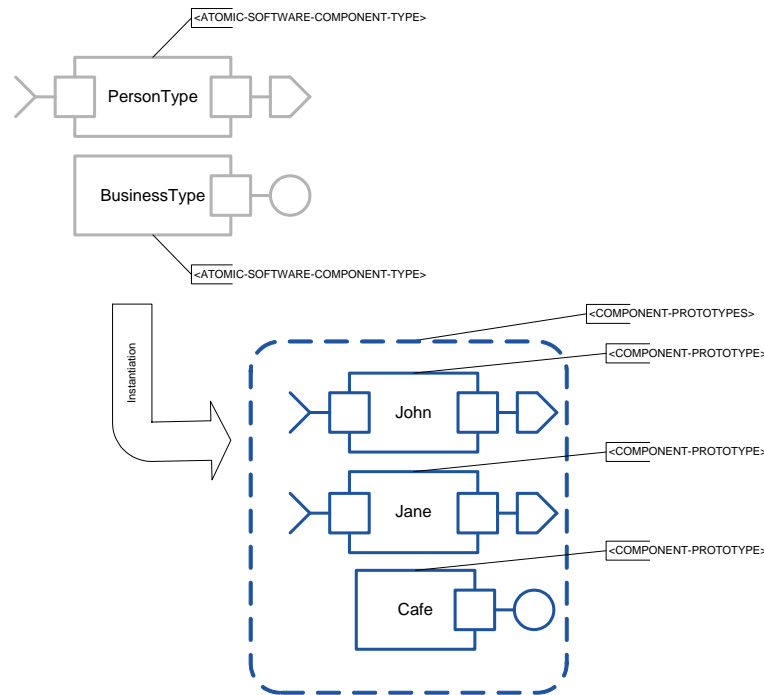


Figure 13.1: Creating Component Instances

```
<ASSEMBLY-SW-CONNECTOR>
...
</ASSEMBLY-SW-CONNECTOR>
</CONNECTORS>
```

Each assembly connector prototype defines one connection between a provided and required port on two component prototypes. Thus each connector prototype creates a link from a single port data item or operation source to a single destination. If either multiple sources or destinations are required then multiple connectors must be defined.

Figure 13.2 shows a producer software component instance being connected to a consumer software component instance:

The provided port is referenced within the connector using the <PROVIDER-IREF> element and the required port using the <REQUESTER-IREF> element.

```
<ASSEMBLY-CONNECTOR-PROTOTYPE>
  <SHORT-NAME>MyConnector</SHORT-NAME>
  <PROVIDER-IREF>
    <CONTEXT-COMPONENT-REF ...
    <TARGET-P-PORT-REF ...
  </PROVIDER-IREF>
  <REQUESTER-IREF>
    <CONTEXT-COMPONENT-REF ...
    <TARGET-R-PORT-REF ...>
  </REQUESTER-IREF>
</ASSEMBLY-CONNECTOR-PROTOTYPE>
```

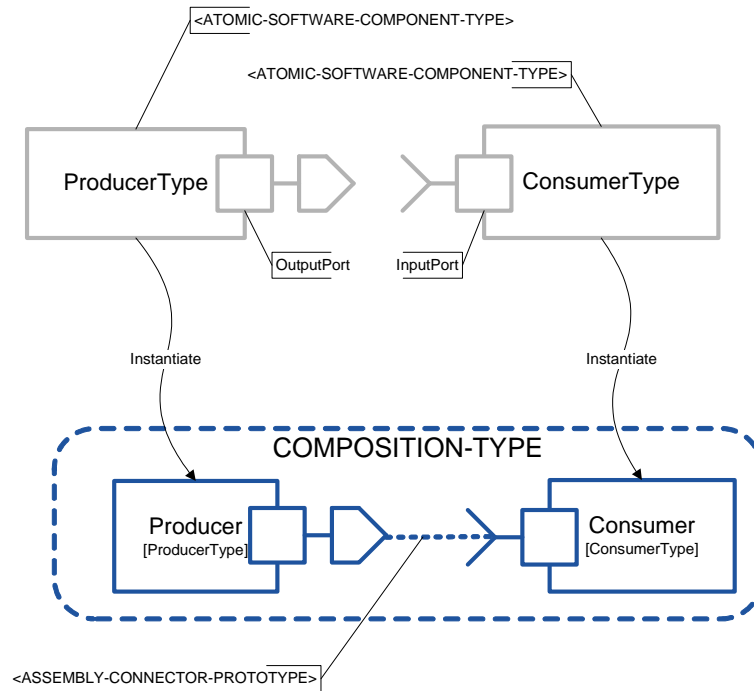


Figure 13.2: Assembly Connectors

Note that since the connection is made between different component prototypes then each component prototype can have different connectors defined.

### 13.3.2 Delegation Connectors

A delegation connector is used to connect a port within a composition to a port on the composition boundary. The composition port can then be used to connect compositions together using an assembly connector.

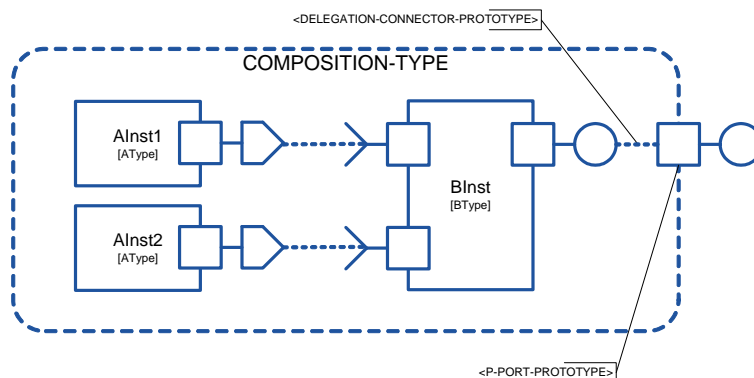


Figure 13.3: Delegation Connectors

To create and use a delegation connector, one must first create a port prototype within the composition type — this port will form one “end” of the connector. The port is created in a form that is identical when port prototypes are created within a software component.

```

<COMPOSITION-TYPE>
...
<PORTS>
  <P-PORT-PROTOTYPE>
    <SHORT-NAME>ServerPortonComposition</SHORT-NAME>
    <PROVIDED-INTERFACE-TREF>/interface/one_op</PROVIDED-INTERFACE-TREF>
  </P-PORT-PROTOTYPE>
  ...
</PORTS>
...
</COMPOSITION-TYPE>

```

Once the port-prototype has been defined the delegation connector can be created. A <DELEGATION-CONNECTOR-PROTOTYPE> element references a port on a component prototype and the port on the composition type:

```

<DELEGATION-CONNECTOR-PROTOTYPE>
  <SHORT-NAME>ServerPortDelegator</SHORT-NAME>
  <INNER-PORT-IREF>
    <CONTEXT-COMPONENT-REF ...
    <PORT-PROTOTYPE-REF ...
  </INNER-PORT-IREF>
  <OUTER-PORT-REF ...
</DELEGATION-CONNECTOR-PROTOTYPE>

```

The <INNER-PORT-IREF> refers to the port on the software component prototype within the composition. The <OUTER-PORT-REF> refers to the port configured on the composition itself.

### 13.3.3 Services and Service Connectors

Service components are added to an application by creating instances of the required service component types, in a similar manner to the way that atomic software components are created.

However, while application software component instances are based within software compositions, service components are allocated on an ECU-wide basis, outside the compositions.

Consequently, the set of services available on an ECU is defined as part of the ECU's configuration. The following figure shows two application software components connected to the same service component via different ports.

Figure 13.4 shows a service instance being used by two software component instances via different ports. The first component instance is connected to the service's requester port; the second to its provider port. In the resulting software composition there are two instances of the component SWC-Type, but only a single instance of the service Service-type.

The ECU's software composition also contains <SERVICE-CONNECTOR> elements, which connect a port on an application software component prototype to a port on a service

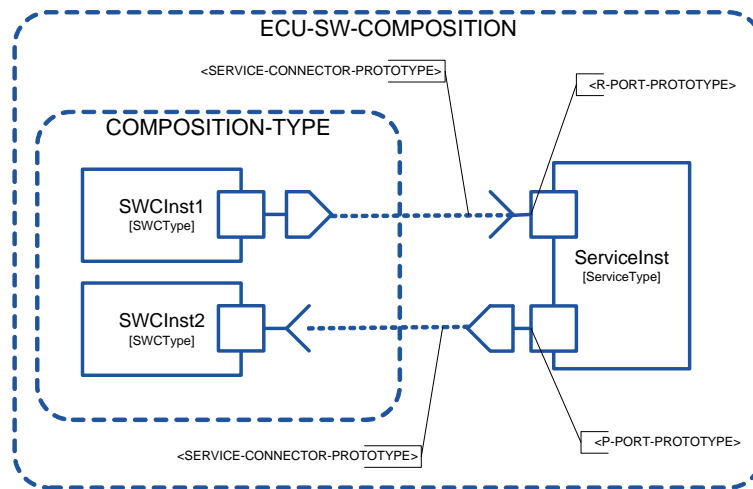


Figure 13.4: Service Connectors

component.

Service connectors are broadly similar to assembly connectors, but differ slightly in their format: rather than referring to the endpoints as a PROVIDER and REQUESTER, the endpoints are referred to as APPLICATION port and the SERVICE port. This helps to prevent a service component being connected to another service component, which is not permitted.



*As with assembly connectors, a service connector must connect exactly one provided port and one required port. However, either the application port or the service port may be the provided port, and the other must then be the required port.*

```
<SERVICE-CONNECTOR-PROTOTYPE>
  <SHORT-NAME>SvcConnector</SHORT-NAME>
  <APPLICATION-PORT-IREF>
    <COMPONENT-PROTOTYPE-REF ...>
    <PORT-PROTOTYPE-IREF ...>
  </APPLICATION-PORT-REF>
  <SERVICE-PORT-IREF>
    <SERVICE-COMPONENT-PROTOTYPE-REF ...>
    <PORT-PROTOTYPE-REF ...>
  </SERVICE-PORT-IREF>
</SERVICE-CONNECTOR-PROTOTYPE>
```

It is also possible to use a service connector between a port on a service component and a port on a software composition.

### 13.3.4 Port Interface Mapping

Prior to AUTOSAR R4.0 the compatibility of connected ports was verified by matching elements based on name, for example, a provided port with elements "A", "B" and "C" would be compatible with a required port with compatible elements "A" and "B" (since



the set of elements in the provider could be a superset of the requirer's elements) however it would not be compatible with a required port containing element "D". To resolve this problem R4.0 introduced the Port Interface Mapping element.

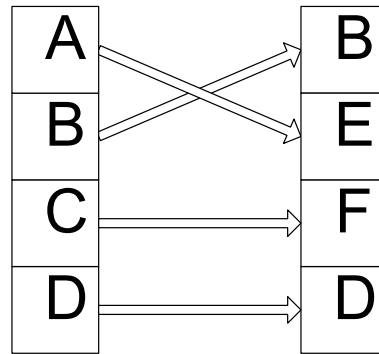


Figure 13.5: Port interface map for an interface containing A, B, C and D connected to an interface containing B, E, F and D.

Figure 13.5 illustrates a port interface map for a provider categorized by an interface containing elements "A", "B", "C", "D" mapped to an interface containing elements "B", "E", "F", "D". The mapping resolves the incompatibility by explicitly specifying the mapping, namely "A" to "E", "B" to "B", "C" to "F" and "D" to "D".



*A port interface mapping must explicitly associate all pairs of elements that are to be connected; when a port interface mapping is applied to a connector the automatic mapping by matching name no longer occurs.*



*This release of RTA-RTE supports port mapping for Client-server operations and arguments, Sender-Receiver data elements, Calibration parameters, Mode declaration groups and external Triggers.*

Once a port mapping between interfaces is defined then it can be applied to a particular assembly connector using a <MAPPING-REF>.

```
<ASSEMBLY-SW-CONNECTOR>
  <SHORT-NAME>Connector1</SHORT-NAME>
  <MAPPING-REF DEST='...'>/pkg/pif/map</MAPPING-REF>
  <PROVIDER-IREF>
    ...
  </PROVIDER-IREF>
  <REQUESTER-IREF>
    ...
  </REQUESTER-IREF>
</ASSEMBLY-SW-CONNECTOR>
```

If desired, a port interface mapping can be reused for multiple connectors, subject to the provider and requirer ports of the connector being categorized by the same interfaces as mapped in the port interface mapping.



*Port interface mappings merely express pairings of elements within a pair of interfaces; the “first” interface may be the interface of the provide port and the “second” the interface of the require port or vice versa. It is not therefore possible to use a port interface mapping on a connector between ports with the same interface.*

RTA-RTE supports nested compositions where possible and port interface mappings can be used on DelegationConnectors as well as AssemblyConnectors in most cases. At the time of writing this does not include client-server port interface mappings; these can only be used on AssemblyConnectors in this release of RTA-RTE.

### Mapping Data Elements and Parameters

A port mapping for Sender-Receiver interfaces or a Parameter interface is defined using a <VARIABLE-AND-PARAMETER-INTERFACE-MAPPING> element which maps elements from one **interface** to elements of another interface. The mapping is performed by reference to the data element (parameter) prototypes and therefore elements with different names can be mapped.

```
<PORT-INTERFACE-MAPPING-SET>
  <SHORT-NAME>pif</SHORT-NAME>
  <PORT-INTERFACE-MAPPINGS>
    <VARIABLE-AND-PARAMETER-INTERFACE-MAPPING>
      <SHORT-NAME>map</SHORT-NAME>
      <DATA-MAPPINGS>
        <DATA-PROTOTYPE-MAPPING>
          <FIRST-DATA-PROTOTYPE-REF DEST='...'>/pkg/ifa/a</FIRST-DATA-
            PROTOTYPE-REF>
          <SECOND-DATA-PROTOTYPE-REF DEST='...'>/pkg/ifb/b</SECOND-DATA-
            PROTOTYPE-REF>
        </DATA-PROTOTYPE-MAPPING>
      </DATA-MAPPINGS>
    </VARIABLE-AND-PARAMETER-INTERFACE-MAPPING>
  ...
```

If a pair of connected data (or parameter) prototypes have incompatible complex types a remapping of the sub-elements of those types can be defined by including <SUB-ELEMENT-MAPPING> elements in the <DATA-PROTOTYPE-MAPPING>.

```
<DATA-PROTOTYPE-MAPPING>
  <FIRST-DATA-PROTOTYPE-REF DEST='...'>/pkg/ifa/dia</FIRST-DATA-PROTOTYPE-
    REF>
  <SECOND-DATA-PROTOTYPE-REF DEST='...'>/pkg/ifb/dib</SECOND-DATA-PROTOTYPE-
    REF>
  <SUB-ELEMENT-MAPPINGS>
    <SUB-ELEMENT-MAPPING>
      <FIRST-ELEMENTS>
        <IMPLEMENTATION-DATA-TYPE-SUB-ELEMENT-REF>
        <IMPLEMENTATION-DATA-TYPE-ELEMENT>
        <CONTEXT-DATA-PROTOTYPE-REFS/>
        <TARGET-DATA-PROTOTYPE-REF DEST='IMPLEMENTATION-DATA-TYPE-
          ELEMENT'>/pkg/RecordTypeTx/a</TARGET-DATA-PROTOTYPE-REF>
      </FIRST-ELEMENTS>
    </SUB-ELEMENT-MAPPING>
  </SUB-ELEMENT-MAPPINGS>
</DATA-PROTOTYPE-MAPPING>
```

```

        </IMPLEMENTATION-DATA-TYPE-ELEMENT>
    </IMPLEMENTATION-DATA-TYPE-SUB-ELEMENT-REF>
</FIRST-ELEMENTS>
<SECOND-ELEMENTS>
    <IMPLEMENTATION-DATA-TYPE-SUB-ELEMENT-REF>
        <IMPLEMENTATION-DATA-TYPE-ELEMENT>
            <CONTEXT-DATA-PROTOTYPE-REFS/>
            <TARGET-DATA-PROTOTYPE-REF DEST='IMPLEMENTATION-DATA-TYPE-
                ELEMENT' INDEX='0'>/pkg/ArrayTypeRx/element</TARGET-DATA-
                PROTOTYPE-REF>
        </IMPLEMENTATION-DATA-TYPE-ELEMENT>
    </IMPLEMENTATION-DATA-TYPE-SUB-ELEMENT-REF>
</SECOND-ELEMENTS>
</SUB-ELEMENT-MAPPING>
...
</SUB-ELEMENT-MAPPINGS>
</DATA-PROTOTYPE-MAPPING>

```

The above example maps the record element RecordTypeTx.a to the array element ArrayTypeRx[0]. In nested complex types nested elements can be mapped by including references under <CONTEXT-DATA-PROTOTYPE-REFS> to give the path down the tree of elements. It is also possible to specify a mapping between elements that themselves have complex types, provided those types are compatible, without having to provide explicit mappings for all their leaves.

### Mapping Operations and Arguments

A port mapping for Client-Server interfaces is defined using a <CLIENT-SERVER-INTERFACE-MAPPING> element which maps operations from one **interface** to operations of another interface and, optionally, arguments within the operations. The mapping is performed by reference to operations prototypes and therefore operations with different names can be mapped.

```

<PORT-INTERFACE-MAPPING-SET>
    <SHORT-NAME>pif</SHORT-NAME>
    <PORT-INTERFACE-MAPPINGS>
        <CLIENT-SERVER-INTERFACE-MAPPING>
            <SHORT-NAME>map</SHORT-NAME>
            <OPERATION-MAPPINGS>
                <CLIENT-SERVER-OPERATION-MAPPING>
                    <FIRST-OPERATION-REF DEST='...'>/pkg/ifa/a</FIRST-OPERATION-REF>
                    <SECOND-OPERATION-REF DEST='...'>/pkg/ifb/b</SECOND-OPERATION-REF>
                >
            </CLIENT-SERVER-OPERATION-MAPPING>
        </OPERATION-MAPPINGS>
    </CLIENT-SERVER-INTERFACE-MAPPING>
    ...

```

Once a port mapping between interfaces is defined then it can be referenced from an assembly connector using an <MAPPING-REF>.

```

<ASSEMBLY-SW-CONNECTOR>

```

```

<SHORT-NAME>Connector1</SHORT-NAME>
<MAPPING-REF DEST='...'>/pkg/pif/map</MAPPING-REF>
<PROVIDER-IREF>
...
</PROVIDER-IREF>
<REQUESTER-IREF>
...
</REQUESTER-IREF>
</ASSEMBLY-SW-CONNECTOR>

```

If desired, a port interface mapping can be reused for multiple connectors (subject to the provider and requirer ports of the connector being categorized by the same interfaces as mapped in the port interface mapping).

In addition to mapping two operations a <CLIENT-SERVER-OPERATION-MAPPING> element can be used to map individual parameters within the two referenced operations. Unmapped arguments are mapped by position. All arguments, whether explicitly mapped by an operation mapping or implicitly mapped by position, must be compatible (e.g. have compatible data types).

```

<CLIENT-SERVER-OPERATION-MAPPING>
  <ARGUMENT-MAPPINGS>
    <DATA-PROTOTYPE-MAPPING>
      <FIRST-DATA-PROTOTYPE-REF DEST='...'>/pkg/ifa/a/a</FIRST-DATA-
        PROTOTYPE-REF>
      <SECOND-DATA-PROTOTYPE-REF DEST='...'>/pkg/ifb/b/b</SECOND-DATA-
        PROTOTYPE-REF>
    </DATA-PROTOTYPE-MAPPING>
    <DATA-PROTOTYPE-MAPPING>
      ...
    </DATA-PROTOTYPE-MAPPING>
  </ARGUMENT-MAPPINGS>
  <FIRST-OPERATION-REF DEST='...'>/pkg/ifa/a</FIRST-OPERATION-REF>
  <SECOND-OPERATION-REF DEST='...'>/pkg/ifb/b</SECOND-OPERATION-REF>
</CLIENT-SERVER-OPERATION-MAPPING>

```



*Within a <DATA-PROTOTYPE-MAPPING> the first referenced prototype must be contained within the first referenced operation and the second referenced prototype must be contained within the second referenced operation.*



*At the time of writing, RTA-RTE does not permit <SUB-ELEMENT-MAPPING> elements to be used to map the elements of incompatible complex types for a <DATA-PROTOTYPE-MAPPING> element within a <CLIENT-SERVER-OPERATION-MAPPING>.*

## Mapping Triggers

A port mapping for Trigger interfaces is defined using a <TRIGGER-INTERFACE-MAPPING> element which maps elements from one **interface** to elements of another interface. The mappings is performed by reference to the

triggers themselves and therefore elements with different names can be mapped.

```
<PORT-INTERFACE-MAPPING-SET>
  <SHORT-NAME>pif</SHORT-NAME>
  <PORT-INTERFACE-MAPPINGS>
    <TRIGGER-INTERFACE-MAPPING>
      <SHORT-NAME>map</SHORT-NAME>
      <TRIGGER-MAPPINGS>
        <TRIGGER-MAPPING>
          <FIRST-TRIGGER-REF DEST='...'>/pkg/ifa/a</FIRST-TRIGGER-REF>
          <SECOND-TRIGGER-REF DEST='...'>/pkg/ifb/b</SECOND-TRIGGER-REF>
        </TRIGGER-MAPPING>
      </TRIGGER-MAPPINGS>
    </TRIGGER-INTERFACE-MAPPING>
  ...
```

Once a port mapping between trigger interfaces is defined then it can be referenced from an assembly connector using an <MAPPING-REF>. This is the same reference tag as the reference for mapping data elements but since an port cannot be categorized by multiple interfaces no ambiguity results.

```
<ASSEMBLY-SW-CONNECTOR>
  <SHORT-NAME>Connector1</SHORT-NAME>
  <MAPPING-REF DEST='...'>/pkg/pif/map</MAPPING-REF>
  <PROVIDER-IREF>
    ...
  </PROVIDER-IREF>
  <REQUESTER-IREF>
    ...
  </REQUESTER-IREF>
</ASSEMBLY-SW-CONNECTOR>
```

If desired, a port interface mapping can be reused for multiple connectors (subject to the provider and requirer ports of the connector being categorized by the same interfaces as mapped in the port interface mapping).

### Mapping Mode Declaration Groups

A port mapping for Mode interfaces is defined using a <MODE-INTERFACE-MAPPING> element which maps elements from one **interface** to elements of another interface. The mappings is performed by reference to the mode declaration group prototypes and therefore elements with different names can be mapped.

```
<PORT-INTERFACE-MAPPING-SET>
  <SHORT-NAME>pif</SHORT-NAME>
  <PORT-INTERFACE-MAPPINGS>
    <MODE-INTERFACE-MAPPING>
      <SHORT-NAME>map</SHORT-NAME>
      <MODE-MAPPINGS>
        <MODE-MAPPING>
          <FIRST-MODE-GROUP-REF DEST='...'>/pkg/ifa/a</FIRST-MODE-GROUP-REF>
        >
```

```

    <SECOND-MODE-GROUP-REF DEST='...'>/pkg/ifb/b</SECOND-MODE-GROUP-
      REF>
  </MODE-MAPPING>
</MODE-MAPPINGS>
</MODE-INTERFACE-MAPPING>
...

```

Once a port mapping between mode interfaces is defined then it can be referenced from an assembly connector using an <MAPPING-REF>. This is the same reference tag as the reference for mapping data elements and triggers but since an port cannot be categorized by multiple interfaces no ambiguity results.

```

<ASSEMBLY-SW-CONNECTOR>
  <SHORT-NAME>Connector1</SHORT-NAME>
  <MAPPING-REF DEST='...'>/pkg/pif/map</MAPPING-REF>
  <PROVIDER-IREF>
    ...
  </PROVIDER-IREF>
  <REQUESTER-IREF>
    ...
  </REQUESTER-IREF>
</ASSEMBLY-SW-CONNECTOR>

```

If desired, a port interface mapping can be reused for multiple connectors (subject to the provider and requirer ports of the connector being categorized by the same interfaces as mapped in the port interface mapping).

### 13.3.5 Partial Record Mappings

AUTOSAR has the concept of *partial record mappings*. These mappings extend the normal type compatibility rules such that a providing Sender-receiver data element that is typed by a record type can contain a **super-set** of the elements in the receiver's record data type.

Elements in the providing and requiring type are matched by name; thus the two types do not need to have the same elements in the same order.

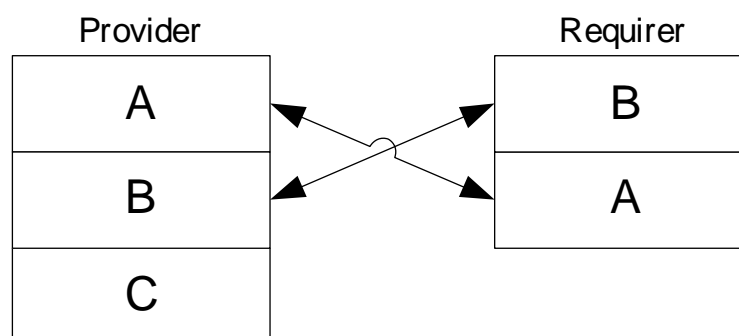


Figure 13.6: Valid Partial Record Mapping

Figure 13.6 illustrates a valid mapping between two record types where the providing

type contains the three data elements “A”, “B” and “C” whereas the requiring type contains data elements “A” and “B” in a different order.

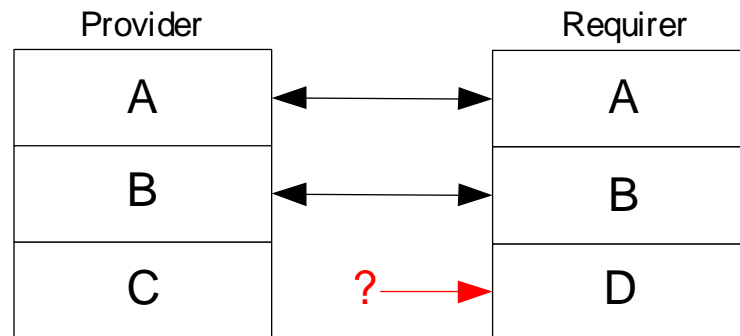


Figure 13.7: Invalid Partial Record Mapping

However Figure 13.7 illustrates an invalid mapping between two record types where the requiring type contains a record element, “D” not included in the providing record type. The providing type must be a super-set of the requiring type.



*Partial record mappings are not supported in R4.0 but the same effect can be achieved by using SubElementMappings. See section 13.3.4.*

## 13.4 FlatMaps

FlatInstanceDescriptors are used in AUTOSAR 4.0 to create readable names for instances in the input model.

FlatInstanceDescriptors are aggregated into a FlatMap which is referenced by the RootSwCompositionPrototype. Thus, for an ECU\_EXTRACT the FlatInstanceDescriptors provide unique names for the instances of VariableDataPrototypes and SwcPrototypes.



*At the time of writing, for historical reasons RTA-RTE also takes the shortName of the FlatInstanceDescriptor as the C-symbol of the measurement buffer in generation phase. This should not be relied upon and may be removed in a future release.*



*At the time of writing, FlatInstanceDescriptors cannot reference ConstantMemorys or StaticMemorys. This is because there is no generated API for accessing ConstantMemorys or StaticMemorys, so the renaming of the buffer cannot be supported.*

### 13.4.1 Configuration

A flatmap contains one or more *flatmap instance* elements that are used to contain the information for the ECU being configured.

```
<FLAT-MAP>
  <SHORT-NAME>Flatmap</SHORT-NAME>
```

```
<INSTANCES>
  <FLAT-INSTANCE-DESCRIPTOR>
    ...
  </FLAT-INSTANCE-DESCRIPTOR>
</INSTANCES>
</FLAT-MAP>
```

A flatmap instance contain an instance references, the “ecu-extract” reference to the associated element (data element, inter-runnable variable, etc.) instance.

```
<FLAT-INSTANCE-DESCRIPTOR>
  <SHORT-NAME>my_argument_buffer</SHORT-NAME>
  <ECU-EXTRACT-REFERENCE-IREF>
    ...
  </ECU-EXTRACT-REFERENCE-IREF>
</FLAT-INSTANCE-DESCRIPTOR>
```

The ECU extract instance reference within a flatmap instance may require multiple CONTEXT nodes to identify the correct element. For example, when referencing a data element as well as the component prototype reference an additional two references are required to specify the port prototype and the data element.

A flatmap is associated with the ECU extract by a reference defined within the <ROOT-SW-COMPOSITION-PROTOTYPE> element:

```
<ROOT-SW-COMPOSITION-PROTOTYPE>
  <SHORT-NAME>mySWComp</SHORT-NAME>
  ...
  <FLAT-MAP-REF>/pkg/flatmap</FLAT-MAP-REF>
  ...
</ROOT-SW-COMPOSITION-PROTOTYPE>
```

All flatmap instances must exist within the same flatmap since the root composition element only contains a single element. However the flatmap element is splittable and therefore RTA-RTE allows it to be distributed across multiple files.

#### 13.4.2 Renaming a SW-C instance

Within RTA-RTE the second reference, to the associated system element, is optional. When omitted RTA-RTE uses the name of the flatmap instance as the **internal** name of the SW-C instance.



*All internal names must be unique. RTA-RTE will raise an error if this constraint is violated by, for example, using a flatmap to assign a duplicate name to a SW-C instance.*

The following flatmap instance renames the internal name of SW-C component prototype /pkg/Compo/swcA1 to “swcAInst”.

```
<FLAT-INSTANCE-DESCRIPTOR>
  <SHORT-NAME>swcAInst</SHORT-NAME>
  <ECU-EXTRACT-REFERENCE-IREF>
```



```
<TARGET-REF DEST='SW-COMPONENT-PROTOTYPE'>/pkg/Compo/swcA1</TARGET-REF>  
</ECU-EXTRACT-REFERENCE-IREF>  
</FLAT-INSTANCE-DESCRIPTOR>
```

When referencing a SW-C prototype the ECU extract reference <TARGET-REF> must use a DEST attribute of SW-COMPONENT-PROTOTYPE.

### 13.4.3 Measurement Buffers

As well as renaming a SW-C instance a flatmap instance can also be used to set the name used for a measurement buffer. When specified the short name of the flatmap instance is used instead of the RTE generated name.

The following flatmap instance renames the measurement buffer of data element “di1” in port prototype “swcA/ra” for SW-C instance “swcA1” to “my\_measurement\_buffer”.

```
<FLAT-INSTANCE-DESCRIPTOR>  
<SHORT-NAME>my_measurement_buffer</SHORT-NAME>  
<ECU-EXTRACT-REFERENCE-IREF>  
<CONTEXT-ELEMENT-REF DEST='SW-COMPONENT-PROTOTYPE'>/pkg/Compo/swcA1</  
CONTEXT-ELEMENT-REF>  
<CONTEXT-ELEMENT-REF DEST='R-PORT-PROTOTYPE'>/pkg/swcA/ra</CONTEXT-  
ELEMENT-REF>  
<TARGET-REF DEST='VARIABLE-DATA-PROTOTYPE'>/pkg/if1/di1</TARGET-REF>  
</ECU-EXTRACT-REFERENCE-IREF>  
</FLAT-INSTANCE-DESCRIPTOR>
```

In the example the ECU extract reference contains three elements:

1. The first <CONTEXT-ELEMENT-REF> element uses a DEST attribute of SW-COMPONENT-PROTOTYPE since it references the component prototype. This attribute is required by RTA-RTE.
2. The second <CONTEXT-ELEMENT-REF> element uses a DEST attribute of R-PORT-PROTOTYPE since it references the port prototype.
3. The final <TARGET-REF> uses a DEST attribute of VARIABLE-DATA-PROTOTYPE since it references the data element.

### 13.4.4 Receive Buffers

To minimize RAM usage RTA-RTE will use an assigned measurement buffer name for the receive buffer where possible. This renaming occurs even if measurement is not enabled for the data element and thus a flatmap instance can be used to give the RTE’s receive buffer a meaningful name.



*Care must be taken to ensure that names assigned using flatmap instances are unique and do not conflict with names used for parameters within RTE API calls (“data”) or return values (“rtn”).*

RTA-RTE will use an name supplied via a flatmap instance for either the provide or require port. However to avoid potential double writes (one to the measurement buffer and one to the receiver) it is recommended to apply the flatmap instance to provider's port. Applying the flatmap to the provider also means that the same name is used for the buffer for all non-filtered receivers on the same ECU.

#### 13.4.5 Inter-Runnable Variables

A flatmap instance can also be used to set the name used for the global buffer used for an inter-runnable variable. When specified the shortName of the flatmap instance is used instead of the RTE generated name.

The following flatmap instance renames the buffer of IRV "irv1" for SW-C instance "swcA1" to "my\_irv\_buffer".

```
<FLAT-INSTANCE-DESCRIPTOR>
  <SHORT-NAME>my_irv_buffer</SHORT-NAME>
  <ECU-EXTRACT-REFERENCE-IREF>
    <CONTEXT-ELEMENT-REF DEST='SW-COMPONENT-PROTOTYPE'>/pkg/Compo/swcA1</
      CONTEXT-ELEMENT-REF>
    <TARGET-REF DEST='VARIABLE-DATA-PROTOTYPE'>/pkg/IBswcA/irv1</TARGET-REF
      >
  </ECU-EXTRACT-REFERENCE-IREF>
</FLAT-INSTANCE-DESCRIPTOR>
```

In the example the ECU extract reference contains two elements:

1. The <CONTEXT-ELEMENT-REF> element uses a DEST attribute of SW-COMPONENT-PROTOTYPE since it references the component prototype. This attribute is required by RTA-RTE.
2. The <TARGET-REF> element uses a DEST attribute of VARIABLE-DATA-PROTOTYPE since it references the inter-runabble variable.

A flatmap instance must also use a DEST attribute of VARIABLE-DATA-PROTOTYPE rather than IRV-HANDLE. This is dictated by the AUTOSAR R4.0 schema that also defines the format of the flatmap instances used by RTA-RTE when processing flatmaps.

#### 13.4.6 Calibration Parameters

The calibration method "None" uses the flatmap name as the name of the instantiated calibration parameter (if no flatmap is available the name of the calibration parameter is used instead).

The following flatmap instance names the instantiated calibration parameter "cp\_value1" for calibration component instance "cpcData1" in provided port "pSys1" to "cv1".

```
<FLAT-INSTANCE-DESCRIPTOR>
  <SHORT-NAME>cv1</SHORT-NAME>
```

```
<ECU-EXTRACT-REFERENCE-IREF>  
  <CONTEXT-ELEMENT-REF DEST='SW-COMPONENT-PROTOTYPE'>/pkg/Compo/cpcData1<  
    /CONTEXT-ELEMENT-REF>  
  <CONTEXT-ELEMENT-REF DEST='P-PORT-PROTOTYPE'>/pkg/cpcData/pSys1</  
    CONTEXT-ELEMENT-REF>  
  <TARGET-REF DEST='PARAMETER-DATA-PROTOTYPE'>/pkg/if_cp1/cp_value1</  
    TARGET-REF>  
</ECU-EXTRACT-REFERENCE-IREF>  
</FLAT-INSTANCE-DESCRIPTOR>
```

The instance reference contains three elements:

1. The first <CONTEXT-ELEMENT-REF> element uses a DEST attribute of SW-COMPONENT-PROTOTYPE since it references the component prototype. This attribute is required by RTA-RTE.
2. The second <CONTEXT-ELEMENT-REF> element uses a DEST attribute of R-PORT-PROTOTYPE since it references the port prototype.  
The ECU extract reference includes a contextualizing port prototype for parameters in calibration SW-Cs only; for shared or per-instance parameter prototypes no port reference is required.
3. The final <TARGET-REF> uses a DEST attribute of PARAMETER-DATA-PROTOTYPE since it references the data element.

#### 13.4.7 McSupportData

McSupportData is the AUTOSAR 4.0 format for exporting information for use in Measurement and Calibration tools. In practice, this is usually converted to A2L by a downstream tool that merges information from the linker map file to calculate the actual memory addresses of the various measurable data items.

The McSupportData is generated every time an RTE is generated. It is written to the file Rte\_McSupportData.arxml and placed in a folder according to the --output option.

For a variable data instance to measurable, and hence to appear in the McSupportData, three conditions must be met:

- RteMeasurementSupport must be true in the ECU Value File.
- The effective SwCalibrationAccess for the VariableDataPrototype must be READ-ONLY
- There must be a FlatInstanceDescriptor referencing the VariableDataPrototype in the context of the desired SwComponentInstance



*If a VariableDataPrototype for a data instance is marked as READ-ONLY but no FlatInstanceDescriptor references it, then a comment is written to the McSupportFile explaining why the McDataInstance is missing and a warning is written to the error stream.*

## McSupportData Structure

---

The generated McSupportData file is structured according to the AUTOSAR Generic Structure Template, that is, elements are placed in subpackages according to their tag. The subpackages are placed in top-level package AUTOSAR\_Rte. For example, the BswImplementation element is placed in subpackage /AUTOSAR\_Rte/BswImplementations, and SwBaseType elements are placed in subpackage /AUTOSAR\_Rte/SwBaseTypes).

Measurables appear in the McSupportData in the form of a <MC-DATA- INSTANCE> element having at least the following children:

- <SHORT-NAME>
- <CATEGORY>
- <FLAT-MAP-ENTRY-REF>
- <SYMBOL>

<SYMBOL> matches the C identifier used in the generated RTE, which may be used to query the compiler's map file to find the memory location of the measurable or characteristic.

In the case of a simple measurable value (<CATEGORY> is VALUE), then there will be a ResultingProperties container containing references to the SwBaseType, DataConstr, CompuMethod, Unit etc. giving the size, encoding, and relation to the physical world required for extraction and display of the data instance.

In the case of a complex measurable value of Category STRUCT, the McDataInstance contains a <SUB-ELEMENTS> container containing a further McDataInstance for each structure member. If the member is itself a STRUCT, then this breakdown continues recursively until the leaf items of Category VALUE are reached. The <FLAT-MAP-ENTRY-REF> is at the top level, since the structure itself is the actual measurable / characteristic.

In the case of Calibration Parameters (characteristics in A2L), these are placed into groups in AUTOSAR, according to their addressing method. RTA-RTE achieves this grouping by creating the individual characteristics as members of a C struct representing the group. In the McSupportData, this is represented by a McDataInstance representing the group, having a SubElements container, which contains the individual characteristics. Note that the <FLAT-MAP-ENTRY-REF> in this case does not appear at the top level, since the actual characteristics are one level deeper. The <SYMBOL> however DOES appear at the top level, as this is the item that will appear in the mapfile. Calculating the exact offset of each characteristic is outside the scope of AUTOSAR and RTA-RTE. The downstream custom tooling must use knowledge of the particular compiler along with the information in the SwBaseTypes of the McSupportData to calculate the offsets.

## Limits of Implementation

---

AUTOSAR specifies how and where type-information (*'Effective DataDefProps'*) should be gathered. RTA-RTE currently implements the following (the list describes the order in which containers are interrogated for the given information - earlier containers take precedence over later containers):

**BaseTypeRef** – ImplementationDataType.

**DataConstr** – ImplementationDataType; ApplicationDataType.

**CompuMethod** – ImplementationDataType; ApplicationDataType.

**Unit** – ApplicationDataType.

**SwCalibrationAccess** – DataPrototype.

SwDataDefProps not mentioned above are not implemented.

## 13.5 Data Conversion

---

### 13.5.1 Purpose

---

Data Conversion allows the system integrator to connect Ports of Software Components with different but “convertible” data types by means of a PortInterfaceMapping (see section 13.3.4). RTA-RTE generates the C code to convert values from one numerical space to the other.

PortInterfaceMappings allow the connection of Ports typed by Interfaces that are not “AUTOSAR Compatible” by specifying how the incompatibility is to be resolved. The explicit connection of a pair of DataPrototypes in the Interfaces by a DataPrototypeMapping within a PortInterfaceMapping causes RTA-RTE to seek to generate a runtime conversion if their types are not “AUTOSAR Compatible”. PortInterfaceMappings must therefore be used when data conversion is necessary.

Data Conversion is attempted for ApplicationPrimitiveDataTypes and for the primitive sub-elements of complex ApplicationDataTypes. The necessary conversion code is automatically generated from the configured relationships between the implementation representation and the physical quantity or semantic meaning for the ApplicationPrimitiveDataTypes involved; see section 5.5.

AUTOSAR R4.0 and RTA-RTE support two types of Data Conversion: TextTable and Linear.

### 13.5.2 TextTable

---

#### Purpose

---

TextTable Data Conversion allows DataPrototypes of enumerated type (or of complex type with primitive elements of enumerated type) to be connected when they have the same or similar semantics but different numerical and/or symbolic representations of those semantics.

## Enabling

RTA-RTE attempts Text Table Data Conversion whenever a TextTableMapping is present in a VariableAndParameterInterfaceMapping referenced by the MappingRef of a Connector.

## Configuration

For illustration we will consider two Enumerated types Enum1 and Enum2 which have the same symbolic point ranges but with different numerical values:

Symbol	Enum1	Enum2
MY_SUCCESS	0	0
MY_WARNING	1	-1
MY_ERROR	2	1

We have a SenderReceiverInterface sr1 containing a single VariableDataPrototype a1, typed by Enum1; and SenderReceiverInterface sr2 contains a single VariableDataPrototype a2 typed by Enum2.

SoftwareComponentType swcTx aggregates provide port p1 characterized by sr1 and SoftwareComponentType swcRx aggregates require port r1 characterized by sr2.

These components are instantiated as swcTx1 and swcRx1 and an assembly connector connects the ports.

So now we have a data flow from an type of Enum1 to a type of Enum2.

The example below shows a suitable TextTableMapping for use in this scenario

```
<VARIABLE-AND-PARAMETER-INTERFACE-MAPPING>
  short_name
  <DATA-MAPPINGS>
    <DATA-PROTOTYPE-MAPPING>
      <FIRST-DATA-PROTOTYPE-REF DEST='VARIABLE-DATA-PROTOTYPE'>myPackage/
        sr1/a1</FIRST-DATA-PROTOTYPE-REF>
      <SECOND-DATA-PROTOTYPE-REF DEST='VARIABLE-DATA-PROTOTYPE'>myPackage/
        sr2/a2</SECOND-DATA-PROTOTYPE-REF>
    <TEXT-TABLE-MAPPINGS>
      <TEXT-TABLE-MAPPING>
        <MAPPING-DIRECTION>BIDIRECTIONAL</MAPPING-DIRECTION>
        <VALUE-PAIRS>
          <TEXT-TABLE-VALUE-PAIR>
            <FIRST-VALUE>0</FIRST-VALUE>
            <SECOND-VALUE>0</SECOND-VALUE>
          </TEXT-TABLE-VALUE-PAIR>
          <TEXT-TABLE-VALUE-PAIR>
            <FIRST-VALUE>1</FIRST-VALUE>
            <SECOND-VALUE>-1</SECOND-VALUE>
          </TEXT-TABLE-VALUE-PAIR>
        </VALUE-PAIRS>
      </TEXT-TABLE-MAPPING>
    </TEXT-TABLE-MAPPINGS>
  </DATA-MAPPINGS>
</VARIABLE-AND-PARAMETER-INTERFACE-MAPPING>
```

```

    <TEXT-TABLE-VALUE-PAIR>
      <FIRST-VALUE>2</FIRST-VALUE>
      <SECOND-VALUE>1</SECOND-VALUE>
    </TEXT-TABLE-VALUE-PAIR>
  </VALUE-PAIRS>
</TEXT-TABLE-MAPPING>
</TEXT-TABLE-MAPPINGS>
</DATA-PROTOTYPE-MAPPING>
</DATA-MAPPINGS>
</VARIABLE-AND-PARAMETER-INTERFACE-MAPPING>

```



Because this mapping is 1:1, i.e. all FirstValues are unique and all SecondValues are unique, it is declared with a MappingDirection of BIDIRECTIONAL. This means that the same VariableAndParameterInterfaceMapping might be used where Enum2 is the transmitted type and Enum1 is the received type. In that case, we say that the communication flow is going “second-to-first”.

#### Generated Code

When Text Table Data Conversion is configured, RTA-RTE alters its writes to receive buffers to include a call to conversion code that translates values in the direction of data flow (in our case, left-to-right from Enum1 to Enum2).

### 13.5.3 Linear

#### Purpose

Linear Data Conversion allows the system integrator to connect data items which represent the same physical quantity even though they may be expressed in different numerical terms. This is expressed by means of Units, in FactorSIToUnit and OffsetSIToUnit; and by CompuMethods, in CompuRationalCoefficients. Linear data conversion encompasses CompuMethods of Category LINEAR and also CompuMethods of Category IDENTICAL, which is just treated as a special case of Linear.

#### Functional Composition

Conceptually, linear data conversion consists of the *functional composition* of the four linear functions to move from the transmitted value  $x$  through the physical representation in the sender’s unit, to the SI unit, to the physical representation in the receiver’s unit, to the received value  $y$ :

$$f = cm_{rx} \circ unit_{rx} \circ unit_{tx} \circ cm_{tx}$$

or

$$y = cm_{rx}(unit_{rx}(unit_{tx}(cm_{tx}(x))))$$

Where  $cm_{tx}$ ,  $unit_{tx}$ ,  $unit_{rx}$  and  $cm_{rx}$  represent each step of the conversion as a linear function that maps the input value to an intermediate output value. See section 5.5 for more details about configuring the numerical and physical representations of `ApplicationPrimitiveDataTypes`.

Functional composition means that RTA-RTE only writes one expression in the generated code rather than nesting function calls corresponding to each of the four potential transformations. Even in the worst case, an entire data transformation of four applied transformations results in a single expression that performs one multiplication, one addition, and one division. In the best case, data conversion is optimized away to a simple assignment.

In addition, RTA-RTE simplifies the terms in the data conversion to remove highest common factors. This step simplifies the resultant expression and can mean that terms are eliminated entirely and avoid needlessly large intermediate values.

### Enabling

RTA-RTE attempts to configure Linear Data Conversion whenever a `PortInterfaceMapping` connects `DataPrototypes` that are typed by `ApplicationPrimitiveDataTypes` which are not “AUTOSAR Compatible” or complex `ApplicationDataTypes` that are shape compatible but whose primitive elements are not “AUTOSAR Compatible”.

For each primitive value subject to Linear Data Conversion, RTA-RTE extracts from the configuration the data conversions expressed by the transmitter’s `CompuMethod`, the transmitter’s `Unit`, the receiver’s `Unit` and the Receiver’s `CompuMethod`. It then combines the four conversion functions using functional composition to arrive at one conversion to be expressed in C.

### Configuration

Configuration of Linear Data Conversion is achieved by expressing each `ApplicationPrimitiveDataType`’s relationship to some `PhysicalDimension`. Then, when data flows from one `DataPrototype` to another, RTA-RTE automatically inserts the necessary code to convert one numerical representation of a physical quantity to another. See section 5.5 to read about the correct declaration of `ApplicationPrimitiveDataTypes`.

The decision of whether an application data type is *convertible* to another application data type is based on their `CompuMethods`, `Units`, and `PhysicalDimensions`:

- Every data type, `CompuMethod`, `Unit`, or `PhysicalDimension` is compatible with itself.
- A `PhysicalDimension` is compatible with another `PhysicalDimension` if the Short-Name and all the exponents match.
- A `Unit` is compatible with another `Unit` if both units reference a compatible `PhysicalDimension`.



If the physical dimension referenced by two units is compatible, then a conversion between them is possible.

- A CompuMethod is “Linear” if its Category is LINEAR. A CompuMethod is also “Linear” if its Category is IDENTICAL.

A CompuMethod is compatible with another CompuMethod if, either:

- Both CompuMethods are Linear and they reference compatible Units, or,
- Both CompuMethods are Linear and neither of them references a Unit.
- A CompuMethod of Category TEXTTABLE is also convertible with the input configuration supplying a mapping between numeric values.
- An ApplicationDataType **is convertible to** another ApplicationDataType if the types have compatible CompuMethods.

#### Generated Code

Linear Data Conversion code is inlined near to the receiver-buffer write in generated code. Zero-effect operations are elided, e.g. division-by-one and add-zero result in no code being generated. So in the extreme case, if the transmitter and receiver’s conversion functions match then no conversion takes place and no conversion code is visible.

RTA-RTE also performs worst-case calculations to determine the potential size of the intermediate values in the generated C expression. If no Platform Type is large enough to hold some intermediate value then an error is raised (but see `--have-64bit-int-types`).

#### 13.5.4 Rational Functions

The ASAM documents, which informed a portion of the AUTOSAR design, include many other categories of CompuMethod besides IDENTICAL, LINEAR and TEXTTABLE, for example RATFUNC.

AUTOSAR does not support data conversion via these categories of CompuMethod and neither does RTA-RTE. However RTA-RTE does accept models that may contain other categories of CompuMethod provided they conform with “AUTOSAR Compatibility” rules. That is, before attempting any Data Conversion, RTA-RTE performs a comparison of the Units and CompuMethods referenced by the sender and receiver, and if they match according to the AUTOSAR 4.0.3 rules, then the IDENTITY transformation is selected, i.e. no conversion code is written.



*RTA-RTE cannot ignore “AUTOSAR Compatible” CompuMethods in a data path where the references Units are not “AUTOSAR Compatible”. An unsupported Category of CompuMethod in that case would cause the input model to be rejected.*

### 13.5.5 Implementation Details

---

### 13.5.6 Internal Precision

---

Any data conversion represented by a `CompuPhysToInternal`, `CompuInternalToPhys`, or `Unit` (`FactorSiToUnit` and `OffsetSiToUnit`) is stored internally as three 64-bit signed integers.

When the data conversions are combined due to data flow through more than one `Unit` or `CompuMethod`, a functional composition takes place resulting in a composite data conversion, which is itself linear. This is also stored as three 64-bit signed integers. This is simplified using Highest Common Factor analysis to avoid unnecessarily large numbers being handled internally or appearing on the target hardware.

In the case of a `Unit`, the configuration input is expressed in two decimal numbers rather than three integers. RTA-RTE represents the two numbers internally as rationals, expressed in the same manner as for `CompuMethods`. This avoids the need for floating-point arithmetic internally or on the target. For example, a `FactorSiToUnit` of 0.01 results in an internal `DataConversion` object with  $num_0 = 0$ ,  $num_1 = 1$  and  $denom = 100$ , giving  $\frac{0+(1*x)}{100}$ .

The conversion of decimals to 64-bit signed integers places a limit of 18 decimal places on the `FactorSiToUnit` and `OffsetSiToUnit`, as well as a limit of 18 significant figures.

### 13.5.7 In Generated Code

---

When generating the C functions to perform Data Conversion, RTA-RTE tracks the minimum (closest to negative infinity) and maximum (closest to infinity) values that can be represented by the incoming type, by each constant, and by the result of each operation. For each operation, RTA-RTE chooses a suitable Platform Type and casts the arguments to each operation to that type. This ensures that the C compiler allows a sufficiently large working location for each operation and truncation will not happen on the target.

For example, consider the following C extract:

```
uint16 argument;  
...  
uint16 result = ( ( 65534 * argument ) / 65532 );
```

The result of this will rarely be correct, since C only allows a `uint16` for the intermediate value. To overcome this, RTA-RTE uses casts like in the following example:

```
uint16 argument;  
...  
uint16 result = ( ( (uint32)65534 * argument ) / 65532 );
```

In fact, RTA-RTE generates many more casts than this, as a way of ensuring that C's promotion rules do not introduce any counterintuitive effects.

If the intermediate value requires 64 bits, then the configuration is rejected, since AUTOSAR does not provide a 64-bit type. If you wish to use 64 bit integers, then specify `--have-64bit-int-types` on the command line and provide definitions in `Platform_Types.h` for `uint64` and `sint64`.

### 13.5.8 Composition and Delegated Ports

---

In a configuration containing a nested hierarchy of composition software components the connection between the provide port of one atomic software component instance and the require port of another may consist of a number of `DelegationConnectors` from the inner provide port to delegated ports on the enclosing compositions, an `AssemblyConnector` and a number of `DelegationConnectors` from delegated require ports on compositions down to the inner require port. In this case a functional composition of the notional data conversion steps along each Connector between the inner provide port and the inner require port occurs; the intermediate representations in the types for the intermediate delegated ports are not explicitly calculated at runtime but rather the overall conversion is performed in one step.

## 14 Composing Basic Software

In Chapter 13 we discussed how application software is partitioned using AUTOSAR software components and how these components are composed into a system. However, unlike a software component, BSW modules are not prototyped within a composition but are instead instantiated within the ECUC file.

### 14.1 Instantiation

The creation of an **RteBswModuleInstance** ECUC container within the RTE's ECUC module configuration defines an instance of the BSW module that can then contain the mapping from schedulable entities to tasks.

In the same way that a software component type can have multiple prototypes a BSW module description can have multiple instances. For example a CAN driver module may have an instance for each CAN controller. The BSW module instance within the ECU configuration therefore references a BSW implementation which in turn references the BSW module description rather than directly referencing the BSW module description.

**ETAS** *BSW modules do not support a "supports multiple instantiation" flag and therefore RTA-RTE assumes this attribute is true for all BSW modules.*

The instantiation of a BSW module instance is illustrated in Figure 14.1.

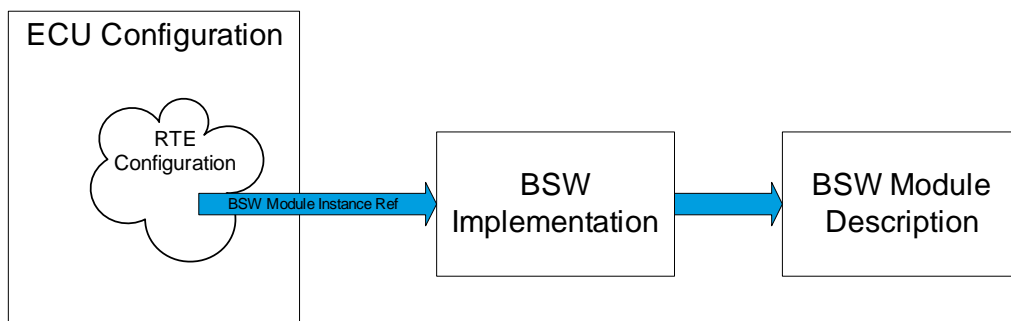


Figure 14.1: BSW Module Instantiation

In addition to referencing the relevant BSW module description the BSW implementation also optionally defines two items that are of interest to RTA-RTE. When defined the `<VENDOR-ID>` and `<VENDOR-API-INFIX>` are used when generating BSW APIs to ensure that each BSW instance has unique names for the generated APIs.

As an example, consider a BSW implementation for a CAN driver with vendor ID of "25" and API infix of "Dev0815". RTA-RTE would then generate API names of the form `SchM_Can_Dev0815_25_<XXX>` and thus ensure that the API was uniquely generated for the specific BSW instance.




*If the `<VENDOR-ID>` and `<VENDOR-API-INFIX>` are not defined within the BSW module's implementation then the APIs of multiple instances of the same BSW module will conflict.*

## 14.2 BSW Scheduler

---

Starting with AUTOSAR R4.0, the generation of a scheduler for basic software has been integrated into RTE generation rather than being available as a separate module as in previous releases. The basic software scheduler is responsible for ensuring that code within BSW modules runs at the correct times and that APIs for ensuring data consistency across multiple BSW modules are available.

RTA-RTE supports generation of a common task framework; BSW *schedulable entities* can be freely mixed with runnable entities from software components and the RTE generator will create the appropriate glue code to ensure that all are executed at the right time.

 *This release of RTA-RTE does not support the generation of shared exclusive areas and thus the `SchM_Enter` and `SchM_Exit` APIs are not supported.*

### 14.2.1 Task Mapping

---

A BSW module instance within the ECU configuration can contain zero or more **RteBswEventToTaskMapping** sub-containers that define the task and position of the schedulable entity associated with a BSW event. As with SW-C runnable entities all BSW events that can start a schedulable entity must have a defined task mapping.

RTA-RTE supports the mixing of BSW schedulable entities and SW-C runnable entities within the same task. This support includes the creation of basic or extended tasks as necessary; if extended tasks are to be avoided then the same precautions described in Section 24.4 apply.

## 14.3 Connections

---

BSW triggers and modes are not related to ports and therefore cannot be connected in the same way as the analogous entities within SWCs. Instead the connections are made within the ECUC using specific containers.

### 14.3.1 Triggers

---

An external trigger used by a BSW module must have the “release” and “required” triggers connected using a **RteBswRequiredTriggerConnection** element in the ECUC file.

Each `RteBswRequiredTriggerConnection` element includes three references:

1. A **RteBswReleasedTriggerModInstRef** that references the BSW module instance.
2. A **RteBswReleasedTriggerRef** that references the released trigger. When combined with the **RteBswReleasedTriggerModInstRef** reference this uniquely identifies the released trigger.
3. A **RteBswRequiredTriggerRef** that references the required trigger.

### 14.3.2 Modes

---

An mode used by a BSW module must have the “provided” and “required” mode declaration group prototypes connected using a **RteBswRequiredModeGroupConnection** element in the ECUC file.

Each **RteBswRequiredModeGroupConnection** element includes three references:

1. A **RteBswProvidedModeGrpModInstRef** that references the BSW module instance.
2. A **RteBswProvidedModeGroupRef** that references the provided mode declaration group prototype. When combined with the **RteBswProvidedModeGrpModInstRef** reference this uniquely identifies the providing mode instance.
3. A **RteBswRequiredModeGroupRef** that references the required mode declaration group prototype.

## 15 Synchronizing BSW and SWC

---

The previous chapter introduced BSW components. This chapter covers how RTA-RTE supports the synchronization of BSW and SWC instances.

### 15.1 Configuration

---

Synchronization configuration consists of two steps: first, one or more `SwcBswMapping` elements must be defined that associate elements in the SWC and BSW together, and second, the created mappings must be associated with SWC/BSW instances.

#### 15.1.1 SWC to BSW Mapping

---

The `SwcBswMapping` element defines the synchronized elements of a SWC type and a BSW module description. The element “associates” elements of two internal behaviors; we’ll see later how the mapping is subsequently referenced from either a `SwcImplementation` or a `BswImplementation` and thus associated with SWC or BSW instances.

The simplest `SwcBswMapping` element simply refers to the two internal behaviors, e.g.:

```
<SWC-BSW-MAPPING>
  <SHORT-NAME>sbm1</SHORT-NAME>
  <BSW-BEHAVIOR-REF DEST='BSW-INTERNAL-BEHAVIOR'>
    ...
  </BSW-BEHAVIOR-REF>
  <RUNNABLE-MAPPINGS/>
  <SWC-BEHAVIOR-REF DEST='SWC-INTERNAL-BEHAVIOR'>
    ...
  </SWC-BEHAVIOR-REF>
  <SYNCHRONIZED-MODE-GROUPS/>
  <SYNCHRONIZED-TRIGGERS/>
</SWC-BSW-MAPPING>
```

One of the referenced internal behaviors must belong to a SWC and one to a BSW module. It is not possible to synchronize two SWCs or two BSW modules with a `SwcBswMapping` element. Within the `SwcBswMapping` element one can list synchronized runnable entities, mode groups and (external) triggers.

The `<RUNNABLE-MAPPINGS>` element encapsulates zero or more runnable-entity to executable-entity mappings, e.g.:

```
<RUNNABLE-MAPPINGS>
  <SWC-BSW-RUNNABLE-MAPPING>
    <BSW-ENTITY-REF DEST='BSW-SCHEDULABLE-ENTITY'>
      ...
    </BSW-ENTITY-REF>
    <SWC-RUNNABLE-REF DEST='RUNNABLE-ENTITY'>
      ...
    </SWC-RUNNABLE-REF>
  </SWC-BSW-RUNNABLE-MAPPING>
</RUNNABLE-MAPPINGS>
```

Each <SWC-BSW-RUNNABLE-MAPPING> synchronizes a SWC *runnable entity* with a BSW *executable entity* (either a <BSW-SCHEDULABLE-ENTITY>, <BSW-INTERRUPT-ENTITY> or <BSW-CALLED-ENTITY>). The referenced runnable or executable entity must be declared within the referenced internal behaviors.



*AUTOSAR forbids the synchronization of an SWC's runnable entity with a category 1 interrupt entity (or with a called entity invoked by a category 1 interrupt entity).*

The <SYNCHRONIZED-MODE-GROUPS> element encapsulates zero or more synchronized mode group declarations, e.g.:

```
<SYNCHRONIZED-MODE-GROUPS>
  <SWC-BSW-SYNCHRONIZED-MODE-GROUP-PROTOTYPE>
    <BSW-MODE-GROUP-REF DEST='MODE-DECLARATION-GROUP-PROTOTYPE'>
      ...
    </BSW-MODE-GROUP-REF>
    <SWC-MODE-GROUP-IREF>
      <CONTEXT-P-PORT-REF DEST='P-PORT-PROTOTYPE'>
        ...
      </CONTEXT-P-PORT-REF>
      <TARGET-MODE-GROUP-REF DEST='MODE-DECLARATION-GROUP-PROTOTYPE'>
        ...
      </TARGET-MODE-GROUP-REF>
    </SWC-MODE-GROUP-IREF>
  </SWC-BSW-SYNCHRONIZED-MODE-GROUP-PROTOTYPE>
</SYNCHRONIZED-MODE-GROUPS>
```

Each <SWC-BSW-SYNCHRONIZED-MODE-GROUP-PROTOTYPE> synchronizes a *provided* or *required* BSW mode declaration group with a particular provided SWC mode declaration group. The referenced BSW mode declaration group must be declared within the BSW type of the referenced BSW internal behavior and the referenced port prototype within the SWC type of the referenced SWC internal behavior.

The <SYNCHRONIZED-TRIGGERS> element encapsulates zero or mode synchronized trigger declarations, e.g.:

```
<SYNCHRONIZED-TRIGGERS>
  <SWC-BSW-SYNCHRONIZED-TRIGGER>
    <BSW-TRIGGER-REF DEST='TRIGGER'>
      ...
    </BSW-TRIGGER-REF>
    <SWC-TRIGGER-IREF>
      <CONTEXT-P-PORT-REF DEST='P-PORT-PROTOTYPE'>
        ...
      </CONTEXT-P-PORT-REF>
      <TARGET-TRIGGER-REF DEST='TRIGGER'>
        ...
      </TARGET-TRIGGER-REF>
    </SWC-TRIGGER-IREF>
  </SWC-BSW-SYNCHRONIZED-TRIGGER>
</SYNCHRONIZED-TRIGGERS>
```



Each <SWC-BSW-SYNCHRONIZED-TRIGGER> synchronizes a *released* or *required* BSW trigger with a particular provided SWC trigger. The referenced BSW trigger must be declared within the BSW type of the referenced BSW internal behavior and the referenced port prototype within the SWC type of the referenced SWC internal behavior.

Neither a synchronized mode group element nor a synchronized trigger can reference a *required* mode declaration group in the SWC. However as we'll see in the following sections this restrictions does not affect the expressive power – it's still possible to create associations that enable either the SWC or the BSW to be the mode manager and to trigger events in either (or both) the SWC and BSW module.

### 15.1.2 Associating a Mapping and an Instance

---

When a `SwcBswMapping` has been defined mapping it is associated with SWC/BSW types via the referenced internal behaviors. However the mapping must also be associated with instances of the SWC and BSW via one or more *implementation references*.

The `SwcImplementation` or `BswImplementation` elements can include a reference to a mapping, e.g.:

```
<SWC-IMPLEMENTATION>
...
<SWC-BSW-MAPPING-REF DEST='SWC-BSW-MAPPING'>
...
</SWC-BSW-MAPPING-REF>
...
</SWC-IMPLEMENTATION>
```

If an SWC/BSW's implementation does not include a mapping reference then RTA-RTE attempts to infer the affected instance from the referenced internal behaviors. This inference is only possible if the SWC/BSW's are instantiated exactly once and therefore when multiple instantiation is used the `SwcBswMapping` references within the implementations are mandatory.

## 15.2 Synchronized Mode Groups

---

Synchronizing mode groups permits both events in both AUTOSAR software components and basic software modules to be triggered by a single mode manager. RTA-RTE supports synchronization where the referenced BSW *provides* or *requires* the mode declaration group.

### 15.2.1 Mode Users

---

Synchronizing a *required* BSW mode group with a *provided* SWC mode group results in the BSW attaching to the mode instance defined by the SWC as a mode user. Any BSW events (e.g. `ModeSwitchEvents`) attached to the required BSW mode group will be triggered when the SWC mode manager changes mode.

The synchronized mode group element within the `SwcBswMapping` element references a provided port/mode group in the SWC. RTA-RTE supports synchronization configurations where RTE events are triggered as a result of the mode switch as well as

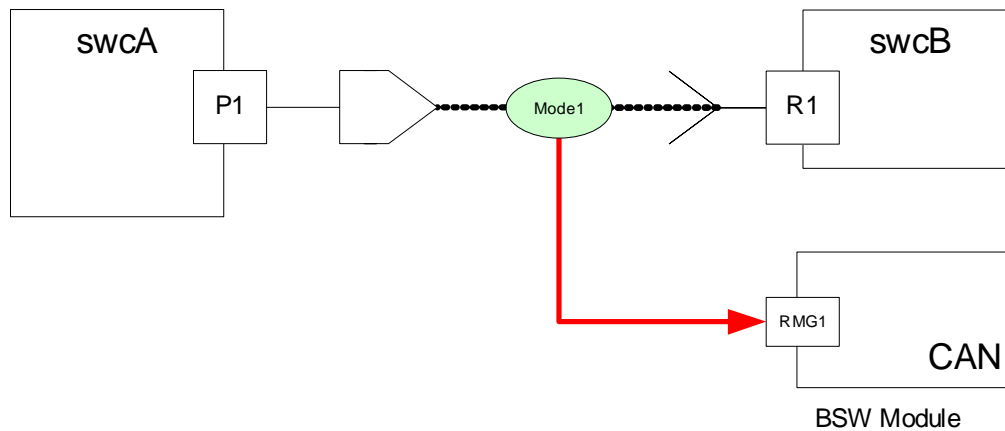


Figure 15.1: Synchronized SWC/BSW mode users

BSW events.

### 15.2.2 Mode Managers

Synchronizing a *provided* BSW mode group with a *provided* SWC mode group results in the BSW attaching to the mode instance defined by the SWC as a mode manager.



When synchronizing provided mode declarations groups then **both** the SWC and BSW become mode managers for the mode instance. AUTOSAR normally forbids multiple mode managers for a mode instance but RTA-RTE allows them if they are synchronized.

When provided BSW and SWC triggers are synchronized it is possible to suppress generation of the Rte\_Switch API by omitting the <MODE-SWITCH-POINT> element or to suppress generation of the SchM\_Switch API by omitting the <MODE-ACCESS-POINT> element.

The mode queue length for a synchronized *provided* BSW mode declaration group is derived from the associated *provided* SWC port/mode group.

## 15.3 Synchronized Triggers

Synchronizing SWC and BSW triggers permits both events in both AUTOSAR software components and basic software modules to be triggered by either SWCs or BSW. RTA-RTE supports synchronization where the referenced BSW *releases* or *requires* a trigger.

All synchronized SWC triggers are *external* triggers. RTA-RTE also supports *internal* triggers for SWCs but it is not possible to synchronize internal and BSW triggers.

### 15.3.1 Trigger Consumers

Synchronizing a *required* BSW trigger with a *provided* SWC trigger results in any <BSW-EXTERNAL-TRIGGER-OCCURRED-EVENT> events in the BSW being activated when the SWC's Rte\_Trigger API is invoked.

The synchronized trigger element within the SwcBswMapping element references a provided port/trigger in the SWC.

### 15.3.2 Trigger Producers

Synchronizing a *provided* BSW trigger with a *provided* SWC trigger results in the generated BSW SchM\_Trigger activating the same events as the Rte\_Trigger API.

When provided BSW and SWC triggers are synchronized it is possible to suppress generation of the Rte\_Trigger API by omitting the <EXTERNAL - TRIGGERING - POINT> element.



*If the SWC includes an ExternalTriggeringPoint for the provided trigger then **both** the SWC's Rte\_Trigger and the BSW's SchM\_Trigger APIs are created. AUTOSAR normally forbids multiple trigger providers but RTA-RTE allows them if they are synchronized.*

## 15.4 Synchronized Runnable Entities

Synchronizing a SWC *runnable entity* to a BSW *executable entity* allows the executable entity to use SWC-API calls such as those related to port-based communication. The SWC is also able to use BSW-API calls.

### 15.4.1 Sender/Receiver

RTA-RTE supports the synchronization of both Sender and Receiver runnable entities with an executable entity, thus an executable entity will be able to make calls to the Sender/Receiver APIs (e.g. Rte\_Write/Rte\_Read).



*Since it is not possible to associate an executable entity with a <DATA - RECEIVED - EVENT>, a receiver executable entity must be started using another event type.*

### 15.4.2 Client/Server

RTA-RTE supports synchronization of executable entities with *client* runnables only (Synchronizing an executable entity with a server runnable makes no sense as there is no way for the executable entity to gain access to the server arguments). A synchronized executable entity can therefore call the *client* API available to the synchronized client runnable.

### 15.4.3 Exclusive Areas

RTA-RTE supports the synchronization of SWC runnable entities that access an exclusive area with executable entities. When synchronized the executable entity will be able to make calls to the Enter/Exit APIs (e.g. Rte\_Enter/Rte\_Exit) and thus “share” access to the same shared data.

Figure 15.2 illustrates a runnable Re2 that both accesses exclusive area EA and is synchronized with schedulable entity **SE**. RTA-RTE will generate Rte\_Enter\_EA and Rte\_Exit\_EA that can be used to control access to the shared data governed by “EA” – depending on the selected implementation mechanism for the exclusive area and the

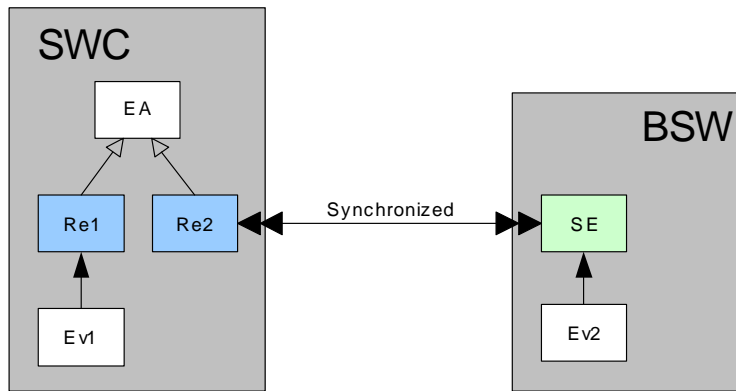


Figure 15.2: Synchronized runnable and executable accessing an Exclusive Area

configured task mappings this could be access to an OS resource, interrupt blocking or a null implementation.

## 16 Accessing NVRAM

---

We saw in Chapter 11 how to create an Nv-block SWC type that containing descriptions of one or more Nv-blocks. In this chapter we'll look at how application SWC can access declared Nv-blocks using port-based communication.

### 16.1 Configuration

---

Communication between application software components occurs over ports and therefore before accessing Nv-data mirrors from an application SWC suitable ports must be declared within the Nv-block SWC and mapped to the required Nv-data ram-block.

#### 16.1.1 Ports

---

Access to non-volatile data by application SWCs occurs through one or more ports declared by the NvBlock software component type. These ports are categorized by Nv-data interfaces – see Section 6.2.

#### 16.1.2 Mapping

---

As well as declaring the ram-block, an Nv-block descriptor also declares the mappings that associate rPort or pPorts within the Nv-Block SWC type and the NvBlockDescriptor's ram-block. The mapping is created using an NV-BLOCK-DATA-MAPPING element:

```
<NV-BLOCK-DATA-MAPPING>
  <NV-RAM-BLOCK-ELEMENT>
    <LOCAL-VARIABLE-REF ...
  </NV-RAM-BLOCK-ELEMENT>
  <READ-NV-DATA>
    ...
  </READ-NV-DATA>
  <WRITTEN-NV-DATA>
    ...
  </WRITTEN-NV-DATA>
</NV-BLOCK-DATA-MAPPING>
```

The mapping's <NV-RAM-BLOCK-ELEMENT> references the Nv-block descriptor's ram-block. RTA-RTE supports a reference made using the following elements.

- <LOCAL-VARIABLE-REF> – maps the port Nv-data to an entire ram-block. The ram-block can be typed by either an application or an implementation data type.
- <AUTOSAR-VARIABLE-IN-IMPL-DATATYPE> – maps the port Nv-data to a ram-block and, optionally, defines a specific implementation data type sub-element within the ram-block for the mapping.
- <AUTOSAR-VARIABLE-IREF> – maps the port Nv-data to a ram-block using an application data type.

Using a sub-element mappings enables efficient use of large Nv-blocks since a single structure representing the entire block can be accessed from multiple ports each

with a mapping to one sub-element of the large single structure. The multiple NV-BLOCK-DATA-MAPPINGS elements for the ram-block means that individual SWCs can independently update just parts of the large Nv-data block.

The mapped sub-element can be any member of a record – it does not have to be a leaf element and can be typed by either a primitive type (e.g. an integer) or can itself be a record.

The association with the port and datum that an application SWC uses to read/write to the ram-block is through either the <WRITTEN-NV-DATA> element (for rPorts) or the <READ-NV-DATA> element (for pPorts). For example:

```
<WRITTEN-NV-DATA>
  <AUTOSAR-VARIABLE-IREF>
    <PORT-PROTOTYPE-REF ...
    <TARGET-DATA-PROTOTYPE-REF ...
  </AUTOSAR-VARIABLE-IREF>
</WRITTEN-NV-DATA>
```

A data mapping element can declare both read and write mappings for the same ram-block. Since read mappings reference pPorts and write mappings reference rPorts then, clearly, different ports can be specified for the mappings.

### Example

To illustrate how NVRAM block data mappings and ports interact consider the following NvBlockDescriptor within NvBlockSwComponentType “swc\_nvblock” that declares ram-block “myRam1”:

```
<NV-BLOCK-DESCRIPTOR>
  <SHORT-NAME>block1</SHORT-NAME>
  <NV-BLOCK-DATA-MAPPINGS>
    ...
  </NV-BLOCK-DATA-MAPPINGS>
  <RAM-BLOCK>
    <SHORT-NAME>myRam1</SHORT-NAME>
    <TYPE-TREF>/Types/Struct1</TYPE-TREF>
  </RAM-BLOCK>
</NV-BLOCK-DESCRIPTOR>
```

Where Struct1 contains two members “A” and “B”. Furthermore, assume that the NvBlockSwComponentType has a require port “r1” containing an Nv-data item “datum” with the same type as structure member “A”. An NvBlockDataMapping can then be established from “r1” to “A” as follows:

```
<NV-BLOCK-DATA-MAPPING>
  <NV-RAM-BLOCK-ELEMENT>
    <AUTOSAR-VARIABLE-IN-IMPL-DATATYPE>
      <ROOT-VARIABLE-DATA-PROTOTYPE-REF>
        /pkg/swc_nvblock/block1/myRam1
      </ROOT-VARIABLE-DATA-PROTOTYPE-REF>
    <TARGET-DATA-PROTOTYPE-REF>
```

```

        /Types/Struct1/A
    </TARGET-DATA-PROTOTYPE-REF>
</AUTOSAR-VARIABLE-IN-IMPL-DATATYPE>
</NV-RAM-BLOCK-ELEMENT>
<WRITTEN-NV-DATA>
    <AUTOSAR-VARIABLE-IREF>
        <PORT-PROTOTYPE-REF>/pkg/swc_nvblock/r1</PORT-PROTOTYPE-REF>
        <TARGET-DATA-PROTOTYPE-REF>/pkg/itf/datum</TARGET-DATA-PROTOTYPE-REF>
    </AUTOSAR-VARIABLE-IREF>
</WRITTEN-NV-DATA>
</NV-BLOCK-DATA-MAPPING>

```

When the mapping is established any writes to the ram-block through required port “r1” and datum “datum” will write directly to structure member “A” and not affect any other element of the ram-block structure.

### 16.1.3 Composition

The ports on the Nv-block SWC are connected to sender-receiver or Nv-data ports on the application SWC. The two ports must be either typed by compatible interfaces (the provider can contain a superset of the requirer’s elements) or a port-interface mapping provided to associate elements within the two interfaces.

## 16.2 Access from Application SWCs

Using appropriate port connections application SWC can read and/or write Nv-data ram-blocks.



*All accessing SWCs must be mapped to the same partition (i.e. OsApplication).*

### 16.2.1 Reading NVRAM data

An application SWC reads data from an Nv-block by:

1. Declare a Sender-receiver **rPort** (with an appropriate interface) and connecting it to a **pPort** on the Nv-Block SWC categorized by an Nv-Data interface.
2. Establish a data-read point or data-access point to create the RTE API to read the NV data.

The RTE generator will then create Rte\_Read, Rte\_DRead, etc. API calls that will directly read the ram-block.

As an example, assume that the configuration contains an NvBlockSwComponentType “swc\_nvblock” declaring a single NvBlockDescriptor “block1”. The Nv-block software component can then declare a provide port “p1” that permits read access to the ram-block by application SWCs as follows.

First, an `NvDataInterface` is created; this will be used by the `NvBlock SWC` to type the provided port:

```
<NV-DATA-INTERFACE>
  <SHORT-NAME>ifnv1</SHORT-NAME>
  <NV-DATAS>
    <VARIABLE-DATA-PROTOTYPE>
      <SHORT-NAME>diN1</SHORT-NAME>
      <TYPE-TREF>/Types/UInt16</TYPE-TREF>
    </VARIABLE-DATA-PROTOTYPE>
  </NV-DATAS>
</NV-DATA-INTERFACE>
```

Next, the `NvBlock SWC` is configured with the provide port “p1” typed by interface “ifnv1”:

```
<NV-BLOCK-SW-COMPONENT-TYPE>
  <SHORT-NAME>swc_nvblock</SHORT-NAME>
  <PORTS>
    <P-PORT-PROTOTYPE>
      <SHORT-NAME>p1</SHORT-NAME>
      <PROVIDED-INTERFACE-TREF>/pkg/ifnv1</PROVIDED-INTERFACE-TREF>
    </P-PORT-PROTOTYPE>
  </PORTS>
  <NV-BLOCK-DESCRIPTORS>
    ...
  </NV-BLOCK-DESCRIPTORS>
</NV-BLOCK-SW-COMPONENT-TYPE>
```

Finally, the `<NV-BLOCK-DESCRIPTOR>` can be configured to both declare `ram-block` “myRamBlock1” and to map provide port “p1” to read the `ram-block`.

```
<NV-BLOCK-DESCRIPTOR>
  <SHORT-NAME>block1</SHORT-NAME>
  <NV-BLOCK-DATA-MAPPINGS>
    <NV-BLOCK-DATA-MAPPING>
      <NV-RAM-BLOCK-ELEMENT>
        <LOCAL-VARIABLE-REF'>/pkg/swc_nvblock/block1/myRamBlock1</LOCAL-
          VARIABLE-REF>
      </NV-RAM-BLOCK-ELEMENT>
      <READ-NV-DATA>
        <AUTOSAR-VARIABLE-IREF>
          <PORT-PROTOTYPE-REF>/pkg/swc_nvblock/p1</PORT-PROTOTYPE-REF>
          <TARGET-DATA-PROTOTYPE-REF>/pkg/ifnv1/diN1</TARGET-DATA-PROTOTYPE-
            REF>
        </AUTOSAR-VARIABLE-IREF>
      </READ-NV-DATA>
    </NV-BLOCK-DATA-MAPPING>
  </NV-BLOCK-DATA-MAPPINGS>
  <RAM-BLOCK>
    <SHORT-NAME>myRamBlock1</SHORT-NAME>
    <TYPE-TREF>/Types/UInt16</TYPE-TREF>
  </RAM-BLOCK>
```



</NV-BLOCK-DESCRIPTOR>

The port “p1” can then be connected to require ports on application SWCs to enable the ram-block data to be read. Multiple application SWCs can connect to the same provided port on an NvBlockSwComponentType to share read-access to the same Nv-block.

### 16.2.2 Writing NVRAM data

---

An application SWC writes data to an Nv-block by:

1. Declare a Sender-receiver **pPort** (with an appropriate interface) and connecting it to an **rPort** on the Nv-Block SWC categorized by an Nv-Data interface.
2. Establish a data-send point or data-write point to create the RTE API to write the NV data.

The RTE generator will then create Rte\_Write or Rte\_IWrite API calls that will update the ram-block.

### 16.3 Access from the NVRAM manager

---

RTA-RTE creates two APIs for the ram-block in each NvBlockDescriptor that enable the AUTOSAR NVRAM Manager to read from, and write to, the ram-block.



*The generated API names include the component prototype name and thus do not appear in the application header file as they cannot form part of the Nv-block SWC's contract.*

#### 16.3.1 Reading the Ram-block

---

The NVRAM manager can use the created Rte\_GetMirror API to read the current ram-block contents to a defined location. The generated signature of the API is:

```
FUNC (VAR (Std_ReturnType, AUTOMATIC), RTE_CODE)  
Rte_GetMirror_<c>_<n> (P2VAR (uint8, AUTOMATIC, RTE_DATA) NVMBuffer)
```

The NVMBuffer parameter defines to where the data is written. The <c> element of the name is the component prototype name and the <n> element the name of the NvBlockDescriptor name.

#### 16.3.2 Writing the Ram-block

---

The NVRAM manager can use the created Rte\_SetMirror API to update the current ram-block contents from a defined location. The generated signature of the API is:

```
FUNC (VAR (Std_ReturnType, AUTOMATIC), RTE_CODE)  
Rte_SetMirror_<c>_<n> (P2VAR (uint8, AUTOMATIC, RTE_DATA) NVMBuffer)
```

The NVMBuffer parameter defines from where the data is copied. The <c> element of the name is the component prototype name and the <n> element the name of the NvBlockDescriptor name.

### 16.3.3 Notifications from the NVRAM Manager

In addition to the `Rte_GetMirror` and `Rte_SetMirror` APIs, RTA-RTE can also generate additional APIs used by the NVRAM manager to pass notifications to applications SWCs.

The APIs are generated in response to declared *role based port assignments* for required client-server ports. For each such assignment, RTA-RTE generates an API to be invoked by the NVRAM manager. The name of the API is based on the declared *role*, the `NvBlock` SWC component prototype name and the `NvBlockDescriptor` name.

```
FUNC(VAR(Std_ReturnType, AUTOMATIC), RTE_CODE)  
Rte_<role>_<c>_<n>(<args>);
```

The `<role>` element of the API name is declared within the role based assignment. The `<c>` element of the name is the component prototype name and the `<n>` element the name of the `NvBlockDescriptor` name.

The API's formal argument list, `<args>`, is derived from the **first** operation in the interface referenced by the require port with the exception of the roles `NvMNotifyJobFinished` and `NvMNotifyInitBlock` in which case the formal argument list is standardized by AUTOSAR.

The generated role API invokes the server through the referenced require client-server port. If multiple role based assignments have the same "role" then the generated API aggregates invocation of all referenced servers.



*The generated API invokes all servers synchronously. To avoid use of `OsEvents` it is strongly recommended to use direct function invocation by omitting the task reference the relevant RTE event mapping.*

The generated API takes no action if the RTE is not started or if the RTE has been stopped.

### 16.3.4 Invocations of the NVRAM Manager by SWCS

As well as the APIs for passing notifications from the NVRAM manager to application SWCs, RTA-RTE also supports the invocation of NVRAM manager APIs from SWCs through generated server runnables for provided client-server ports..

RTA-RTE generates the runnable entity body for all *connected* provide ports that have a configured `OperationInvokedEvent`. The generated runnable entity body uses the symbol name from the runnable entity declaration – this must therefore be globally unique. The formal argument list consists of three elements:

1. An optional instance handle (included only when the `NvBlock` SWC is declared as supporting multiple instances, see Section 8.14).
2. The port-defined arguments (for the referenced server port, see Section 8.13.2).
3. The operation arguments from the operation in the server's ports interface with the **same name** as the "role".

The generated body for the runnable invokes the C-based API of the NVRAM manager based on the declared “role”. All runnable parameters, apart from the instance handle, are passed directly to the invoked NVRAM API.

# **Part V**

# **Deployment**

## 17 Defining the ECUs and the Networks

Before any composition can be mapped to ECUs on a vehicle network we need to define the components from which the physical system is created. We will do this in Chapter 18. In this chapter we show how to define the parts from which it will be built.

This is the same AUTOSAR mechanisms as we saw when building the software system as a composition: first we had to define the types of the object we could use (our software components) before the composition could be built from instances of those objects (software component instances).

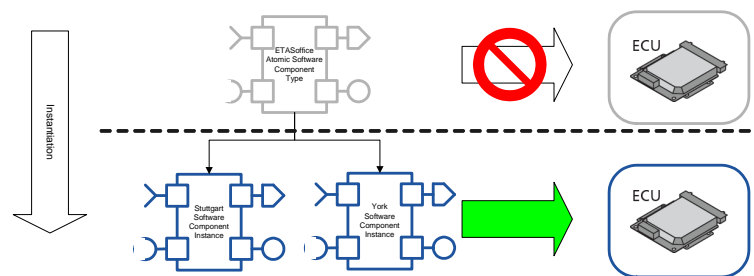


Figure 17.1: Instances for Hardware Mapping

For the ECU and the network we must define:

- What types of ECU exist and their physical characteristics.
- What ECU instances exist.

This information is contained in <ECU>, <ECU-INSTANCE> elements. A reference from the <ECU-CONFIGURATION> element identifies the appropriate <SYSTEM> element which in turn references the associate <ECU-INSTANCE> element.

```

<AR-PACKAGE>
  <SHORT-NAME>pkg</SHORT-NAME>
  <ELEMENTS>
    <ECU>
      <SHORT-NAME>eType</SHORT-NAME>
      ...
    </ECU>
    <ECU-INSTANCE>
      <SHORT-NAME>eInst</SHORT-NAME>
      ...
    </ECU-INSTANCE>
    <!-- System element references from ECU-CONFIGURATION
         using an ECU-EXTRACT-REF -->
    <SYSTEM>
      ...
      <ECU-INSTANCE-REF>/pkg/eInst</ECU-INSTANCE-REF>
    </SYSTEM>
  </ELEMENTS>
</AR-PACKAGE>

```

While it possible to put software component configurations and system deployment elements in the same AUTOSAR XML file, it is more common to separate the distinct aspects of system engineering into (at least) two separate files. This means that multiple different software component allocations can be managed easily from a single “repository” of software components.

## 17.1 ECU Type Definition

---

Every configuration you make will need to define at least one ECU type. An ECU is conceptually similar to an SWC-type in that it defines the general properties of an abstract ECU. Similarly, it is an instance of the ECU onto which software component prototypes are allocated.

The definition provides a name for the ECU type so that it can be referenced from the <SYSTEM> description as well as including a reference to a <COMPOSITION-TYPE> definition that indicates the mapped software component prototypes and their types.

```
<ECU>
  <SHORT-NAME>ECUType</SHORT-NAME>
  <PORTS>
    ...
  </PORTS>
</ECU>
```

### 17.1.1 Communication Ports

---

When the ECU is not stand alone it must provide the opportunity to create instances of communication ports. <PORTS> element allows the specification of the physical ECU interface(s) provided by the ECU, for example, <ECU-COMMUNICATION-PORT>, <PERIPHERAL-HW-PORT> etc.

Only the <ECU-COMMUNICATION-PORT> definitions are relevant for RTE configuration as they provide the definition of which physical communication interfaces exist on the ECU.

Note that the communication port does not specify that an ECU has a port. The declaration simply states that an instance of the ECU can be created that can create instances of the defined port. In this regards, the communication port is analogous to a type.

It is only necessary to define an <ECU-COMMUNICATION-PORT> for each different class of communication port on the ECU. For example, if your ECU has 4 CAN channels but all the channels have the same properties then you would only define a single <ECU-COMMUNICATION-PORT>.

Each communication port is named and defined the <NUMBER>, <SIZE> and <TYPE> of the physical buffers as well as their <DIRECTION>. These are not required for the RTE but are mandatory in the ECU configuration description XML so must be provided. The following example shows the general configuration:

```
<ECU-COMMUNICATION-PORT>
  <SHORT-NAME>CANPortType</SHORT-NAME>
```

```
<!-- Remaining configuration mandatory
      but not used by RTE -->
<DIRECTION>IN</DIRECTION>
<BUFFERS>
  <BUFFER>
    <NUMBER>1</NUMBER>
    <SIZE>5</SIZE>
    <TYPE>input</TYPE>
  </BUFFER>
</BUFFERS>
</ECU-COMMUNICATION-PORT>
```



*It is outside the scope of this user document to describe how additional ECU hardware is defined in XML.*

## 17.2 ECU Instances

The `<ECU-INSTANCE>` element instantiates an ECU type (defined within the ECU description—see Section 17.1):

```
<ECU-INSTANCE>
  <SHORT-NAME>ECUI</SHORT-NAME>
  <CONNECTORS>
    <COMMUNICATION-CONNECTOR>
      <SHORT-NAME>comCon1</SHORT-NAME>
      <ECU-COMM-PORT-INSTANCES/>
    </COMMUNICATION-CONNECTOR>
  </CONNECTORS>
</ECU-INSTANCE>
```

Although this is documented in the AUTOSAR Specification, it does not refer to an ECU type, and RTA-RTE does not attempt to process this element.

## 18 Mapping Software Components to ECUs

A <SYSTEM> element is used to map the software component model defined by a <COMPOSITION-SW-COMPONENT-TYPE> onto the physical system defined by the referenced <ECU-INSTANCE>.



*RTA-RTE does not provide support for determining an appropriate system definition. This information must be known by the ECU Integrator before the RTE Generation phase.*

The <SYSTEM> specifies how component prototypes in the <COMPOSITION-SW-COMPONENT-TYPE> are mapped into an <ECU-INSTANCE>.

```
<SYSTEM>
  <SHORT-NAME>MySystem</SHORT-NAME>
  <MAPPING>
    <SHORT-NAME>MyMapping</SHORT-NAME>
    <DATA-MAPPINGS />
    <SW-MAPPINGS>
      <SWC-TO-ECU-MAPPING>
        <SHORT-NAME>map_ECU1</SHORT-NAME>
        <COMPONENT-IREFS>
<!-- component instances are mapped to the ECU here -->
          </COMPONENT-IREFS>
          <ECU-INSTANCE-REF DEST='ECU-INSTANCE'>/MyPackage/ECUInstance1</ECU-
            INSTANCE-REF>
        </SWC-TO-ECU-MAPPING>
      </SW-MAPPINGS>
    </MAPPING>
  <ROOT-SOFTWARE-COMPOSITION>
    <SHORT-NAME>MyCompositionInstance</SHORT-NAME>
    <SOFTWARE-COMPOSITION-TREF DEST='COMPOSITION-SW-COMPONENT-TYPE'>/MyPackage
      /MyCompositionType</SOFTWARE-COMPOSITION-TREF>
  </ROOT-SOFTWARE-COMPOSITION>
</SYSTEM>
```

Note that the <SYSTEM> element contains a <ROOT-SOFTWARE-COMPOSITION> element, which represents an instance of the CompositionType referred to in its <SOFTWARE-COMPOSITION-TREF>.

### 18.1 Mapping Component Prototypes

The <SYSTEM-MAPPING> element encapsulates one or more <SWC-T0-ECU MAPPING> elements. Each <SWC-T0-ECU MAPPING> element defines a mapping of software component prototypes to an ECU instance. For a particular RTE generation phase there is only one relevant ECU instance.

```
<SWC-T0-ECU-MAPPING>
  <SHORT-NAME>map_ECU1</SHORT-NAME>
  <!-- Map these component instances...-->
  <COMPONENT-IREFS>
    <COMPONENT-IREF>
```



```

    <CONTEXT-COMPOSITION-REF DEST='SOFTWARE-COMPOSITION'>/MyPackage/
      MyCompositionInstance</CONTEXT-COMPOSITION-REF>
    <TARGET-COMPONENT-REF DEST='COMPONENT-PROTOTYPE'>/MyPackage/
      MyCompositionType/SenderPrototype</TARGET-COMPONENT-REF>
  </COMPONENT-IREF>
<COMPONENT-IREF>
  <CONTEXT-COMPOSITION-REF DEST='SOFTWARE-COMPOSITION'>/MyPackage/
    MyCompositionInstance</CONTEXT-COMPOSITION-REF>
  <TARGET-COMPONENT-REF DEST='COMPONENT-PROTOTYPE'>/MyPackage/
    MyCompositionType/ReceiverPrototype</TARGET-COMPONENT-REF>
</COMPONENT-IREF>
</COMPONENT-IREFS>
<!-- Onto this ECU instance -->
<ECU-INSTANCE-REF DEST='ECU-INSTANCE'>/MyPackage/ECUInstance1</ECU-INSTANCE-
  REF>
</SWC-TO-ECU-MAPPING>

```

A <COMPONENT-IREF> specifies an atomic software component prototype in the full context of the (possibly nested) software composition.



*An atomic software component prototype is one whose type is not a software composition, i.e. its <TYPE-TREF> indicates an application software component type.*

The first element within a <COMPONENT-IREF> is the <CONTEXT-COMPOSITION-REF>, which points to the top-level software composition—the <ROOT-SOFTWARE-COMPOSITION> element within the <SYSTEM>. Each of the following elements is a reference to a component prototype within the composition type corresponding to the previous reference.

If the composition is nested, it may be necessary to insert one or more <CONTEXT-COMPONENT-REF> elements. Each component prototype ref specifies a component prototype which is itself an instance of a composition type. One such <CONTEXT-COMPONENT-REF> is needed for each level of nesting until we reach the composition type containing the atomic software component instance.

Finally there must be a <TARGET-COMPONENT-REF>. This refers to the component prototype that is an instance of an application software component type. This completely specifies the software component instance and completes the <COMPONENT-IREF>.

The <ECU-INSTANCE-REF> references the <ECU-CONFIGURATION> element to which the mapping applies.

## 18.2 SWC Implementation Selection

RTA-RTE supports the *SwcToImplMapping* element to choose which (of potentially many) implementations applies to a SW-C type.

Each SW-C to implementation mapping selects an implementation for one or more SW-C **instances** and hence, indirectly, for the SW-C type. For example, to select im-

plementation “impl” for instance “swcA” one could use:

```
<SWC-TO-IMPL-MAPPING>
  <SHORT-NAME>impl_A</SHORT-NAME>
  <COMPONENT-IMPLEMENTATION-REF>/pkh/impl</COMPONENT-IMPLEMENTATION-REF>
  <COMPONENT-IREFS>
    <COMPONENT-IREF>
      <CONTEXT-COMPOSITION-REF>/pkg/sys/SwComp</CONTEXT-COMPOSITION-REF>
      <TARGET-COMPONENT-REF>/pkg/Compo/swcA</TARGET-COMPONENT-REF>
    </COMPONENT-IREF>
  </COMPONENT-IREFS>
</SWC-TO-IMPL-MAPPING>
```

The SW-C to implementation mappings are contained within the System Mapping element. See Section 18.1 for detailed information on mapping SW-C prototypes to ECUs.

### 18.3 Configuring Service Components on an ECU

Application software components are mapped to ECUs, as described above, using the <SW-MAPPINGS> element. In contrast, service components are assigned to ECUs as part of the ECU configuration process. This is done using the XML element <ECU-SW-COMPOSITION>, which is referenced from the ECU configuration itself.

```
<ECU-CONFIGURATION>
  <SHORT-NAME>ECU1Config</SHORT-NAME>
  <ECU-EXTRACT-REF>/autosar/system1 </ECU-EXTRACT-REF>
  <ECU-SW-COMPOSITION-REF>/pkg/ECU1SwComp</ECU-SW-COMPOSITION-REF>
  ...
</ECU-CONFIGURATION>

<ECU-SW-COMPOSITION>
  <SHORT-NAME>ECU1SwComp</SHORT-NAME>
  <COMPONENTS>
    ...
  </COMPONENTS>
  <CONNECTORS>
    ...
  </CONNECTORS>
</ECU-SW-COMPOSITION>
```

The <ECU-SW-COMPOSITION> element is very similar to the <ROOT-SOFTWARE-COMPOSITION> described in Section 13.1, except that it only contains the service components and service connectors.

```
<COMPONENTS>
  <SERVICE-COMPONENT-PROTOTYPE>
    <SHORT-NAME>myService</SHORT-NAME>
    <SERVICE-COMPONENT-TREF>/pkg/svcType</SERVICE-COMPONENT-TREF>
  </SERVICE-COMPONENT-PROTOTYPE>
</COMPONENTS>
```

A connection between a service component prototype and an application component prototype is established using a <SERVICE-CONNECTOR-PROTOTYPE> element.

```
<CONNECTORS>
  <SERVICE-CONNECTOR-PROTOTYPE>
    <SHORT-NAME>serviceConnector1</SHORT-NAME>
    <APPLICATION-PORT-IREF.../>
    <SERVICE-PORT-IREF.../>
  </SERVICE-CONNECTOR-PROTOTYPE>
</CONNECTORS>
```

This `<ECU-SW-COMPOSITION>` element of the ECU configuration is used by RTA-RTE to determine which services are required in an application, and to establish the connections between these services and the application software components.

## 18.4 Mapping Runnable Entities to Tasks

---

Mapping software component instances to ECU instances does not provide sufficient information for the ECU to be integrated. Recall that a software component is multi-threaded because it can contain multiple runnables.

The model of concurrency represented by runnable entities in software components needs to be mapped to the model of concurrency supported by the ECU. In AUTOSAR, this means that runnable entities must be mapped to tasks in the AUTOSAR OS.

The RTE generator uses the mapping information in the ECU configuration description to map runnable entities into named tasks which are then created automatically by RTA-RTE.

The process involves three parts:

1. Defining the Os tasks within the Os module configuration and their mutual concurrency.
2. Mapping the runnable entities into the defined tasks. This is done in the Rte module configuration.
3. Defining any supporting Os objects that the RTE needs as a consequence of the mapping. For example it may need Os objects to schedule the tasks, to execute only the activated runnables in each task execution and to serialize access to application / BSW module code and/or internal state. RTA-RTE optionally provides this additional OS configuration as an output, see Chapter 20.

### 18.4.1 Task Definition

---

RTA-RTE uses the OS module configuration within your ECU-Configuration description to determine the available tasks. Tasks into which runnable entities are mapped will have their bodies generated by the RTA-RTE RTE generator.

Each task is declared in the ECU configuration description for the OS within an instance of the container type:

```
/AUTOSAR/0s/0sTask
```

You can create as many tasks as required for the RTE, up to a maximum of 256 or the limit imposed by your AUTOSAR OS whichever is the smaller.

#### 18.4.2 Configuration

---

The mapping of a runnable entity instance to an OS tasks occurs within an instance of the container type:

```
/AUTOSAR/EcucDefs/Rte/RteSwComponentInstance/RteEventToTaskMapping
```

The runnable entity mapping container defines the parameters related to the mapping and references the appropriate runnable entity:

- **PositionInTask**—The position of the runnable entity within the task (this must be specified as an unsigned integer).
- **RTEEventRef**—The RTE Event responsible for activating the runnable entity name. This must be specified as a single, absolute, target reference to the RTE Event within the SW-C internal behaviour definition.
- **MappedToTaskRef**—The OS task to which the runnable entity is mapped (this must be specified as an absolute reference to the OS Task definition within the OS configuration container).

A single task may execute one or more runnable entities.

The relevant `SoftwareComponentInstanceRef` is taken from the `/AUTOSAR/EcucDefs/Rte/RteSwComponentInstance` that encapsulates the runnable entity mapping.

To avoid ambiguity, all runnable entities mapped to a task must have unique positions within the task. The positions specified for runnable entities within a task need not be contiguous.

#### 18.4.3 Hints for Mapping

---

Multiple Category 1 (Basic) runnable entities can be mapped to the same task as they cannot block and have a finite execution time. It is assumed that an AUTOSAR application will contain multiple tasks, each of which is dedicated to running several basic runnable entities.

Mapping multiple Category 2 (Extended) runnable entities to a single task is not recommended since they can block and therefore significantly affect the timing (jitter) of subsequent runnable entities mapped to the task. For this reason it is recommended to map each Category 2 runnable entity to a single OS task.

A single task may perform multiple roles; e.g. executing different event-triggered runnable entities or executing several runnable entities in sequence.

When defining the mapping of runnable entities to tasks the following rules should be followed:

- Category 1 runnable entities can be mapped to either “basic” or “extended” tasks.
- There should be a 1:1 mapping between Category 2 runnable entities and “extended” tasks.
- To minimize RTE generated code within task bodies, time-triggered runnable entities with different periods should be mapped to different tasks and time-triggered and communication-triggered runnable should be mapped to different tasks.
- All tasks should be mapped to the same OS application.

#### 18.4.4 Mode Switch events

---

AUTOSAR requires that all runnables triggered by ModeSwitchEvents for the same mode instance must be mapped to the same task. The runnables must be positioned within the task such that all “on-exit” runnables occur first, followed by all “on-transition” runnables, and finally all “on-exit” runnables.



*The order in which ModeSwitchEvent triggered runnables are mapped to tasks defines the order in which they are invoked at run-time.*

#### 18.4.5 Avoiding Category 2 (Extended) Tasks

---

The RTE generator is required to generate task bodies that contain the invocations of runnable entities. Depending on the runnable placement within tasks the generated body can use OS events which mean that the task is an “extended” task.

It may be desired to avoid the creation of extended tasks. These can be avoided by:

- Do not use WaitPoints within runnable entities.
- Do not use synchronous inter-task or inter-ECU client-server calls.
- Do not map runnable entities triggered by timing events and runnable entities triggered by other events (other than OperationInvokedEvent when the client is in the same task) to the same task. When such a mapping exists, the RTA-RTE RTE generator uses extended tasks and OS events to ensure the correct runnable is executed in response to each RTE event.
- Do not map multiple alarm-triggered runnable entities to the same task.

### 18.5 How Runnables get activated

---

There are subtle differences between how task activation works in the AUTOSAR OS and how runnable-entity activation works in AUTOSAR RTE and ETAS RTA-RTE. This section explains the issues.

### 18.5.1 Task Activation in AUTOSAR OS

---

AUTOSAR OS specifies a counted, limited task-activation scheme where the ActivateTask API increments an activation counter (up to a pre-configured limit) and the counters are used to decide what task should be executed next whenever a Dispatch Point is encountered (e.g. call to certain OS APIs, termination of a Task). Importantly, the activation count is not decremented until the task has run to termination.

In particular:

- If the activation limit for a task is set to 1, then an attempt to activate a task while it is already running will fail with E\_OS\_LIMIT.
- If the activation limit for the task is set to  $n$ , then effectively the activations are queued up to  $n$  activations. Note that while the task is executing that execution occupies one of these 'queue' positions.

### 18.5.2 Runnable Activation in AUTOSAR RTE 4.0.3

---

Runnable activation is complicated by the fact that an activation should not be lost if it occurs during execution of the runnable [AR 4.0.3 SWS\_RTE Figure 4.17]. Additionally, multiple runnables can be mapped to a single task and AUTOSAR 4.0.3 specifies that each RteEvent has an independent state-machine [AR4.0.3 SWS\_RTE Page 113].

General Case: Sender-receiver, etc.

---

Aside from the special cases of OperationInvokedEvent and ModeSwitchEvent (see below), AUTOSAR 4.0.3 specifies an uncounted runnable-activation scheme whereby a runnable becomes eligible to run when any of the relevant configured events occurs. Notionally, a flag is set when the RteEvent occurs, then tested-and-reset in Rte-generated task body before calling the C-function that implements the runnable.

In particular:

- If a runnable is activated multiple times before it enters, it is entered once.
- If a runnable is activated while it is executing, the activation is not lost, and the runnable is eligible to run again.

Special Case: OperationInvokedEvent

---

Server runnables (i.e. those started by OperationInvokedEvent) participate in a queued activation scheme [AR 4.0.3 SWS\_RTE Page 113]. Here, subsequent client calls can be made before previous calls are resolved, and RTE will activate the server runnable multiple times, once per client request (subject to the configured queue length).

Special Case: ModeSwitchEvent

---

ModeSwitchEvents participate in a batched activation scheme [AR 4.0.3 SWS\_RTE Section 4.4.4] whereby all the runnables configured for a particular mode transition (in-

cluding OnExit, OnTransition or OnEnter) are executed in strict PositionInTask order after the mode transition conditions are met.

In particular:

- If a mode transition is initiated when a task containing the relevant Mode Switch runnables is already executing and has passed the point of checking the conditions for some runnable relevant to that transition, no runnable for that transition will be executed on that pass through the task body. The task body will be re-executed when the outstanding mode transition is detected and the runnables will all be executed in that execution.



*Activation Flags may still be required for ModeSwitchEvents to support MinimumStartInterval or ModeDisablingDependencies.*

#### Special Case: Queued Sender-Receiver

In fact, queued sender-receiver is activated in the same pattern as the general case, that is, Activation is not queued. It is expected that the code within the runnable itself will loop until the queue is empty [AR 4.0.3 SWS\_RTE Section 4.4.4].

### 18.5.3 Runnable Activation in ETAS RTA-RTE

Depending on the nature of the RteEvents mapped to a given OsTask, ETAS RTA-RTE may employ a variety of mechanisms in the task glue code to ensure that the runnables are activated correctly.

#### When all Mapped RteEvents are Periodic

In this case the OsTask need only be Basic with an Activation Limit of 1. If the periods of the mapped runnables differ then prescalers are implemented as necessary.

#### When Periodic and Sporadic RteEvents are mapped or some Category 2 runnable is mapped

In this case the OsTask must be Extended and auto-started at OS start-up. The glue code does not terminate and the notional Activation Flags are implemented by OsEvents.

#### Sporadic RteEvents only

When only category 1 runnables started by sporadic events are mapped to a task, the OsTask need only be Basic and RTA-RTE uses a system of Activation Flags to keep track of what runnables are eligible to run. When the task to which an RteEvent has been mapped executes, RTE glue code in the task body tests and resets the flag before executing the runnable body. Even in the case where exactly one RteEvent is mapped to the task, the flag is necessary to support the case where the runnable is activated while it is executing [AR 4.0.3 SWS\_RTE Figure 4.17].

- When this scheme is used, RTA-RTE requires that the Activation Limit of the OsTask

is 2, and will write a warning message if the OS configuration does not satisfy this.

The Activation limit of 2 is used so that a task activation is not lost if the runnable is activated during the task's execution. If the task is executing and has passed the point of testing the runnable's entry conditions, then a runnable activation results in setting the runnable activation flag in the RTE, and incrementing the Task's Activation Count in the OS (remember, the Activation Count is already 1 if the Task is executing). When the Task terminates, the Activation Count becomes 1, and the task will be executed again at the first opportunity according to its priority.

Advantage:

- If a runnable is activated again soon after it already began executing, then it is entered again with minimum delay.

Disadvantage:

- If a runnable is activated multiple times before the entry condition test in the task body, then the runnable will correctly be executed once, but after the runnable is completed and the task has terminated the Activation Count in the OS will still be 1. So the Task will enter again at the first opportunity, even though there may be nothing to do. ("Empty Execution Problem")

Variant Case: `--deviate-prefer-no-empty-executions`

In the case where all the runnables in the Task have the same entry conditions, then an optimization can be made that eliminates the Empty Execution Problem.

If the option `--deviate-prefer-no-empty-executions=on` is supplied and all the runnables in the Task have the same entry conditions, then the optimization is made and the `OsTask's` Activation Limit must be 1. RTA-RTE will warn if the Os configuration does not conform.

Advantages:

- Tasks satisfying the conditions above will not execute when there is nothing to do (no Empty Execution Problem).
- RTA-RTE will avoid generating the Activation Flag(s) and supporting code for tasks satisfying the conditions above.

Disadvantage:

- A runnable activation occurring soon after the runnable is started may be lost.



This is because, now that the Activation Flag has been removed, runnable activation behaves exactly the same as Task Activation (documented above). If the runnable activation occurs during the Task execution, the corresponding task activation is lost (because Activation Count remains at 1 during the Task execution). The Task will not be activated until the RteEvent occurs again after Task termination.

#### Activation Limit and error hook

---

It is required to set the Activation Limit according to the messages written by RTA-RTE in order to preserve the documented activation semantics.

Regardless of the Activation Limit, a burst of RteEvents can exceed the limit. This is correct, designed behavior of RTA-RTE. When the Activation Limit is exceeded, RTA-RTE will ignore the E\_OS\_LIMIT return status from the OS ActivateTask API, and will manage the entry of runnables by means of generated flags.

- If you have configured ErrorHook in the OS then it may be executed as a result of RTE's attempts to activate tasks over their limit. Make sure that your ErrorHook handles E\_OS\_LIMIT, at least from ActivateTask.
- It is a mistake to increase the Activation Limit to avoid E\_OS\_LIMIT because this will make the Empty Execution Problem worse.

## 19 Inter-ECU Communication

---

Once software components are mapped to ECUs, and their runnables are mapped to tasks, you know which inter-Software Component communication is local to an ECU and which is remote.

For remote (inter-ECU) communication you need to map the data items communicated between software components (which will be data elements for sender-receiver communications or parameter values for client-server operations) into system signals communicated by the AUTOSAR COM and the sub-COM communication stack.

This chapter first describes how to set up the general framework for inter-ECU communications, and then describes how to configure data mappings for sender-receiver communication and for client-server operations.

### 19.1 System Communications

---

Inter-ECU sender-receiver or client-server communication requires the XML configuration for the RTE to include the entire network configuration for inter-ECU communication. This means you will need to configure:

- `<SYSTEM-SIGNAL>` and `<SYSTEM-SIGNAL-GROUP>` elements that hold data that is communicated between ECUs. These are abstract representations of AUTOSAR COM signals and signal groups in the AUTOSAR system description.
- `<I-SIGNAL-I-PDU>` elements which indicate how one or more signals may be packed into a single PDU for concurrent transmission.

There are different types of PDU corresponding to the different layers of the communication network, but only the I-PDU (Interaction Layer Protocol Data Unit) type is relevant to the RTE, as IPDUs are the units of data used by COM.

- A bus-specific frame element is used to transmit data between ECUs in a communication channel and is composed of one or more PDUs, which in turn are composed of individual signals.

Communications based on the CAN or LIN protocols a frame is limited to containing only a single PDU.

Figure 19.1 illustrates the RTE relevant elements of the mapping of an AUTOSAR sender-receiver signal to an IPdu.

#### 19.1.1 Signals and Signal Groups

---

A signal, or message, is the basic unit of communication between ECUs, usually carrying a single logical item of data, such as a value representing engine temperature. AUTOSAR distinguishes between three kinds of signal: system signals, interaction layer signals (i-signals) and COM signals, all of which may refer to the same data item. To understand inter-ECU communication within RTA-RTE, it is important to understand what is referred to by the different types of signal.

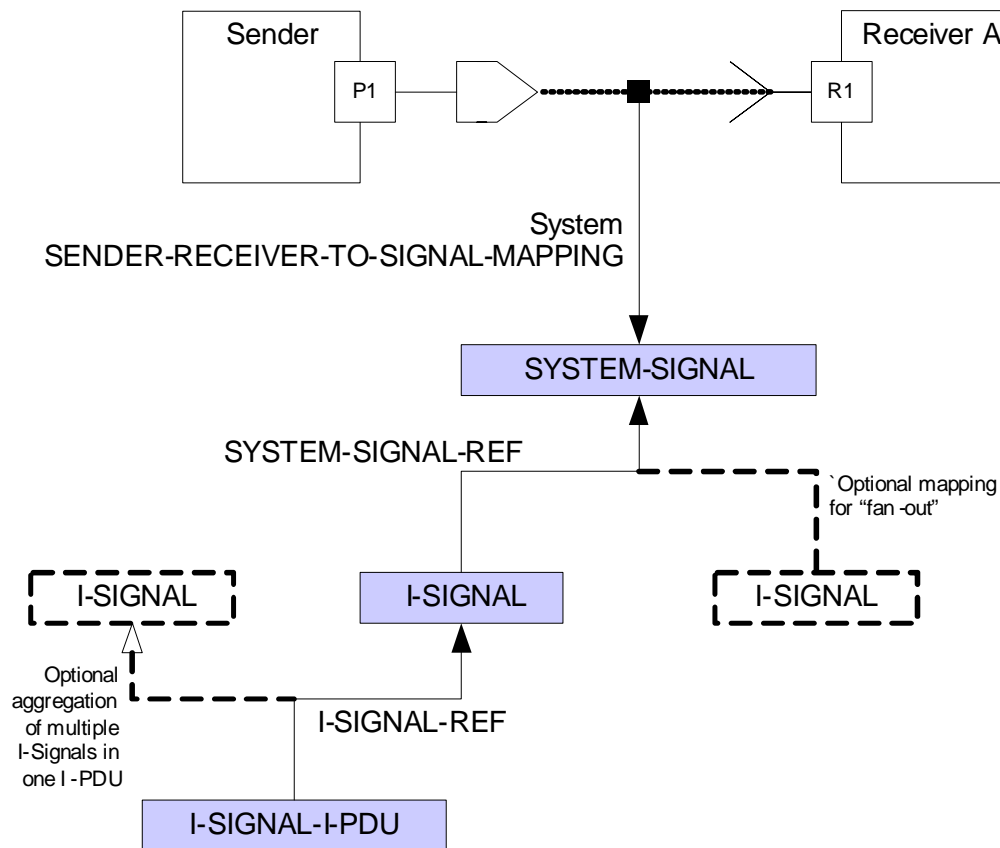


Figure 19.1: RTE-relevant Inter-ECU Mappings

**System Signal** —the most abstract representation of a message used to carry data, and is the kind of signal referred to when data mappings are configured (see below).

**I-Signal** —an instance of a system signal in a particular interaction layer PDU. A separate class is used in order to support “fan-out”, enabling a single system signal to be referred to in multiple PDU instances, sent to different destinations. Consequently, there may be  $n$  I-Signals corresponding to a single system signal.

**Com signal** —an element of the COM module configuration, representing a concrete instance of a signal. Com signals correspond directly to i-signals, i.e. for every i-signal in the system definition that is required for the ECU being generated, there should be a Com signal. The configuration of Com signals includes detailed information about the communication stack that is not included in the high-level system definition, such as repetition counts, filters, etc.

Each <SYSTEM-SIGNAL> represents an abstract block of data that is to be transmitted. It contains only the name by which the system signal can be referred. A <SYSTEM-SIGNAL> is therefore an abstract representation of a signal in AUTOSAR COM. However, the relationship between a <SYSTEM-SIGNAL> and a Com signal is not a sim-

ple one-to-one mapping, as system signal may be included in multiple frames each of which will require separate COM signals.

```
<SYSTEM-SIGNAL>
  <SHORT-NAME>Signal_gen_pa_rec_speed</SHORT-NAME>
</SYSTEM-SIGNAL>
```

An `<I-SIGNAL>` is an instance of a system signal in a particular interaction layer PDU. The `<I-SIGNAL>` refers to a system signal.

```
<I-SYSTEM>
  <SHORT-NAME>isig</SHORT-NAME>
  <SYSTEM-SIGNAL-REF>/signals/sig1</SYSTEM-SIGNAL-REF>
</I-SYSTEM>
```

Several signals can be collected into a **signal group** to ensure their simultaneous transmission. This technique is used to map complex data types into signals for communication. A signal group contains only a name and a list of references to basic system signals.

The following example maps multiple system signals into a signal group. Note that the packing of signals within the group into IPDUs is not specified as part of the signal group specification.

```
<SYSTEM-SIGNAL-GROUP>
  <SHORT-NAME>SignalGroup_gen_pa_rec</SHORT-NAME>
  <SYSTEM-SIGNALS-REFS>
    <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
      /signals/Signal_gen_pa_rec_speed
    </SYSTEM-SIGNAL-REF>
    <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
      /signals/Signal_gen_pa_rec_freq
    </SYSTEM-SIGNAL-REF>
    ...
  </SYSTEM-SIGNALS-REFS>
</SYSTEM-SIGNAL-GROUP>
```

An `<I-SIGNAL=GROUP>` is an instance of a system signal group in a particular interaction layer PDU. The `<I-SIGNAL-GROUP>` refers to a system signal group.

```
<I-SYSTEM-GROUP>
  <SHORT-NAME>isig</SHORT-NAME>
  <SYSTEM-SIGNAL-GROUP-REF DEST="SYSTEM-SIGNAL-GROUP">
    /signals/sigGroup1
  </SYSTEM-SIGNAL-GROUP-REF>
</I-SYSTEM-GROUP>
```

### 19.1.2 PDU Types

An `<I-SIGNAL-I-PDU>` element represents a collection of (system) signals for transfer between nodes in a communications network via references to I-Signals.

There are different types of PDU corresponding to the different layers of the communication network, but only the I-PDU (Interaction Layer Protocol Data Unit) type is relevant to the RTE, as IPDUs are the units of data used by COM. Specifically, only <I-SIGNAL-I-PDUs> are considered, as these carry application data between software components. (The terms “IPDU” hereafter refers to a I-Signal IPDUs).

As with other AUTOSAR elements, PDUs use a “type” and “prototype” method where an <I-SIGNAL-I-PDU> element defines a PDU structure and instances of the type are used as the <PDU-T0-FRAME-MAPPING> element.

An <I-SIGNAL-I-PDU> element defines the overall length (in bits) and contains one or more references to its included I-Signals and their mapped bit position.

```
<I-SIGNAL-I-PDU>
  <SHORT-NAME>IPdu_p2m_value</SHORT-NAME>
  <LENGTH>16</LENGTH>
  <SIGNAL-T0-PDU-MAPPINGS>
    <I-SIGNAL-T0-I-PDU-MAPPING>
      <SHORT-NAME>is2ipdu_p2m_value</SHORT-NAME>
      <I-SIGNAL-REF DEST='I-SIGNAL'>/T0optimizeRTE/isig_p2m_value</I-SIGNAL-REF>
      <PACKING-BYTE-ORDER>MOST-SIGNIFICANT-BYTE-LAST</PACKING-BYTE-ORDER>
      <START-POSITION>0</START-POSITION>
    </I-SIGNAL-T0-I-PDU-MAPPING>
    ...
  </SIGNAL-T0-PDU-MAPPINGS>
</I-SIGNAL-I-PDU>
```

The <LENGTH> element defines the size of the I-PDU in bits. Note that the length of the PDU can be greater than the sum of the included signal lengths.

The <I-SIGNAL-I-PDU> element is independent of the physical bus over which data may be transferred.

### 19.1.3 Communication Clusters

A communication describes a set of physical channels used to connect ECUs and hubs. An ECU belongs to a communication cluster if one of its communication ports is connected to one of the physical channels of the cluster. The content of a communication cluster depends on the type of bus used for a system; for example, it may just contain a description of a CAN bus.

The separate communication cluster types are used for different bus types, for example, a <CAN-CLUSTER> might define:

```
<CAN-CLUSTER>
  <SHORT-NAME>bob</SHORT-NAME>
  <CAN-CLUSTER-VARIANTS>
  <CAN-CLUSTER-CONDITIONAL>
  <PHYSICAL-CHANNELS>
  <CAN-PHYSICAL-CHANNEL>
    <SHORT-NAME>CHAN0</SHORT-NAME>
```

```
<FRAME-TRIGGERINGS>
<CAN-FRAME-TRIGGERING>
  <SHORT-NAME>trig0</SHORT-NAME>
  <FRAME-REF DEST="CAN-FRAME">/pkg/frame</FRAME-REF>
  <CAN-ADDRESSING-MODE>EXTENDED</CAN-ADDRESSING-MODE>
  <IDENTIFIER>456</IDENTIFIER>
</CAN-FRAME-TRIGGERING>
</FRAME-TRIGGERINGS>
</CAN-PHYSICAL-CHANNEL>
</PHYSICAL-CHANNELS>
<SPEED>500000</SPEED>
</CAN-CLUSTER-CONDITIONAL>
</CAN-CLUSTER-VARIANTS>
</CAN-CLUSTER>
```

RTA-RTE does not use information present in communication clusters.

#### 19.1.4 Frames

Each frame represents a communication protocol independent assembly of PDUs (and hence system signals) that will be transmitted across the network. A frame is therefore the AUTOSAR XML equivalent of an IPDU in AUTOSAR COM.

Each frame has a length and a set of <PDU-T0-FRAME-MAPPINGS>. Different bus types have different frame definitions, for example, a CAN frame might be defined as:

```
<CAN-FRAME>
  <SHORT-NAME>frame0</SHORT-NAME>
  <FRAME-LENGTH>4</FRAME-LENGTH>
  <PDU-T0-FRAME-MAPPINGS>
  <PDU-T0-FRAME-MAPPING>
    <SHORT-NAME>f1</SHORT-NAME>
    <PDU-REF DEST="I-SIGNAL-I-PDU">/signals/ipdu1</PDU-REF>
    <START-POSITION>0</START-POSITION>
  </PDU-T0-FRAME-MAPPING>
  </PDU-T0-FRAME-MAPPINGS>
</CAN-FRAME>
```

The <FRAME-LENGTH> specifies the length of the frame in bytes. The length can be of arbitrary size<sup>1</sup>, but you must be aware that the decision of which signals are located in which frames and their corresponding length will be influenced by the physical communication system you use between ECUs.

#### 19.1.5 COM OIL Generation

The RTE generator uses information from the system description to generate the COM API calls used to initiate communication and to respond to communication events indicated by COM.

For compatibility with previous AUTOSAR releases versions of the COM stack, the

<sup>1</sup>AUTOSAR currently restricts frame lengths to 8 bytes.

RTE generator can optionally also output a COM OIL configuration file suitable for a COM generator. This maps objects in the system description into the MESSAGES, NETWORKMESSAGES and IPDUs understood by COM.

**ETAS** *The generation of COM configuration is not part of the AUTOSAR RTE specification. RTA-RTE supports generation of a COM 1.0 OIL configuration.*

The generation of COM OIL in this release of RTA-RTE is limited to simple configurations featuring one-off, triggered transmission only, and does not take into account more complex features like cyclic timing or repetition counters. For more complex configurations it is necessary to configure the COM stack separately.

## 19.2 Inter-ECU Sender-Receiver Communication

For remote (inter-ECU) sender-receiver communication you need to map the data element prototypes transferred between software components into system signals communicated by the AUTOSAR COM and the sub-COM communication stack. This is done in the <DATA-MAPPINGS> section of the system mapping (see Section 18) element.

```
<SYSTEM>
  <SHORT-NAME>MySystem</SHORT-NAME>
  <MAPPINGS>
    <SYSTEM-MAPPING>
      <SHORT-NAME>mapping</SHORT-NAME>
      <DATA-MAPPINGS>
        <!-- Map data prototypes to signals -->
        ...
      </DATA-MAPPINGS>
    ...
  </MAPPING>
  <!-- Other SYSTEM elements -->
</SYSTEM>
```

Primitive data types are mapped to a single signal using a SenderReceiverToSignalMapping; complex data types are mapped to several signals in a signal group using a SenderReceiverToSignalGroupMapping.

Whether an AUTOSAR signal is primitive or complex depends on the Category of the ImplementationDataType used to represent the signal:

- VALUE — primitive. Use SenderReceiverToSignalMapping and see 19.2.1
- STRUCTURE — complex. Use SenderReceiverToSignalGroupMapping and see 19.2.2
- ARRAY — either. If the array is fixed-length and is small enough to fit in a single signal then it may be treated as primitive and mapped to a single signal. In any case it is permitted to treat an array as complex and map it to multiple, grouped signals.
- TYPE\_REFERENCE — effectively has the same category as the referenced type.

In a full system description, data mappings are described for provider (sender) ports only; the connectors in the software composition indicate which receiver ports are connected to these, which allows the mapping of received data elements to signals to be inferred also. In an ECU extract of a system description, in which only the software components relevant to a given ECU are included, it may be necessary to describe mappings for a requirer port. For example, the receiver side data mappings are needed if the corresponding provider port is not included in the ECU extract.

Both primitive and complex mapping element types follow the same basic pattern; an instance reference to a data element prototype that specifies the data to be sent inter-ECU and a mapping that specifies the system signal(s) used to carry the data.

### 19.2.1 Primitive Types

---

VariableDataPrototypes mapped to primitive ImplementationDataTypes are mapped using the SenderReceiverToSignalMapping element. One element must be configured for each AUTOSAR signal.

```
<SENDER-RECEIVER-TO-SIGNAL-MAPPING>
  <DATA-ELEMENT-IREF>
    <CONTEXT-COMPONENT-REF ... >
      /pkg/composition/producer
    </CONTEXT-COMPONENT-REF>
    <CONTEXT-PORT-REF ...>
      /pkg/swctype/pa
    </CONTEXT-PORT-REF>
    <TARGET-DATA-PROTOTYPE-REF ... >
      /pkg/interface/value
    </TARGET-DATA-PROTOTYPE-REF>
  </DATA-ELEMENT-IREF>
  <SYSTEM-SIGNAL-REF ... >
    /signals/Signal_gen_pa
  </SYSTEM-SIGNAL-REF>
</SENDER-RECEIVER-TO-SIGNAL-MAPPING>
```

The SenderReceiverToSignalMapping's <DATA-ELEMENT-IREF> specifies the component instance, port and data element from which the communication originates. If you use hierarchical composition, i.e. with delegation connectors, then you will need to reference each level in the hierarchy using multiple <COMPONENT-PROTOTYPE-REF> elements.

### 19.2.2 Complex Types

---

Complex types, which may be either records or arrays, are mapped using the SenderReceiverToSignalGroupMapping element. One element must be configured for each complex AUTOSAR signal. All signals for a single complex type are mapped to a signal group and therefore benefit from atomic transmission.

The SenderReceiverToSignalGroupMapping is similar to the mapping element for primitive types in that it contains a DATA-ELEMENT-IREF to select the AUTOSAR signal. However, unlike the primitive mapping the selection of system signals is more involved



since a recursive mapping of the record/array elements to system signals is required.

```
<SENDER-RECEIVER-TO-SIGNAL-GROUP-MAPPING>
  <DATA-ELEMENT-IREF>
    <CONTEXT-COMPONENT-REF ... >
      /pkg/composition/producer
    </CONTEXT-COMPONENT-REF>
    <CONTEXT-PORT-REF ...>
      /pkg/swctype/pa
    </CONTEXT-PORT-REF>
    <TARGET-DATA-PROTOTYPE-REF ... >
      /pkg/interface/value
    </TARGET-DATA-PROTOTYPE-REF>
  </DATA-ELEMENT-IREF>
  <SIGNAL-GROUP-REF ... >
    /signals/SignalGroup
  </SIGNAL-GROUP-REF>
  <TYPE-MAPPING>
    ...
  </TYPE-MAPPING>
</SENDER-RECEIVER-TO-SIGNAL-GROUP-MAPPING>
```

As with the mapping for primitive types, the <DATA-ELEMENT-IREF> specifies the component instance, port and data element from which the communication originates. If you use hierarchical composition, i.e. with delegation connectors, then you will need to reference each level in the hierarchy using multiple <CONTEXT-COMPONENT-REF> elements.

A group mapping includes the <TYPE-MAPPING> element that is the root for the recursive mapping of system signals within the signal group referenced by <SIGNAL-GROUP-REF>. The example will map the following C structure R1 to signals where R1 is defined as:

```
struct R1 {
    UInt8    Speed;
    UInt8    Freq;
};
```

As explained above, the mapping of a complex type can be recursive since complex data types can themselves be defined recursively; for example, a structure R2 could be defined in terms of structure R1 as well as additional primitive data elements. The recursive mapping can produce very large, very complex, mappings and therefore to ease understanding the type mapping for R1 uses a flat definition only.

```
<TYPE-MAPPING>
  <SENDER-REC-RECORD-TYPE-MAPPING>
    <RECORD-ELEMENT-MAPPINGS>
      <!-- Mapping for member Speed' ' -->
      <!-- Mapping for member Freq' ' -->
    </RECORD-ELEMENT-MAPPINGS>
  </SENDER-REC-RECORD-TYPE-MAPPING>
```

```
</TYPE-MAPPING>
```

First, the <RECORD-ELEMENT-MAPPINGS> element must define the mapping for structure member "Speed" which has type UInt8:

```
<SENDER-REC-RECORD-TYPE-MAPPING>
  <RECORD-ELEMENT-MAPPINGS>
    <!-- Mapping for member "Speed" (R1.Freq) -->
    <SENDER-REC-RECORD-ELEMENT-MAPPING>
      <RECORD-ELEMENT-REF>/types/R1/Speed</RECORD-ELEMENT-REF>
      <SYSTEM-SIGNAL-REF>/pkg/sig_a</SYSTEM-SIGNAL-REF>
    </SENDER-REC-RECORD-ELEMENT-MAPPING>
    <!-- Mapping for member "Freq" (R1.Freq) -->
    <SENDER-REC-RECORD-ELEMENT-MAPPING>
      <RECORD-ELEMENT-REF>/types/R1/Freq</RECORD-ELEMENT-REF>
      <SYSTEM-SIGNAL-REF>/pkg/sig_b</SYSTEM-SIGNAL-REF>
    </SENDER-REC-RECORD-ELEMENT-MAPPING>
  </RECORD-ELEMENT-MAPPINGS>
</SENDER-REC-RECORD-TYPE-MAPPING>
```

The structure R1 uses only simple types and therefore contains a single mapping level. If it contained a nested structure then the definition would use nested <SENDER-REC-RECORD-ELEMENT-MAPPING> elements.

The example above illustrates the mapping of a record type; arrays are mapped similarly, using the <SENDER-REC-ARRAY-TYPE-MAPPING> element within a complex type mapping.

### 19.3 Inter-ECU Client-Server Communication

For remote (inter-ECU) client-server communication you need to map the operation argument prototypes passed between the client and server software components into system signals communicated by the AUTOSAR COM and the sub-COM communication stack. As with sender-receiver communication, this is done in the DataMapping section of the system's Mapping element (see Section 19.2).

Unlike the data mappings for sender-receiver communications, which may use individual signals for primitive data elements or signal groups for complex elements, client-server data mappings always require a signal group. This is because all the parameters of an operation are transmitted in an atomic transfer operation. A mapping for one operation, including both the parameters and other meta-data, is encapsulated within a <CLIENT-SERVER-T0-SIGNAL-GROUP-MAPPING> element. This will contain some or all of the following sub-elements:

- A <MAPPED-OPERATION-IREF> which identifies the software component instance, port and operation defining this client-server communication. If you use hierarchical composition, i.e. with delegation connectors, then you will need to reference each level in the hierarchy using multiple <CONTEXT-COMPONENT-REF> elements.
- Either a <REQUEST-GROUP-REF> or a <RESPONSE-GROUP-REF>, which identifies the

system signal group to be used to collect the signals for the request (call) or response of the operation.

- A set of <PRIMITIVE-TYPE-MAPPINGS> which provide mappings for any arguments with simple types. These are described in more detail below.
- A set of <COMPOSITE-TYPE-MAPPINGS> which provide mappings for any arguments with complex types (records or arrays). These are described in more detail below.
- An optional <APPLICATION-ERROR> element which identifies the system signal used to carry an application-defined error code returned from the server.
- An optional <EMPTY-SIGNAL> used for operations with no parameters. This is necessary to avoid the transfer of signal groups which contain no signals.
- An optional <SEQUENCE-COUNTER> mapping which identifies a signal used to pass a counter for tracking request-to-response pairings.

For each inter-ECU client-server operation you must specify a pair of mappings, using separate <CLIENT-SERVER-TO-SIGNAL-GROUP-MAPPING> elements. One of these maps arguments of the calling side of the operation, the other maps those of the response. The presence of either a <REQUEST-GROUP-REF> or a <RESPONSE-GROUP-REF> indicates the direction (client-to-server or server-to-client) of a particular signal group mapping.

<CLIENT-SERVER-TO-SIGNAL-GROUP-MAPPING>

```
<APPLICATION-ERROR>
  <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
    /system/signal_srv_reply1_appError
  </SYSTEM-SIGNAL-REF>
</APPLICATION-ERROR>

<MAPPED-OPERATION-IREF>
  <CONTEXT-COMPONENT-REF DEST="... ">
    /composition/Test552/client
  </CONTEXT-COMPONENT-REF>
  <CONTEXT-PORT-REF DEST="R-PORT-PROTOTYPE">
    /client/cswc/ra
  </CONTEXT-PORT-REF>
  <TARGET-OPERATION-REF DEST="OPERATION-PROTOTYPE">
    /interfaces/simple_if1/op_async
  </TARGET-OPERATION-REF>
</MAPPED-OPERATION-IREF>

<PRIMITIVE-TYPE-MAPPINGS>
  ...
</PRIMITIVE-TYPE-MAPPINGS>

<RESPONSE-GROUP-REF DEST="SYSTEM-SIGNAL-GROUP">
  /system/signal_srv_reply1_group
</RESPONSE-GROUP-REF>
```

```
<SEQUENCE-COUNTER>
  <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
    /system/signal_srv_reply1_seqCount
  </SYSTEM-SIGNAL-REF>
</SEQUENCE-COUNTER>

</CLIENT-SERVER-TO-SIGNAL-GROUP-MAPPING>
```

The element used to map argument prototypes to system signals depends on whether the communicated argument's data type is primitive or complex. These mappings are described in the following sections.

### 19.3.1 Primitive Argument Types

Primitive types are mapped using the <CLIENT-SERVER-PRIMITIVE-TYPE-MAPPING> element. One element must be configured for each operation argument. If there is more than one primitive argument, their mappings should be grouped together inside a <PRIMITIVE-TYPE-MAPPINGS> element.

```
<PRIMITIVE-TYPE-MAPPINGS>
  <CLIENT-SERVER-PRIMITIVE-TYPE-MAPPING>
    <ARGUMENT-REF DEST="ARGUMENT-PROTOTYPE">
      /interfaces/simple_if1/op_sync/a
    </ARGUMENT-REF>
    <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
      /system/signal_cli_request1_pA
    </SYSTEM-SIGNAL-REF>
  </CLIENT-SERVER-PRIMITIVE-TYPE-MAPPING>
</PRIMITIVE-TYPE-MAPPINGS>
```

The 'client-server primitive type mapping <ARGUMENT-REF> specifies the argument prototype from which the communication originates (and which implicitly lies within the referenced operation).

### 19.3.2 Complex Argument Types



*This release of RTA-RTE does not support interECU transmission of complex types using client-server communication.*

Complex types, which may be either records or arrays, are mapped within a <COMPOSITE-TYPE-MAPPINGS> block, using a set of XML structures very similar to those used for sender-receiver mappings. One element must be configured for each complex argument prototype. The only significant difference is that in the client-server case no signal group is required for a complex mapping, as all the signals associated with an operation are always mapped to a dedicated signal group for either the request or response.

The <CLIENT-SERVER-RECORD-ELEMENT-MAPPING> is similar to the mapping element for primitive types in that it contains an <ARGUMENT-PROTOTYPE-REF> to select the argument. However, unlike the primitive mapping, the selection of system signals is more

involved since a recursive mapping of the record/array elements to system signals is required.

```

<COMPOSITE-TYPE-MAPPINGS>
  <CLIENT-SERVER-RECORD-TYPE-MAPPING>
    <RECORD-ELEMENT-MAPPINGS>

      <!-- Element 1: Speed -->
      <CLIENT-SERVER-RECORD-ELEMENT-MAPPING>
        <RECORD-ELEMENT-IREF>
          <COMPONENT-PROTOTYPE-REF DEST="COMPONENT-PROTOTYPE">
            /composition/Test705/Client
          </COMPONENT-PROTOTYPE-REF>
          <PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">
            /client/swClient/ra
          </PORT-PROTOTYPE-REF>
          <ARGUMENT-PROTOTYPE-REF DEST="ARGUMENT-PROTOTYPE">
            /interface/client_server/op_sync/b
          </ARGUMENT-PROTOTYPE-REF>
          <RECORD-ELEMENT-REF DEST="RECORD-ELEMENT">
            /test_types/R2/Speed
          </RECORD-ELEMENT-REF>
        </RECORD-ELEMENT-IREF>
        <SIGNAL-REF DEST="SYSTEM-SIGNAL">
          /system/signal_cli2bnc_B_Speed
        </SIGNAL-REF>
      </CLIENT-SERVER-RECORD-ELEMENT-MAPPING>

      <!-- Element 2: Freq -->
      <CLIENT-SERVER-RECORD-ELEMENT-MAPPING>
        <RECORD-ELEMENT-IREF>
          <COMPONENT-PROTOTYPE-REF DEST="COMPONENT-PROTOTYPE">
            /composition/Test705/Client
          </COMPONENT-PROTOTYPE-REF>
          <PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">
            /client/swClient/ra
          </PORT-PROTOTYPE-REF>
          <ARGUMENT-PROTOTYPE-REF DEST="ARGUMENT-PROTOTYPE">
            /interface/client_server/op_sync/b
          </ARGUMENT-PROTOTYPE-REF>
          <RECORD-ELEMENT-REF DEST="RECORD-ELEMENT">
            /test_types/R2/Freq
          </RECORD-ELEMENT-REF>
        </RECORD-ELEMENT-IREF>
        <SIGNAL-REF DEST="SYSTEM-SIGNAL">
          /system/signal_cli2bnc_B_Freq
        </SIGNAL-REF>
      </CLIENT-SERVER-RECORD-ELEMENT-MAPPING>

    </RECORD-ELEMENT-MAPPINGS>
  </CLIENT-SERVER-RECORD-TYPE-MAPPING>
</COMPOSITE-TYPE-MAPPINGS>

```

The mapping of an array argument prototype is similar to the record type shown here, but uses the <CLIENT-SERVER-ARRAY-ELEMENT-MAPPING> element. As with inter-ECU sender-receiver communications, nested types can also be mapped, again with the exception that arrays of arrays are currently not supported.

## 20 Using the OS and COM Configurations

The OS and COM modules for a particular ECU must provide the RTE for that ECU with all the services it requires in order to fulfil the requirements of the software components and BSW modules on the ECU. For instance the RTE may require particular OsResources for serialization of access to application / BSW module code and/or internal state and may require particular COM signals for communication with software components on other ECUs. The configurations of the OS and COM modules must therefore eventually contain all the required configuration objects for the RTE, in addition to configuration for the ECU environment (e.g. OS target) and Os objects for other BSW modules.

As mentioned in section 18.4 the RTE generator in RTE phase uses part of the OS configuration to determine the execution contexts of runnables, the concurrency model for the system and the task bodies it must generate. It is possible to supply just this part of the OS configuration at RTE generation time and subsequently add the other required configuration before the OS generation or configuration step. Similarly it is possible to supply just part of the COM configuration at RTE generation time and add the rest before generating or configuring COM.

The RTE generator optionally outputs two configuration files that can be useful when taking this approach:

1. An Operating System configuration file containing both additional configuration for Os objects that were in the input configuration but need particular properties for the RTE to function correctly and additional Os objects required for the scheduling and serialization of the Os tasks by the RTE. Depending on the selected OS (RTE option `--operating-system`) this could be an AUTOSAR XML file or a legacy OIL file. The default name of the file depends on the OS used; for example, for the AUTOSAR R4.0 OS plug-in it is `osNeeds.arxml`, for the AUTOSAR R1.0 plug-in it is `rta-osek.oil`, for the OSEK v2.2.3 plug-in it is `osek.oil`. The name can be changed using the command-line option `--os-file`.
2. A COM v1.0 configuration file called `rta-com.oil`.

These configuration files are not complete; they must be combined with other configuration files, not generated by RTA-RTE, before being passed to the operating system and communication service configuration tools.



*These files contain partial configuration information for the respective modules. The scope of the generated configuration data are those parts of configuration necessary for the RTE to work, namely Os tasks and supporting Os objects and COM signals and signal groups. If additional OS/COM configuration is provided as input to the RTE generator then this is ignored.*



*The output of OS and COM configuration can be ignored if not required or not suitable. The plug-ins should not be removed from the the RTE generator configuration file.*

It is possible and sometimes necessary to supply more than the minimum OS configuration on the input to the RTE generator. In particular, in a partitioned system it is necessary to supply the OsApplications and the mapping of tasks to them; in this case the additional OS configuration for the RTE that is generated by the RTE generator does not place the additional Os objects in OsApplications and cannot be used directly. For R4.0 it is possible to disable the generation of additional OS configuration and have the RTE generator check that the supplied configuration is sufficient, using the `--strict-config-check` option.



*When the OS and/or COM configuration output by the RTE generator is ignored (and the strict configuration check is not enabled) it is important that all the Os and COM objects that the generated RTE uses are configured with the expected names, at latest by the OS and COM generation or configuration step, otherwise compilation or link errors will result. It is also important to ensure that all Os tasks and ISRs in which RTE code may execute are configured as accessing the necessary Os objects used by the RTE, otherwise runtime errors may occur. Details of the Os objects typically required by the RTE may be found in the OS Configuration section of the RTA-RTE Reference Manual.*

## 20.1 Operating System Configuration

All AUTOSAR application software components' access to the OS occurs via the RTE. The RTE insulates application software components from the need to access OS API calls directly and hence RTA-RTE provides no mechanism for them to do so.

### 20.1.1 Scheduling Policy

By default, all runnable entities are scheduled pre-emptively based on priorities specified during configuration.

Runnable entities that must guard against access to an exclusive area at runtime can use the RTE concurrency control API to force non-preemption by declaring an exclusive area around relevant code regions. All runnable entities in a prototype of a component that specify the same explicit exclusive area are scheduled non pre-emptively.

### 20.1.2 OS Counters

The generated RTE uses OS objects, either periodic alarms or schedule tables, to trigger runnable entities activated by timing events. The alarm/table is driven by a counter `Rte_Tick_Counter`. The rate at which the counter must be driven depends on configured timing events – for convenience the output file `Rte_Const.h` contains the required period (in microseconds) as `RTE_PERIODIC_COUNTER_TICK_INTERVAL_US`.

The generated RTE uses OS alarms for sporadic events such as timeouts. If such alarms are required then the counter `Rte_TOut_Counter` is created in the OS configuration. To avoid unacceptable jitter on timeouts the `Rte_TOut_Counter` is always ticked at 1ms. For convenience the output file `Rte_Const.h` contains the required period (in microseconds) as `RTE_ALARM_COUNTER_TICK_INTERVAL_US`.



Because the AUTOSAR OS and OSEK OS forbid SetEvent and ActivateTask API calls within an alarm callback, it is necessary not to directly tick Rte\_Tout\_Counter but instead to call the generated function Rte\_Tick\_Timeouts. The Rte\_Tick\_Timeouts API ticks Rte\_Tout\_Counter and then performs any RTE state maintenance needed if timeouts occurred.

### 20.1.3 Scheduling Periodic Runnable Entities

The RTE generator uses the periodicity information from the RTE configuration files to generate one or more schedule tables to execute periodic runnable entities (those triggered by TimingEvent). The schedule table specifies when each runnable entity is executed within the table hyper-period (the point at which the sequence of runnable entities on the table repeats).

As an example of how the RTE generator builds schedule tables, consider two runnable entities, *A* and *B*, with periods of 3ms and 4ms respectively. The periodic schedule generated by the RTE generator has a hyper-period of 12ms and would invoke the runnable entities as follows:

Time	Action
0ms	Invoke <i>A</i> , invoke <i>B</i>
3ms	Invoke <i>A</i>
4ms	Invoke <i>B</i>
6ms	Invoke <i>A</i>
8ms	Invoke <i>B</i>
9ms	Invoke <i>A</i>
12ms	Repeat

The schedule may result in the execution of one or more tasks depending on the jitter requirements and category of runnable entities contained in the schedule. Typically only a single task is required since Category 1 runnable entities cannot block and must have finite execution time. However, multiple tasks may be required to meet the jitter requirements.

For example consider two runnable entities, each with an execution time of 7ms, executed at 5ms and 10ms offsets into a schedule. If each runnable entity has an acceptable start jitter of 5ms then a single task can be used to invoke the runnable entities—the first will start at 5ms, the second at 12ms. However if the acceptable start jitter is reduced to 1ms then two tasks are now required since the first will now execute at 5ms and the second task will execute at 10ms and hence the jitter requirements are met even though the response time of the first runnable entity is extended.

### 20.1.4 OsResources

The generated RTE uses an OsResource for the serialization of access to internal state. In a multi-core system a different OsResource is used for each core. OsResources can

also be used to serialize access to application / BSW module code, depending on the configured protection requirements and the possible concurrency of runnables implied by the task mappings and task configuration.

Any task or ISR in whose context RTE code may run must be marked as locking the RTE-internal OsResource, or the RTE-internal OsResource for the core on which it executes in a multi-core system. This includes tasks whose bodies are generated by the RTE generator, i.e. that have runnable entities and/or schedulable entities mapped to them, and tasks and ISRs from which BSW modules may call RTE generated notification functions, such as COM reception notifications for example.

Similarly, tasks and ISRs in whose context protected application / BSW module code executes must be marked as locking the corresponding OsResources used by the RTE to provide the protection.

See the *OS Configuration* section of the *RTA-RTE Reference Manual* for details.

#### 20.1.5 Using the OS Configuration

---

The OS configuration generated by the RTE generator contains only the task configuration for the tasks you defined at system deployment time, the schedule tables for runnable entities triggered with periodic TimingEvents and OsResources needed for serializing access to application code or internal state. You will need to include the generated XML / OIL file with the other XML / OIL files that define other aspects of your system, for example:

- message/task mappings generated by the COM configuration process
- a driver interrupt for the schedule table
- other interrupt handlers
- other tasks for non-RTE managed ECU functionality
- additional counters, alarms etc.

For OIL configuration files you can use the `#include` mechanisms to integrate the RTE OS configuration with your other OS configuration files. You should consult your OS documentation for further information or for details of how to merge XML configuration files.

## 20.2 Communication Stack Configuration

---

When component prototypes that communicate are allocated to different ECU instances, the RTE generator will generate an OIL configuration file for the communication stack.

### 20.2.1 Using the COM Configuration

---

The `rta-com.oil` configuration file generated by the RTE generator contains the messages and network messages (messages packed into COM IPDUs) and the IPDU frame

types. The configuration also contains the declaration of which OS tasks access the specified messages.

You will need to include the `rta-com.oil` file with any additional configuration you might have. The following example shows the general structure:

```
    CPU MyECU {
        OS dummyOS {
        };

        COM Somename {
            COMTIMEBASE = 0.001;
            COMAPPMODE = "COMAppMode1";
            COMSTATUS = COMEXTENDED;
        };

        #include "rta-com.oil"
    };
```

The COM configuration should be processed with the COM configuration tool to generate the data structures for the communication layer. Please consult your COM documentation for details.



*The configuration of the COM layer does not include configuration of the network drivers (e.g. CAN drivers, FlexRay drivers etc.) or the PDU Router. This must be done independently using your AUTOSAR XML files and network driver configuration tools.*

## 21 Debugging Implementations with VFB Tracing

---

Virtual Function Bus (VFB) Tracing permits the monitoring of AUTOSAR signals as they are sent and received across the VFB.

VFB tracing is implemented as a series of hook functions that are invoked automatically by the generated RTE when interesting events within the RTE occur. Each hook function corresponds to a single trace event, for example task dispatch or runnable entity invocation.

The trace events supported by RTA-RTE are defined in Section 21.2. RTA-RTE provides a means by which VFB tracing can either be globally enabled or disabled (if disabled, VFB tracing has no run-time effect) as well as the enabling or disabling of individual trace events. Configuration of VFB tracing for RTA-RTE is described in Section 21.3.

### 21.1 Enabling Tracing

---

The creation of VFB Tracing hook calls within the generated code is controlled by either a command-line option or an RTE configuration parameter within the ECUC.

If VFB Tracing is disabled then no hook calls are created in the generated code.



*Even if hook calls are generated they will have no runtime effect unless definitions in `Rte_Cfg.h` are in place to globally enable VFB tracing to enable the required individual hook calls.*

### 21.2 Supported Trace Events

---

RTA-RTE supports the following VFB Trace events:

- RTE API call start and return.
- Inter-ECU signal transmission, reception and reception notification.
- Runnable entity invocation and termination.
- OS Event set, wait and return.
- OS Task activation and dispatch.

#### 21.2.1 RTE API Start

---

RTE API Start is invoked by the RTA-RTE when an API call is made by a software component. The hook function must have the name:

```
Rte_<api>Hook_<swc>_<ap>_Start
```

Where <api> is the API root name (Write, Call, Feedback, etc.), <swc> the software-component type name and <ap> the access point (combination of port name and data item name/operation name separated by an underscore).

### 21.2.2 RTE API Return

---

RTE API Return is invoked by the RTA-RTE just before an API call returns control to a component. The hook function must have the name:

```
Rte_<api>Hook_<swc>_<ap>_Return
```

Where <api> is the API root name (Write, Call, Feedback, etc.), <swc> the software-component type name and <ap> the access point (combination of port name and data item name/operation name separated by an underscore).

### 21.2.3 COM Signal Transmission

---

A trace event indicating a transmission request of an Inter-ECU signal or signal group by the RTE. The hook function is invoked by the RTA-RTE just before Com\_SendSignal or Com\_SendSignalGroup is invoked. The hook function must have the name:

```
Rte_ComHook_<signal>_SigTx
```

Where <signal> is the system signal name.

### 21.2.4 COM Signal Reception

---

A trace event indicating an attempt to read Inter-ECU signal by the RTE. The hook function is invoked by the RTA-RTE after successful return from Com\_ReceiveSignal. The hook function must have the name:

```
Rte_ComHook_<signal>_SigRx
```

Where <signal> is the system signal name.

### 21.2.5 COM Notification

---

A trace event indicating the start of a COM notification. Invoked by RTA-RTE generated code on entry to the COM call-back. The hook function must have the name:

```
Rte_ComHook_<signal>
```

Where <signal> is the system signal name.

### 21.2.6 OS Task Activation

---

A trace event invoked by RTA-RTE generated code immediately before activating the specified task. The hook function must have the name:

```
Rte_Task_Activate
```

The Rte\_Task\_Activate hook is passed the task handle of the activated task.

### 21.2.7 OS Task Dispatch

---

A trace event invoked by the RTA-RTE immediately on dispatch of the specified generated task (provided it contains runnable entities). The hook function must have the name:

## Rte\_Task\_Dispatch

The Rte\_Task\_Dispatch hook is passed the task handle of the activated task.

### 21.2.8 OS Task Set Event

---

A trace event invoked immediately before generated RTA-RTE code attempts to set an OS Event. The hook function must have the name:

Rte\_Task\_SetEvent

The parameters of the hook call define the task for which the event is being set and the event mask.

### 21.2.9 OS Task Wait Event

---

A trace event invoked immediately before generated RTA-RTE code attempts to wait for an OS Event. The hook function must have the name:

Rte\_Task\_WaitEvent

The parameters of the hook call define the task waiting for the event and the event mask.

### 21.2.10 OS Task Wait Event Return

---

A trace event invoked immediately before generated RTE code returns from waiting for an OS Event. The hook function must have the name:

Rte\_Task\_WaitEventRet

The parameters of the hook call define the task returning from the wait and the event mask.

### 21.2.11 Runnable Entity Invocation

---

A trace event invoked by the RTE just before execution of a 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. The hook function must have the name:

Rte\_Runnable\_<swc>\_<reName>\_Start

Where <swc> is the software-component type name and <reName> the runnable entity name.

### 21.2.12 Runnable Entity Termination

---

A trace event invoked by the RTE immediately after execution returns to RTE code from a runnable entity. This trace event occurs before write-back of data elements is made to support the Rte\_IWrite API Call. The hook function must have the name:

Rte\_Runnable\_<swc>\_<reName>\_Return

Where <swc> is the software-component type name and <reName> the runnable entity name.

## 21.3 Configuration

---

When generation of VFB trace hooks is enabled, RTA-RTE always places hook calls in the generated RTE. However unless both VFB Tracing is globally enabled and the individual trace event(s) enabled the generated hooks will have no run-time effect.

The RTE configuration header file, Rte\_Cfg.h, contains user definitions that globally enable or disable VFB tracing and enable individual trace events.

### 21.3.1 Enabling VFB Tracing

---

VFB Tracing is globally enabled when the RTE configuration header file defines RTE\_VFB\_TRACE as a non-zero integer, for example:

```
#define RTE_VFB_TRACE (1)
```

If RTE\_VFB\_TRACE is not defined in the RTE configuration header file VFB Tracing is globally disabled.

### 21.3.2 Enabling VFB Trace Events

---

Using definitions in the RTE configuration header file it is possible to enable or disable individual trace events, for example, it is possible to enable just “Task dispatch” events and to ignore all others.



*Individually enabled trace events have no effect if VFB tracing is globally disabled.*

An individual trace event is enabled when there is a #define in the RTE configuration header file for the hook function name. The following example enables VFB Tracing for “Task Dispatch” events

```
#define Rte_Task_Dispatch (1)
```

The name of the #define must be the same as the hook name.

## 21.4 Trace Event Header File

---

RTA-RTE creates a header file, Rte\_Hook.h, that defines the signature of each hook function and is responsible for disabling the trace event so that its hook call has no run-time effect if the hook function name is not defined in the RTE configuration header file.

## 21.5 Implementing Hook Functions

---

For each enabled trace event RTA-RTE will invoke a hook function. The implementation of the function is not provided by RTA-RTE but instead must be supplied by the user. The implementation is free to perform any actions required within the hook function, e.g. log the invocation.

The `Rte_Hook.h` header file should be included in any user code that implements hook functions. Additional header files, such as `Rte_Type.h`, may be necessary depending on the enabled trace events and the data types used within the program.

The following example implements a hook function for “Task Dispatch”:

```
#include <Os.h>
#include "Rte.h"
#include "Rte_Hook.h"

FUNC(void, RTE_APPL_CODE)
Rte_Task_Dispatch(TaskType t)
{
    if ( t == taskA )
    {
        /* Log dispatch of taskA */
    }
}
```



## 22 Data Transformation

---

RTA-RTE supports data transformation, introduced in AUTOSAR 4.2, for sender/receiver communications. When data transformation is requested for a signal, RTA-RTE will transform any data items that are sent or received via the `Rte_Write` and `Rte_Read` APIs associated with that signal.

On the transmission side of a communication, the generated `Rte_Write` API calls the configured sequence of `BswModuleEntrys` to convert the port data to a byte array (array of `uint8`) and passes the resulting array to `COM_SendSignal` for transmission.

On the reception side of a communication, the data is received from COM as a byte array (array of `uint8`), and the generated `Rte_Read` API calls the configured sequence of `BswModuleEntrys` to convert the byte array back into port data.

### 22.1 Extent of support

---

Data transformation in RTA-RTE is based on the specification in AUTOSAR version 4.2.2. The scenarios supported are described in Section [22.1.1](#).

RTA-RTE does not support the complete AUTOSAR specification for data transformation. Section [22.1.2](#) outlines the limitations in the support for data transformation in the current implementation.

There are a number of differences between the implementation of data transformation in RTA-RTE and the published AUTOSAR 4.2.2 specification. These differences are highlighted in Section [22.1.3](#).

#### 22.1.1 Supported scenarios

---

The following are supported by RTA-RTE:

**Configurations supplied as an ECU\_EXTRACT:** RTA-RTE supports data transformation for configurations containing an AUTOSAR System of category `ECU_EXTRACT`, where signals are mapped to delegation ports on the `RootSwComposition`.

**Explicit unqueued sender/receiver communications:** RTA-RTE supports data transformation for explicit sender/receiver communications using the `Rte_Read` and `Rte_Write` APIs.

**Serialization with SOME/IP plus optional end-to-end protection:** RTA-RTE supports serialization of the input data using the `SOME/IP` transformer, optionally followed by end-to-end protection using the `E2E` transformer.

**Out-of-place buffering:** RTA-RTE supports the use of the `E2E` transformer when it is configured for out-of-place buffering.

**Reporting of transformer errors to calling SWC:** RTA-RTE supports extending the `Rte_Read` and `Rte_Write` APIs with the optional `transformerError` parameter to provide transformer error information to the calling SWC.

### 22.1.2 Limitations

---

RTA-RTE does not currently support data transformation for these configurations:

- Transformer chains (DataTransformations) of length 3 or more.
- Transformer chains containing TransformationTechnologys of SECURITY or CUSTOM classes.
- Transformers using in-place buffering (where BufferProperties.inPlace is TRUE)
- Transformers requiring access to the original data (where the needsOriginalData attribute is TRUE)
- Transformer chains that use the COM-based transformer for serialization

RTA-RTE does not currently support data transformation for these types of communication:

- Implicit sender/receiver communication (Rte\_IWrite and Rte\_IRead APIs)
- Queued sender/receiver communication (Rte\_Send and Rte\_Receive APIs)
- Client/server communication (Rte\_Call and Rte\_Result APIs)
- External trigger communication (Rte\_Trigger API)
- Any form of intra-ECU communication

RTA-RTE does not currently support transformation in the following APIs used for un-queued explicit sender/receiver communication:

- Rte\_Invalidate
- Rte\_DRead
- Rte\_Feedback

RTA-RTE does not support the use of data transformation for any inter-ECU communication where the same data element is sent or received via multiple signals. This includes configurations where multiple SystemSignals or SystemSignalGroups are mapped to the same data element, as well as signal fan-in and fan-out as a result of multiple ISignals referencing the same SystemSignal.

### 22.1.3 Deviations from AUTOSAR

---

The implementation of data transformation in RTA-RTE deviates from the AUTOSAR specification in the following ways:

## Buffer Size

---

To eliminate problems caused by different interpretations of the AUTOSAR standard, and to minimize the cost of corrections to the configuration, RTA-RTE does not attempt to calculate the buffer size from the input type size and the `bufferProperties.BufferComputation`.

Instead, RTA-RTE requires that an `XfrmBufferSizeInBytes` be supplied for each `XfrmImplementationMapping`. This specifies the size of the buffer for the transformed data, i.e. the output buffer on the sending side and the input buffer on the receiving side. The specified buffer size must include an allowance for any header that the transformer function may add.

`XfrmBufferSizeInBytes` is a mandatory integer parameter that must be between 1 and 65535, since the C-type of `bufferLength` in the calling protocol is `uint16`. If any `XfrmImplementationMapping` does not contain a parameter `XfrmBufferSizeInBytes`, RTA-RTE raises error 2968.

## Prototypes for transformer functions

---

When a transformer function is called, RTA-RTE forward declares the function by writing the function prototype to `Rte.c`. The declaration is generated without the use of compiler abstraction macros (`FUNC`, `CONSTP2VAR` etc.).

Version 4.2.2 of the AUTOSAR specification is ambiguous as to how transformer functions should be declared. According to SWS\_Rte 07283, the declaration should be generated in the Module Interlink Header File (MIHF), however there is no requirement to include MIHFs in `Rte.c`, nor any mechanism to know what MIHFs might be appropriate for a given transformer (given only a reference to a `BswModuleEntry`, RTA-RTE does not know which BSW Module's ExecutableEntity it belongs to).



*RTA-RTE does not currently implement RfC 73044 (adopted in AUTOSAR versions 4.3.0 and later), which adds a new attribute to `bswModuleEntries` to determine whether or not RTE is responsible for generating the function prototype in the MIHF.*

### 22.1.4 Unsupported feature combinations

---

When data transformation is in use, the use of these features is not supported:

- Data conversion
- Signal fan-in and fan-out
- MEMORY optimization strategy

The use of these features combined with data transformation may result in RTA-RTE exiting with an error, or other unexpected behaviour.

## 22.2 Enabling data transformation in an input configuration

---

To enable data transformation, the input configuration must contain the following elements:

- An AUTOSAR system of category ECU\_EXTRACT with signals mapped via delegation ports (section [22.2.1](#))
- System model elements associated with transformation (section [22.2.2](#))
- Mappings to transformer implementation functions (section [22.2.3](#))
- BSW entries corresponding to transformer functions (section [22.2.4](#))
- COM signals to send the transformed data (section [22.2.5](#))

### 22.2.1 System category and signal mapping

---

RTA-RTE only supports data transformation for AUTOSAR Systems that are of category ECU\_EXTRACT. The use of any System category other than ECU\_EXTRACT whenever `XfrmImplementationMappings` are present in the input configuration (see section [22.2.3](#)) will cause RTA-RTE to exit with an error.

The signals that transport transformed data must be mapped to delegation ports on the `RootSwComposition`. RTA-RTE will exit with an error if transformation is enabled for a signal that is mapped directly to an inner port on a SWC contained within the `RootSwComposition`.

Each delegation port to which a transformed signal is mapped must be connected to a single port on an atomic SWC. RTA-RTE will raise errors if a delegation port on the `RootSwComposition` that is ultimately associated with a transforming `ISignal` is referenced by multiple `DelegationSwConnectors` or if the inner port is a delegation port on a `CompositionSwComponentType`.

The port interfaces through which transformed data is communicated must be directly compatible so that the `DelegationSwConnector` does not reference a `PortInterfaceMapping` - RTA-RTE does not support the use of data conversion in combination with data transformation. RTA-RTE will raise an error when a `DelegationSwConnector` that is ultimately associated with a transforming `ISignal` references a `PortInterfaceMapping`.

RTA-RTE does not support the mapping of multiple `SystemSignals` to a data element in a port interface when transformation is in use for that data element. RTA-RTE will raise an error when multiple `SystemSignals` are mapped to the same data element across a delegation port or a connected inner port and any of those signals are associated with a transforming `ISignal`.

RTA-RTE does not support the use of signal fan-in or fan-out (by referencing the same `SystemSignal` from multiple `ISignals`) when transformation is in use. RTA-RTE will

raise an error if multiple `ISignals` reference the same `SystemSignal` and any of those `ISignals` reference a `DataTransformation`.

### 22.2.2 System model elements

For each transformation, the input model must contain the required model elements:

- A `DataTransformation` referencing the required `TransformationTechnologys`
- `TransformationTechnologys`, each referencing a `BufferProperties`

Data transformation is enabled on an `ISignal`-by-`ISignal` basis. To enable data transformation for an `ISignal`, it must reference a `DataTransformation` which in turn must reference at least one `TransformationTechnology`.

Each `TransformationTechnology` referenced from a `DataTransformation` must in turn reference a valid `BufferProperties`. RTA-RTE does not require the `bufferComputation` attribute to be present inside `BufferProperties` as it uses the alternative `XfrmBufferSizeInBytes` to determine the transformer buffer size. RTA-RTE ignores any `bufferComputation` that may be present.

If the requested `DataTransformations` or `TransformationTechnologys` are invalid or are not found in the input model, RTA-RTE will reject the configuration with an error.



*RTA-RTE will not raise errors when a configuration contains `DataTransformations` or `TransformationTechnologys` that are invalid or missing if they are not referenced from any `ISignal` (and so are not required to perform data transformation).*

### 22.2.3 Mapping in the ECU Value File

When transformation is enabled for an `ISignal`, the input configuration must contain `XfrmImplementationMappings` referencing that `ISignal` and the `TransformationTechnologys` aggregated by the requested `DataTransformation`.

Zero or more `XfrmImplementationMappings` are provided in an `EcucModuleConfigurationValues` with `DefinitionRef Xfrm`. Each `XfrmImplementationMapping` identifies the `BswModuleEntry` that will implements a single `TransformationTechnology` on the data transported by an `ISignal`.

An `XfrmImplementationMapping` can contain two references to `BswModuleEntry`s - one for the sending-side transformer implementation and one for the receiving side inverse transformer implementation. In cases where the same `ISignal` is both sent and received on the same ECU and so both references are required, these must be supplied in the same `XfrmImplementationMapping` container. Supplying separate `XfrmImplementationMappings` for the same `ISignal` and `TransformationTechnology` pair will cause RTA-RTE to raise an error.

Each `XfrmImplementationMapping` must include an `XfrmBufferSizeInBytes` parameter that specifies the size of the transformer buffer for that

ISignal/TransformationTechnology combination. RTA-RTE will raise an error when an input configuration contains an XfrmImplementationMapping that does not include the XfrmBufferSizeInBytes parameter.



*RTA-RTE will issue a warning if an input configuration contains any XfrmImplementationMappings that are not required to perform data transformation. To prevent this warning, you should remove all unused XfrmImplementationMappings from the ECU value file.*



*RTA-RTE will reject a configuration with an error if it detects an invalid XfrmImplementationMapping, even if that XfrmImplementationMapping is not required to perform transformation in the system. To prevent this, you should remove all invalid XfrmImplementationMappings from the ECU value file.*

#### 22.2.4 BSW module entries for transformer functions

The functions that are called to perform transformation must be defined as BswModuleEntries. RTA-RTE uses the BswModuleEntry.shortName as the name of the function that it calls to perform the transformation.



*RTA-RTE calls transformer functions based entirely on the BswModuleEntry.shortName. It does check for compliance with the naming scheme detailed in AUTOSAR requirement [SWS\_Xfrm\_00062].*

Prototypes for the transformer functions are declared in the generated Rte.c file before they are used.



*RTA-RTE raises error 2972 when a configuration requires it to generate multiple incompatible function prototypes for a C function that is used as a transformer implementation.*

#### 22.2.5 COM signals

The COM configuration changes when data transformation is used, as data is transmitted in the form of a byte array regardless of its underlying data type. When transformed data is transmitted or received via a 'ComSignal', the user must configure that signal as type UINT8\_N, not the original data type.

The length of the COM signal must be sufficient to convey the data in its transformed format. On the sending side of a communication, the COM signal must be the same size as the transformed data produced by the the last transformer in the chain (by execution order). On the receiving side, the signal must be the same size as the transformed data that is provided to the first transformer in the chain (by execution order).

In both cases, the size of the COM signal will be the same value that is supplied for the XfrmBufferSizeInBytes in the XfrmImplementationMapping that references both the final TransformationTechnology in the chain and the associated ISignal.

## 22.3 Working with data transformation

### 22.3.1 Serialization and data types

Data transformation is used to serialize complex data into, which simplifies the process of transmitting it via COM. The first transformer in a chain (on the sender side) is always of class `SERIALIZER`, and it produces output as a byte array. Subsequent transformers in the chain (such as the E2E transformer) perform further transformation on the serialized data in this byte-array form before it is sent via COM.

RTA-RTE supports serialization of primitive, array and struct data, each of which may be typed by an `ApplicationDataType` or an `ImplementationDataType`. Data to be serialized is passed to the serializing transformer function as a pointer to the `ImplementationDataType` associated with the `VariableDataPrototype`. The type of the pointer is determined either directly from its `typeTRef` or via the `DataTypeMapping` associated with its `ApplicationDataType`.

With array data, the data to be transformed is passed to the transformer function as an array reference, i.e. as a pointer to the `ImplementationDataType` of the array element. When serializing arrays whose element type is an array or struct, RTA-RTE must be able to declare an argument to the transformer function of a type that is a pointer to that element type. The array element must therefore be defined as a separate `ImplementationDataType`, and then used in an `ImplementationDataTypeElement` of category `TYPE_REFERENCE`.



*RTA-RTE will reject any configuration where data transformation is used with an array if the `ImplementationDataTypeElement` is not a `TYPE_REFERENCE`.*

### 22.3.2 Transformer error reporting

RTA-RTE can provide detailed transformer error feedback to callers of `Rte_Read` and `Rte_Write` APIs, enabling them to react to specific failures of individual transformers in a chain. The error information is transmitted by extending these APIs with an additional OUT parameter named `transformerError`.

The `Rte_Read` and `Rte_Write` APIs are extended for a port whenever its `PortPrototype` is referenced by a `PortAPIOption` that has the attribute `errorHandling` set to `TRANSFORMER-ERROR-HANDLING`.



*The APIs are extended even when transformation is not requested for any of the data elements provided or required by that port, in which case, the error information provided always indicates that no transformation error occurred.*

If the APIs are not extended, RTA-RTE provides basic notification that a transformer error has occurred to the callers of the `Rte_Read` and `Rte_Write` APIs via the return codes for those APIs, but detailed information is not be available.

# **Part VI**

# **Advanced Concepts**



## **23 RTE Generation**

---

### 23.1 Identifier length

---

AUTOSAR R4.0 increased the maximum identifier length to 127 characters from the 31 characters allowed in previous releases.

Additionally a new RTE generation parameter can be defined in the ECU configuration that sets the maximum number of characters that are significant from the perspective of the toolchain. If two identifiers generated by RTA-RTE are not distinguishable within the specified limit then a warning is issued.

The default value of the `RteToolChainSignificantCharacters` parameters is 31. The default can be changed using either the `--toolchain-significant-len` or via the `RteToolChainSignificantCharacters` parameter within the ECUC file.

## 24 Operating System Considerations

---

### 24.1 OS Trigger Selection

---

By default RTA-RTE creates OS objects, e.g. a schedule table or alarms, to implement periodic events. In case this is not desired, RTA-RTE supports the AUTOSAR *OS Interaction* mechanism where the ECU configuration can define an existing OS object to schedule periodic RTE and BSW events.

#### 24.1.1 Alarms

---

To use an existing `OsAlarm` to schedule a periodic event the ECU configuration must reference the alarm in the event's task mapping (see Section 18.4 for details on runnable to task mapping).

If the task mapping also specifies an `OsEvent` to use then the generated RTE code will use the OS event to trigger the RTE event.



*If the task mapping also specifies an `OsEvent` then the alarm expiry must actually set the event!*

RTA-RTE's use of an existing alarm is subject to the following constraints:

- The alarm expiry must either activate the task containing the runnable or set an OS event in the task. RTA-RTE does not explicitly check this – if the alarm fails to activate the task or set the event then the RTE event will not be triggered.
- The alarm period must be the same as the periodic event's period.

If multiple timing events with the same period are mapped to the same task and all are triggered by the **same** `OsAlarm` then RTA-RTE will optimize the generated task body. The level of optimization possible depends on whether or not expiry of an alarm sets an event or activates a task. In the best case when all runnables are Category 1 and all are activated by the same alarm then RTA-RTE will add no additional code to the task body.

#### 24.1.2 Schedule Table

---

One or more AUTOSAR schedule tables can also be used to trigger runnables. As with OS alarms the runnable's task mapping must specify the schedule table entry which either activates the runnables task or sets an event.

### 24.2 Task Recurrence

---

RTA-RTE can construct OS mechanisms, typically either periodic alarms or schedule tables, to activate periodic runnables. If this is not the desired behaviour and you want to use alarms or schedule tables that already exist in your wider OS configuration then the `--task-recurrence` command-line option can be used to both disable RTA-RTE's generation of OS-based mechanism and to specify the recurrence rate for each task or `OsEvent` used for periodic runnables.



*RTA-RTE does not validate that the specified task or OsEvent occurs at the rate specified using the --task-recurrence command-line option.*

As an example, assume that task taskA contains two mapped RTE events with periods 10ms and 20ms. The --task-recurrence command-line option can be used to specify the rate (in seconds) at which the task will be activated as follows:

```
--task-recurrence taskA=0.01
```

When given the above option RTA-RTE can accept runnable mappings whose period is an integral multiple of 0.01s. If this constraint is violated, for example with an RteEvent triggering a runnable with a period of 0.005s, then a build-time error is raised.

If Os events are required to schedule runnables within a task (e.g. when periodic and sporadically triggered runnables are mixed in a task) then the --task-recurrence command-line option can be used to specify the recurrence rate for an individual event. For example, assuming a 10ms RteEvent is triggered by OsEvent evA in taskA one could use:

```
--task-recurrence taskA.evA=0.01
```

When the --task-recurrence command-line option is used once then it **disables** RTA-RTE's generation of OS-based mechanisms for all periodic runnables and therefore the option should give a recurrence rate for all tasks that contain periodic runnables. This can be either by specifying the option multiple times or by specifying the option's parameter as a comma-separated list of task/rate pairs, for example:

```
--task-recurrence taskA.evA=0.01,taskA.evB=0.01
```

### 24.3 Schedule Points

RTA-RTE supports the specification of *schedule points* within generated tasks. When a schedule point is reached execution of a non-preemptive task is suspended if there is a higher priority non-preemptive task waiting. Schedule points can be specified for each instance of an RTE event; thus it is possible to have a schedule point only within certain tasks when runnables from different SW-C instances are mapped to different tasks.

As with the selection of use OS alarm the configuration of a schedule point occurs within the ECU configuration's runnable to task mapping. The mapping has an extra parameter, RteOsSchedulePoint, that defines whether the schedule point is omitted (NONE) or included (CONDITIONAL or UNCONDITIONAL).

A conditional schedule point only occurs if the runnable is executed. An unconditional schedule point is always inserted after the runnable's position within the task.

**ETAS** *The addition of task schedule points is RTA-RTE specific*

## 24.4 Basic and Extended Tasks

---

The RTE generator generates the task bodies based on the RunnableEntityMappings in the ECU description file. The RTE generator creates “glue” code in the task bodies to ensure that the correct runnable is executed only in the correct conditions, e.g. if two TimingEvents with different periods are mapped to the same task, then pre-scalers are implemented so that the runnables do not enter every time the task enters.

In OS Terminology, a task that may wait on an event is an *Extended Task*. The RTE must use an extended task when a Runnable Entity may wait, e.g. there is a WaitPoint configured in the configuration file, or the RunnableEntity is a synchronous client.

If all the RunnableEntityMappings for a given task refer only to periodic RTEEvents (TimingEvents), then the RTE Generator configures an efficient periodic activation of the task, e.g. by Schedule Table or Alarms.

If all the RunnableEntityMappings for a given task refer only to sporadic RTEEvents (events that are not TimingEvents) then the task is activated by the RTE when the conditions are detected, e.g. in the COM callback for received data.

If the RunnableEntityMappings for a given task refer to a mixture of periodic and sporadic RTE Events, then the RTE generator uses OS Events to handle the independent activation of the Runnables, and the task becomes an *Extended Task*, regardless of whether or not a Runnable contains a WaitPoint.

For some projects it might not be possible to use extended tasks; e.g. your chosen Operating System does not support them. To prevent the RTE generator from using OS events in generated code the following steps should be taken:

- Do not map periodic and sporadic runnable entities to the same task.  
When such a mapping exists, the RTE generator uses extended tasks and OS events to ensure the correct runnable is executed in response to each RTE event (but see [24.5](#) below). This restriction does not apply to server runnables that are *pure* since RTA-RTE simply uses a direct function call.
- Do not use WaitPoints (i.e. blocking RTE API calls).  
The implementation of a WaitPoint requires the use of OS Events and therefore any task to which the runnable is mapped becomes an extended task.
- Avoid synchronous inter-task client-server communication unless the server is marked as subject to concurrent execution.  
The implementation of synchronous inter-task client-server communication uses an OS Event so that the client can wait for the server’s reply therefore the client’s task becomes an extended task. This restriction does not apply if the server is *pure* since RTA-RTE simply uses a direct function call.

The efficiency of generated tasks can be maximized by adhering to the following mapping suggestions:

- Do not map runnable entities triggered by timing events with different periods to the same task.

While the RTE generator supports arbitrary mapping of runnable entities triggered by timing events to the same tasks it does this by implementing counters within the glue code so that slower triggered runnable entities are executed at the correct rate. If all runnable entities mapped to the task have the same period then these counters are not required.

Note that the period of the task is related to the greatest common divisor of the period of the mapped runnables, e.g. runnables at 200ms and 100ms may result in the task being activated every 100ms, but runnables at 200ms and 180ms result in the task being activated every 20ms.

If a task contains only periodic runnables and those runnables all have the same period, the the task is activated at that period and the Runnables are unconditionally entered—no counters are required.

- Avoid explicitly associating an OS event with an RTE Event mapping for a TimingEvent in the ECU configuration.

When the creation of OS event for TimingEvent mappings is entirely within the domain of RTA-RTE then it can optimize the event usage for events with the same period. This can reduce the number of OS events compared to the one-OS-event-per-mapping approach of the ECU configuration.

## 24.5 Forced-basic semantics

 *This is a vendor-specific feature not present in the AUTOSAR standards*

To support users who cannot use extended tasks, e.g. because their selected OS does not support them, RTA-RTE supports *forced-basic semantics*. This allows the mixture of periodic and sporadic runnables on the same task without the use of OS Events by means of a special Runnable activation scheme.



*This feature does not allow Runnables with WaitPoints to be mapped to basic tasks. A Category 2 Runnable still requires an Extended Task.*

The configuration of forced-basic semantics can be set for specific tasks using vendor-specific configuration items in the RTE Module Configuration, or set globally for all tasks on the command line (see *RTA-RTE Reference Manual*).

When a task is marked with forced-basic semantics, and it contains a mixture of periodic and sporadic runnables, then the task's activation policy is the same as if it only contained periodic runnables. The sporadic runnables are placed in the task guarded by activation flags, which are set by the RTE when a Runnable becomes due for activation.

As an example, assume that a task contains two runnable entities, a sporadic (e.g. DataReceivedEvent) and periodic (TimingEvent) runnable entity, where the periodic runnable is mapped at position 1 within the task. Figure 24.1 shows conventional

AUTOSAR-compliant activation and dispatch. In this case the sporadic runnable runs as soon as it is activated.

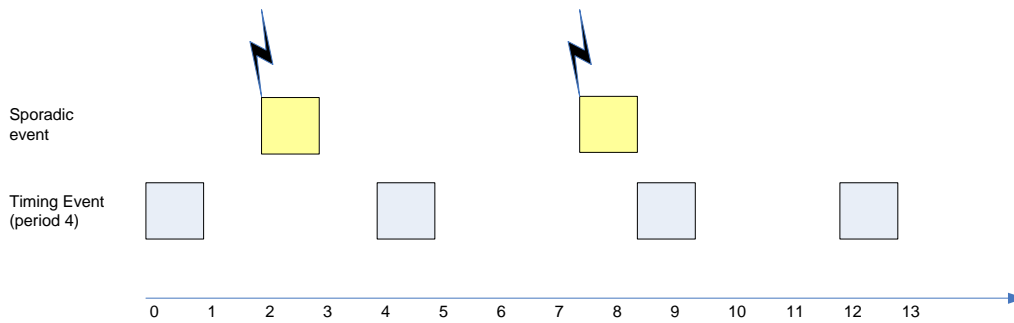


Figure 24.1: Runnable activation without “force basic” semantics.

Figure 24.2 shows the same activation pattern as Figure 24.1 but this time “force basic” semantics are enforced for the task containing the two runnables. Note that while the activation of the sporadic runnable happens at the same point in time the dispatch (execution) of the runnable is delayed until the periodic runnable has run.

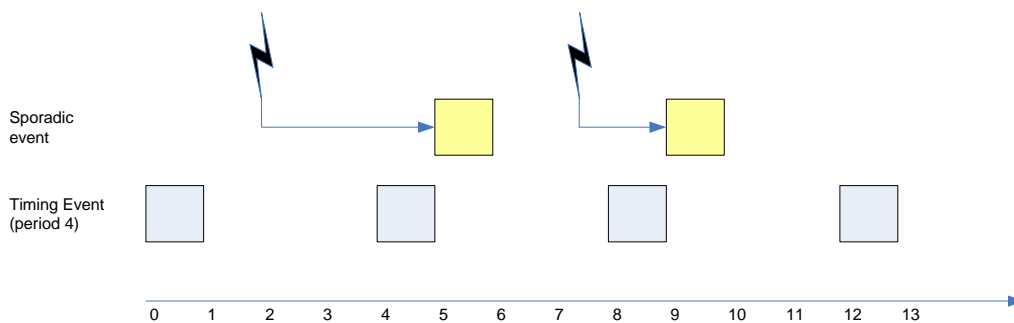


Figure 24.2: Runnable activation using “force basic” semantics.



*Mixing periodic and sporadic events on a basic task introduces latency on the sporadic events between the conditions occurring and the event being recognized.*

Figure 24.2 shows task taskA with forced-basic semantics containing a Runnable re\_te1 started by TimingEvent with a 4s period, and a Runnable re\_dre1 started by DataReceiveEvent. At runtime, taskA is started every 4s and re\_te1 is always entered. When data is received on the relevant port and dataitem, the RTE runtime sets a flag to record that re\_dre1 is due to run, but does not activate taskA because that would disturb the timing characteristics of re\_te1. Thus, in this example, re\_dre1 might experience up to nearly 4s of lag between the data arriving and the Runnable entering.

Note that in these circumstances, you can take advantage of prescalers for periodic runnables. If taskA in the example above also contained a Runnable started by TimingEvent with a period of 100ms, then taskA would enter every 100ms rather than 4s, and the lag of re\_dre1 would be reduced (although the overhead of activating taskA more frequently must also be considered).

It is not recommended to use forced-basic semantics if your OS supports Extended Tasks.

## 25 Understanding Deployment Choices

---

Your choice of component deployment at ECU integration time can have a large effect on overall vehicle system performance. In this chapter you will learn what effects your choices have on the behaviour of the RTE generator when it creates task bodies and on the behaviour of the RTE API for users.

Communicating components may be distributed across a physical network or co-located on the same ECU. There are three possible distribution patterns:

- **Intra-task** — communication occurs between components executed by the same task (and is therefore serialized).
- **Inter-task** — communication occurs between notionally concurrent tasks in the same ECU.
- **Inter-ECU** — communication occurs between concurrently executing components located on physically distinct ECUs.

In all cases, RTA-RTE is responsible for managing the communication between software components.

The following sections describe, in terms of message sequence charts, how RTA-RTE manages the communication for each of the allocations and each communication paradigm.

### 25.1 Intra-task

---

Intra-task communication occurs between runnable entities executed by the same task. There is low overhead and it is easy to reason about the sequence of execution.

#### 25.1.1 Intra-task Sender-Receiver

---

RTA-RTE implements intra-ECU communication itself—COM is only used when the destination is located on a remote ECU. This means that:

Requests received from local and remote components are combined into a single queue so starvation of either local or remote requests cannot occur.

The implementation of intra-ECU communication is more efficient since it avoids the expense of passing signals to COM only for COM to return the signal to the RTE as it is for a local destination.

Figure 25.1 illustrates the sequence of steps the thread of control executing a runnable entity in an application software component takes when performing intra-task sender-receiver communication.

In detail, the steps illustrated in Figure 25.1 are:

1. The sending component initiates communication via the RTE API call `Rte_Send` (or `Rte_Write` API call if the data is unqueued).



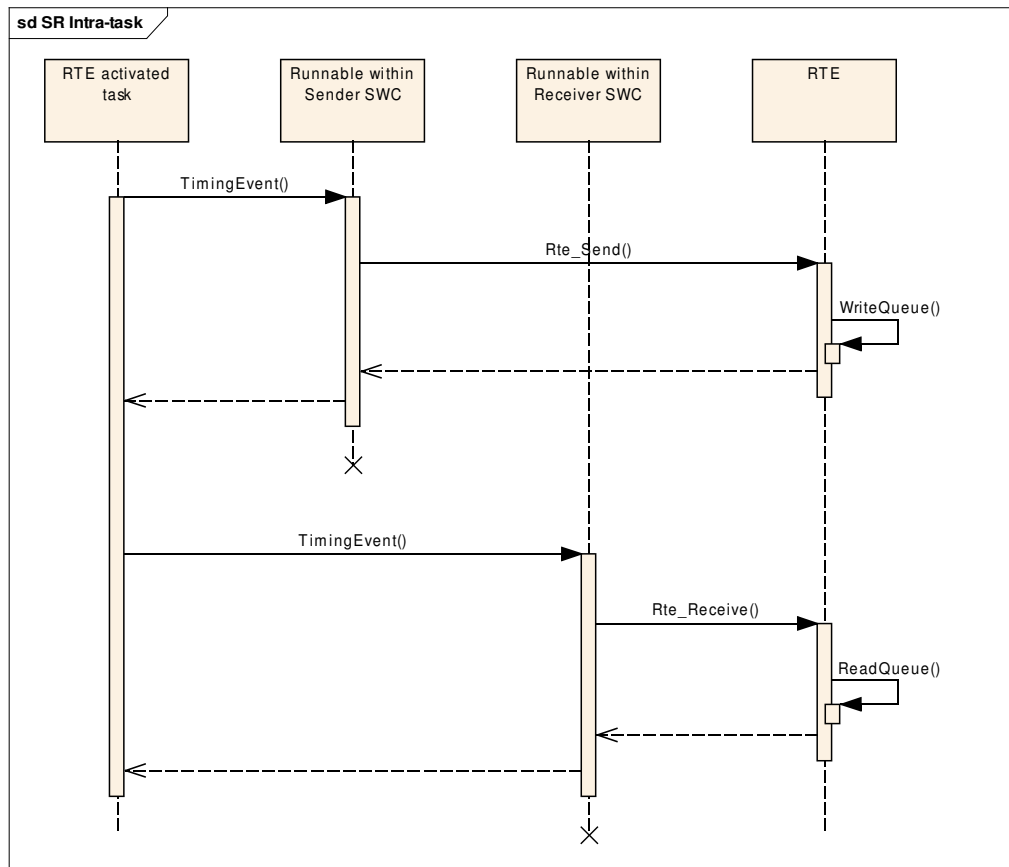


Figure 25.1: Intra-Task sender-receiver communication

2. The generated RTE function places the data into the receiver's queue and returns control to the sender. Note that the "queue" may consist of a single element if the data is un-queued.
3. The system switches from executing a runnable entity in the sending component to executing a runnable entity in the receiving component. Since both runnables are mapped to the same task no context switch is required.
4. The receiving component reads the data from the queue using the Rte\_Receive API call (or Rte\_Read if the data is unqueued).
5. The RTE returns the data and the receiving component processes the data. If queued communication is used and no data is present in the queue then RTE\_E\_NO\_DATA is returned.

### 25.1.2 Intra-task Client-Server

Figure 25.2 illustrates a single prototype of the intra-task client-server communication model that has been specified using the "CLIENT\_MODE synchronous" attribute.

The steps involved in Figure 25.2 are:

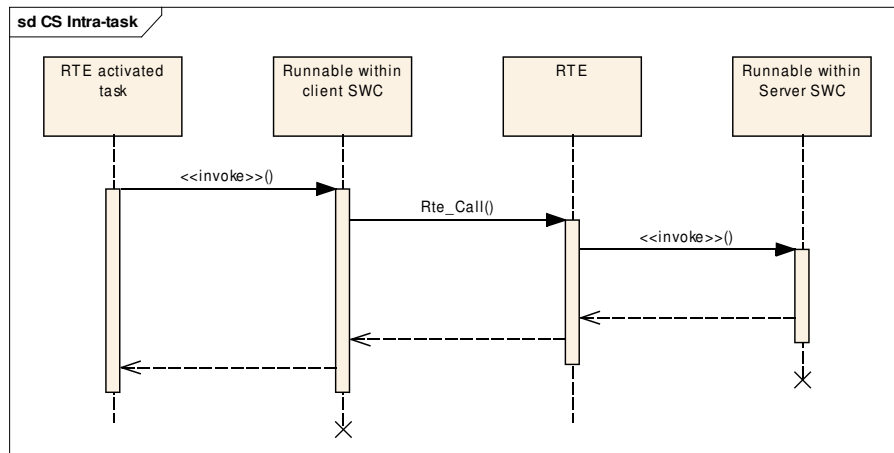


Figure 25.2: Inter-task sender-receiver

1. The client invokes the server via the `Rte_Call` API call.
2. The generated RTE function directly invokes the runnable entity specified by the user in the server component. This is “safe” with regard to re-entrancy since the server, which is mapped to the same task as the caller, cannot otherwise be executing.
3. The return value (if present) is passed to the RTE generated code.
4. The RTE returns control to the client.

## 25.2 Inter-task

Inter-task communication occurs between components that have been mapped to the same ECU but are executed by different threads of control (tasks).

For inter-task communication the RTE performs the role of the physical network, e.g. a CAN bus, present in inter-ECU communication and transports signals and data between tasks. Since communication occurs within an ECU, and not over a real network, it is intrinsically reliable (data can be neither lost nor corrupted) and instantaneous (data arrive immediately it is sent).

### 25.2.1 Inter-task Sender-Receiver

Figure 25.3 illustrates the sequence of steps the thread of control executing a runnable entity in an application software component takes when performing inter-task sender-receiver communication. The example uses reliable, buffered and blocking communication.

In detail, the steps illustrated in Figure 25.3 are:

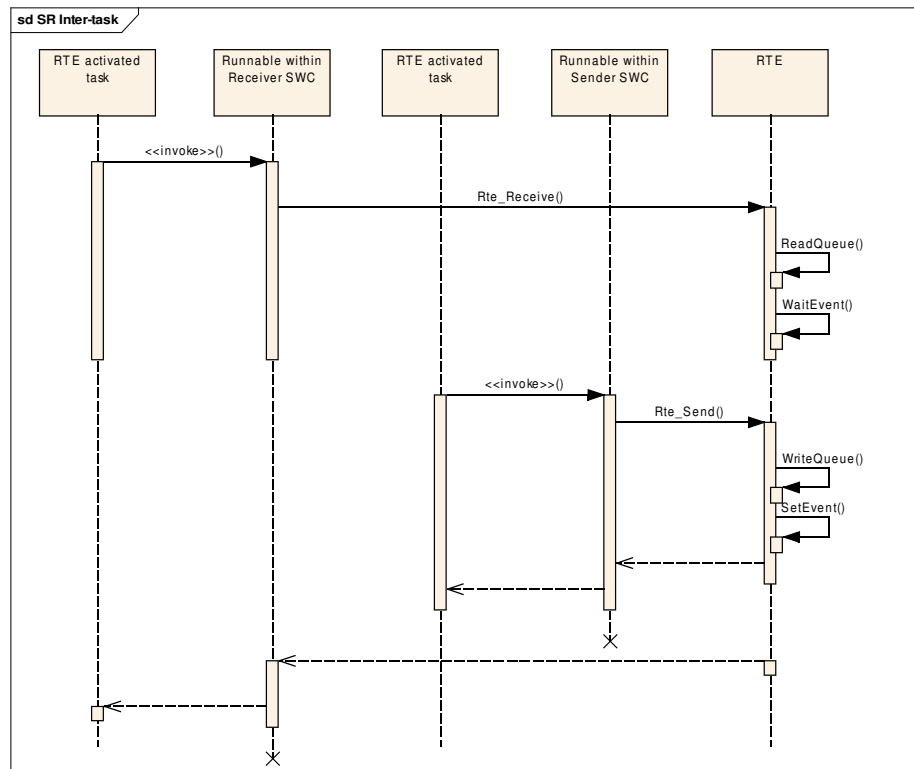


Figure 25.3: Inter-task sender-receiver (Queued reception with wake-up-of-wait-point receive mode)

1. Receiver indicates that it is ready to read data by invoking the Rte\_Receive API call. The Receive API is mapped by the RTE API mapping onto a generated function in the RTE (the function call is shown as a transfer of flow of control from component to RTE).
2. The RTE generated function translates the read request into a fetch from an RTA-RTE managed queue. The calling thread of control will be suspended if the queue is empty otherwise data is returned immediately and execution continues at the final step.
3. The sending component sends the data. This occurs independently of the receiver attempting to read and process data.
4. RTA-RTE appends the data to an RTE managed queue using the internal WriteQueue function at which point the suspended thread of control (the receiver) is resumed automatically.
5. RTA-RTE returns control to the sender.
6. The read from the OS queue returns via the return from RTA-RTE's invocation of

the ReadQueue function. The suspended thread resumes execution of RTE generated code.

7. Control returns to the receiver's thread of control.

The sequence in Figure 25.3 demonstrates a queued reception using "wake up of wait point" receive mode. Using a different combination of attributes the following behaviors are possible:

- **Un-queued reception**—the receiver "polls" the buffer for the current data by invoking the generated Rte\_Read function which will always return immediately with the most recent data available. If the "poll" is scheduled to occur at regular intervals, i.e. from an event-triggered runnable entity, the receiver is said to be cyclic.
- **"Activation of runnable entity" receive mode**—the receiver is activated (dispatched) when data is placed in the buffer. The receiver always consumes a single data item that is passed as the parameters of the runnable entity. In this case the component is only executed when data is available and the runnable entity is the receiver's task.

## 25.3 Inter-task Client-Server

AUTOSAR COM does not support remote function invocation. Therefore inter-task (and inter-ECU) client-server communication is implemented using paired sender-receiver communications. The initial communication sends the server invocation request (and parameters) to the server component. The server subsequently uses a second transmission to return the reply (if any).

The transformation of inter-ECU client-server into paired sender-receiver communications is illustrated in Figure 25.4.

The translation from client-server to paired sender-receiver communication is performed automatically by the RTA-RTE generator. The RTE generator creates API functions that perform the communication between the client and server and map the client-server communication to the necessary internal RTE and COM calls necessary for the paired sender-receiver communication. These functions form part of the infrastructure that is generated to permit communication to occur between application software components.

The RTE generated functions also marshal arguments and, if necessary, interact with the communication service to send the request and data over the network.

Figure 25.5 illustrates inter-task client-server communication using the asynchronous invocation. In detail, the steps illustrated in Figure 25.5 are:

1. The client component invokes the server using the generated Rte\_Call API call.
2. The RTE appends the request parameters to the server's RTE-managed queue of client requests and activates the task mapped to the server's runnable entity.

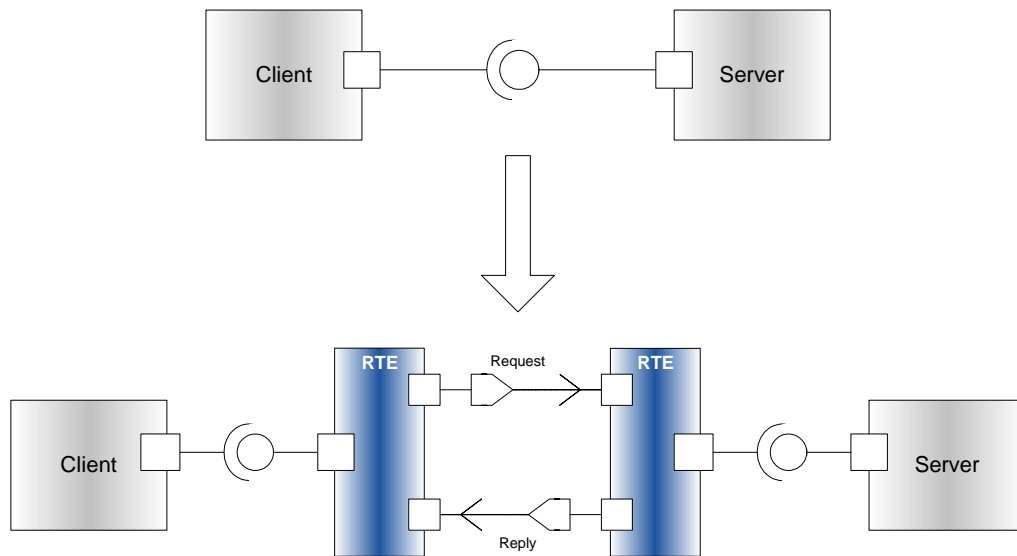


Figure 25.4: Client-server implemented as paired sender-receiver communication

3. The RTE returns control to the client component.
4. The task assigned by the RTE to the server's runnable entity executes and RTA-RTE generated code within the task reads from the RTE-managed queue of client requests.
5. The RTE generated code invokes the server's runnable entity passing the request parameters to the server.
6. The server processes the request and returns the reply. Since the communication is inter-task there is no physical network involved.
7. The RTE writes the reply to the client's result queue. The client runnable (or, another runnable in the component) can then access the result using the `Rte_Result` API.

## 25.4 Inter-ECU

Inter-ECU communication occurs between application software components that are located on physically separate ECUs and therefore each component can be executed concurrently. All Inter-ECU communication operates over a physical network (e.g. CAN bus, LIN, MOST, etc.) and as well as communication between ECUs in a vehicle is considered to include the following cases:

One of the "ECUs" is located in an off car computer, e.g. on the Internet, and is accessed via a wireless network.

Multiple CPUs within a single ECU that communicate via a communications mechanism internal to the ECU such as dual-ported RAM.

The semantics of inter-ECU communication is set by the specified communication attributes.

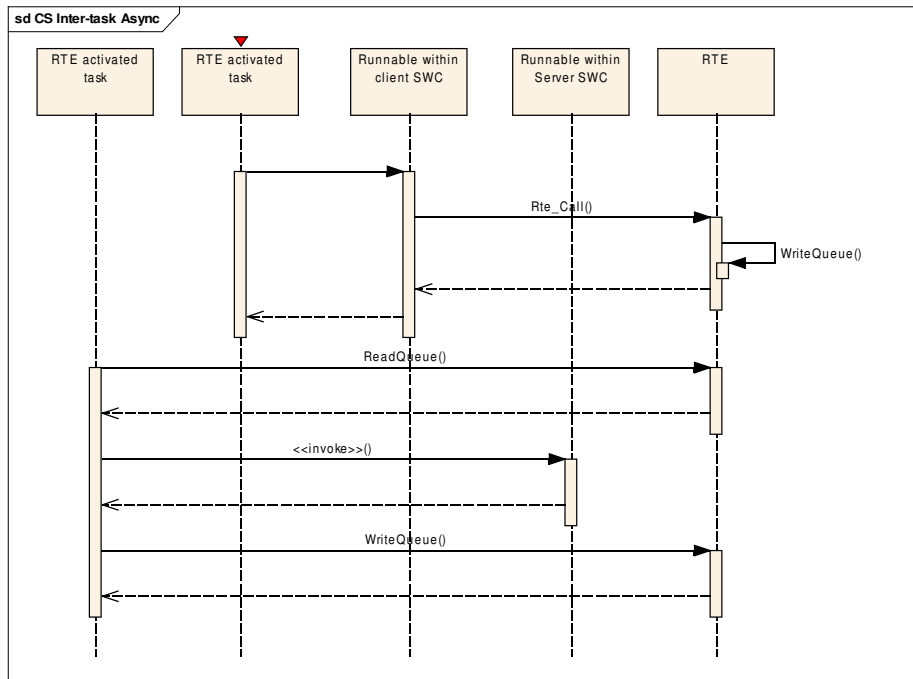


Figure 25.5: Inter-task asynchronous client-server communication

### 25.4.1 Inter-ECU Sender

The sequence of steps the thread of control executing a runnable entity in an application software component takes when it initiates a sender-receiver Inter-ECU communication using reliable asynchronous communication is illustrated in Figure 25.6.

In detail, the steps illustrated in Figure 25.6 are:

1. The sending component sends a signal using the `Rte_Send` (or `Rte_Write` API call if data is un-queued). The send is mapped by the RTE generator onto a generated RTE API function.
2. The RTE invokes the communication service to queue the data from the signal. The communications layer queues the data for transmission over the network and returns.
3. The generated RTE function returns control to the software component. Note that at this stage transmission is neither necessarily complete nor acknowledged.
4. The COM module's `MainFunction` performs the network transmission and invokes a callback defined within the RTE to indicate successful transmission.
5. The RTE's callback activates the runnable entity defined within the SW-C's RTE event definition.

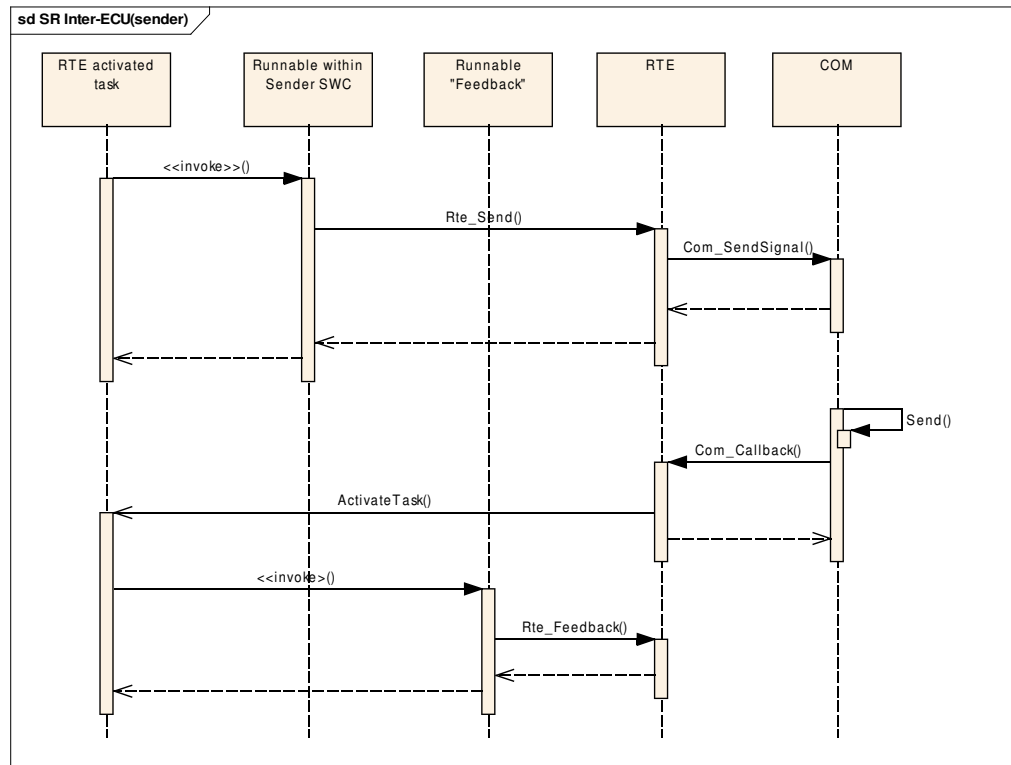


Figure 25.6: Inter-ECU sender-receiver (Sender-side) with transmission acknowledgment.

6. The “feedback” runnable entity is invoked and in turn invokes the generated RTE function to read the feedback status.

#### 25.4.2 Inter-ECU Receiver

Figure 25.7 illustrates the opposite end of the communication where the receiving runnable entity accepts data from the network and processes it. The example in Figure 25.7 uses queued buffering with “wake up of wait point” receive mode. All steps occur within the same ECU.

In detail, the steps illustrated in Figure 25.7 are:

1. The receiving component attempts to read existing queued data. The Rte\_Receive API is mapped by the RTE API mapping generator onto a generated function in the RTE.
2. The RTE attempts to read data from an RTA-RTE managed queue. The RTE generated API function will behave differently depending on the communication attributes:

If the “wake up of wait point” receive mode is specified then a blocking read is required and execution of the calling thread will be suspended if no data is available.

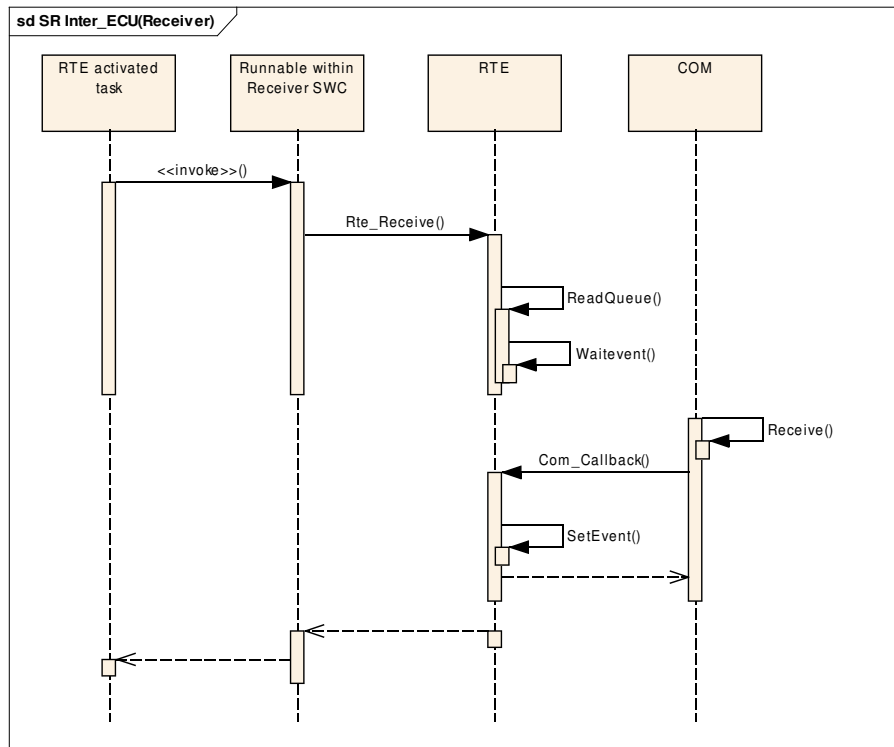


Figure 25.7: Inter-ECU sender-receiver (Receiver-side) with queued reception and “wake\_up\_of\_wait\_point” receive mode.

Otherwise a non-blocking read is required and RTE\_E\_NO\_DATA will be returned if no data is available.

3. For the purposes of this example we assume no data is available and so the calling thread of control is suspended. If data were available the attempt to read data from the queue would succeed and the call would return immediately without voluntarily relinquishing control of the CPU.
4. AUTOSAR COM on the receiving ECU receives data from the network—in this case synchronously with a call to the COM module’s MainFunction—and notifies RTA-RTE that new data has been received using a callback.
5. The RTE sets the OS event associated with the RTA-RTE managed queued which causes the suspended Rte\_Receive API to resume.
6. Once the suspended task has been resumed, it returns control to the generated RTE which in turn returns the newly received data to the original caller.



### 25.4.3 Inter-ECU Client

Figure 25.8 illustrates the client-side actions for synchronous inter-ECU client-server communication. Figure 25.8 illustrates the server-side actions.

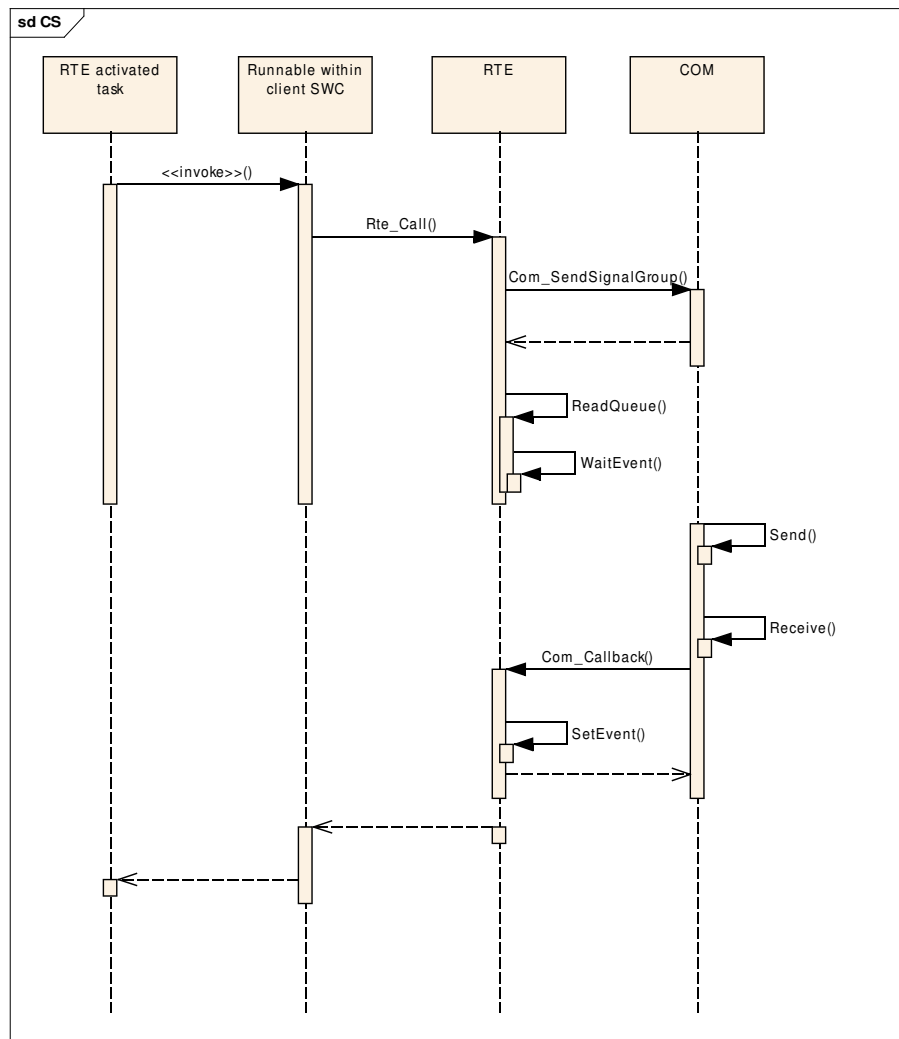


Figure 25.8: Inter-ECU synchronous client-server (Client-side).

1. The client “invokes” the server—this is mapped to the invocation of the generated `Rte_Call` function within the RTE.
2. The call parameters are marshaled (i.e. serialized) by the RTE and passed to the communication service for transmission as a signal group. The communication service queues the data for transmission and returns control to the generated RTE function.

3. The RTE attempts to immediately read the server's reply. However since the reply is not available the task is suspended.
4. The COM main function is invoked and transmits the client's request.
5. At some point in the future, COM detects reception of the server's reply and invokes the appropriate callback in the RTE. The callback sets the OS event associated with the client's reply queue and returns control to COM.
6. The suspended task resumes and returns the server's reply to the caller.

#### 25.4.4 Inter-ECU Server

---

Figure 25.9 illustrates the server-side actions for synchronous inter-ECU client-server communication.

1. The server's COM receives the request and invokes an RTE generated callback.
2. The RTE callback reads the client request and writes the data to a queue managed by the RTE. The callback then activates the task mapped by the RTE to trigger the server's runnable entity. The RTE then returns control to the communication service.
3. The task assigned by the RTE to the server's runnable entity starts, reads the request from the queue and executes the server runnable entity.
4. The reply from the server runnable entity is collected. The reply from the server runnable entity is sent to the client using COM. The task mapped to the server's runnable entity terminates.

The above examples uses synchronous communication so an OS event is used to block the client until the return value is available. The RTE also supports asynchronous communication these require only minor changes to the sequence, for example:

- The wait on the OS event does not occur and hence the caller is resumed immediately after the client's request has been passed to COM.
- The setting of the OS event when the server's reply is available may not occur depending on the configuration of the client's `AsynchronousServerCallReturn` event. For example, configuration of activation of the runnable entity receive mode may require the RTE to activate a task instead.

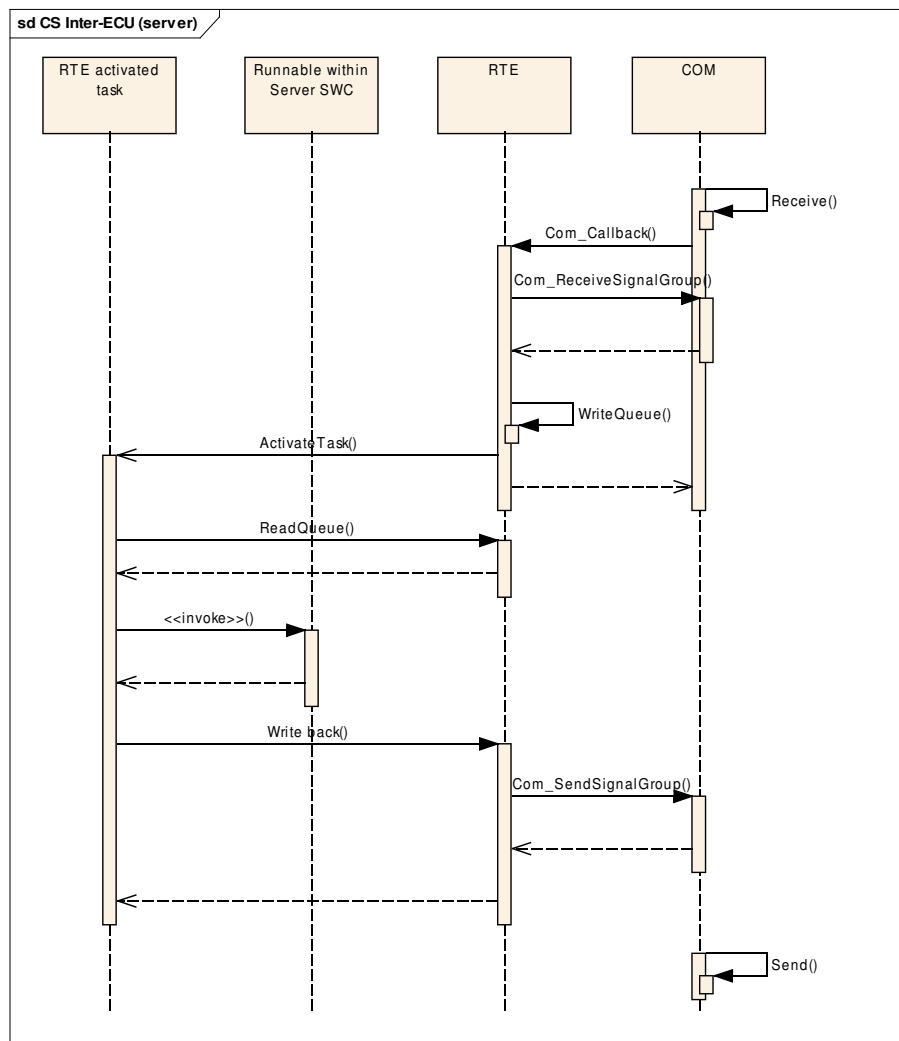


Figure 25.9: Inter-ECU synchronous client-server (server-side).

## 26 Optimization

---

Several optimizations are possible to influence the memory usage and runtime of the generated RTE.

### 26.1 Buffers for Inter-ECU Reception

---

The `--optimize` option tunes RTE generation for runtime or memory use.

`--optimize=speed` (the default) generates an RTE that uses RTE-managed buffers for inter-ECU communication. This is faster when there are multiple readers, but requires a separate memory allocation.

`--optimize=size` generates an RTE that invokes the COM API each time the value is read by the application. This is slower when there are multiple readers, but but reduces RAM usage and, if the value is only accessed once, code size.

For any individual data element, direct access to COM is enabled if all of the following are true:

- The data element prototype is unqueued (a queued data element requires an RTE managed queue).
- Reception is explicit (implicit reception requires an RTE managed buffer).
- There are no intra-ECU transmitters (an inter-ECU transmitter requires an RTE managed buffer into which the data is written and to ensure that inter-ECU and intra-ECU data is coordinated then the former must use the same buffer).
- Transmission is 1:1.

Direct access to COM for a data item is disabled if:

- The data element has “keep” invalidation policy, the invalid value is equal to the initial value and “handle never received” is false, or
- The data item has “handle never received” enabled.

The `optimize` option has no effect on Contract phase execution of RTA-RTE.

### 26.2 Direct invocation of the RTE API

---

The API mapping created in the application header file will directly invoke the generated RTE API functions if the software-component is known not to be multiply instantiated (see Section 8.14).

For example, consider a simple `Rte_Write` API:

```
Std_ReturnType Rte_SWCa_Write_p1_d1(uint8 data)
{
    Rte_000000 = data;
}
```

```
    return RTE_E_OK;
}
```

RTA-RTE can inline the API to a C comma-expression within the SWC's application header file and thus avoid the function call overhead:

```
extern uint8 Rte_000000;
#define Rte_Write_p1_d1(data) ( ( Rte_000000 = (data) ), RTE_E_OK )
```

The inline comma-expression can be used in the same way as a call to the generated RTE function but without the overhead of the function call.

This optimization occurs in both Contract and RTE generation phases.

In addition to examining the input XML, in RTE generation phase RTA-RTE also counts the number of actual instances on the ECU for which the RTE is being generated and if this is one applies the optimization even if the software-component is potentially multiply instantiated.

## 26.3 Sender-Receiver Communication

---

### 26.3.1 Direct Read and Write

---

Direct invocation of the RTE API does not eliminate the function call overhead of the RTE API. Therefore RTA-RTE includes an additional optimization in RTE phase that in-lines the generated functions for explicit sender-receiver communication and explicit inter-runnable variable access into the RTE API mapping.

The optimization occurs if RTA-RTE judges that the generated API function is sufficiently simple for in-lining to be beneficial; the function body must consist of a simple assignment with or without concurrency control. Optimization is therefore dependent on:

- Communication uses primitive types.
- The number of destination components; ideally one.
- The runnable to task mapping of the sender and receiver. RTA-RTE eliminates explicit data consistency mechanism if the task mapping guarantees protection. Thus ideally the sender and receiver will be in the same task or mapped to tasks at the same priority.

Optimization to direct read/write occurs in both vendor and compatibility operating modes

### 26.3.2 Implicit Communication

---

AUTOSAR requires task-specific copies of the data read/written by implicit communication to ensure that values remain stable during the execution of a runnable entity and to ensure that they are not visible to other runnables until after the writing runnable has terminated.

The task-specific copies consume RAM and therefore RTA-RTE includes the `--implicit-use-global-buffers` command-line option to directly use global buffers where possible.

When enabled (option parameter '1' or '2') the global buffer is accessed directly by the Implicit API calls provided **all** of the following constraints are met:

- A shared buffer is required (if separate buffers for reader and writer are required then optimization to use global buffers will occur for readers only).
- No "status" information is required.
- There are no COM mappings for the data instance. This restriction is required to ensure data stability – if COM mappings exist then COM can preempt the runnable and disturb the global buffer during runnable execution.
- There is a single destination buffer – i.e. one global buffer shared by all receivers. For intra-ECU communication this will typically be the case unless filters are applied by one or more receivers. This restriction is necessary to ensure that the generated implicit API macros have a single value to write.

In addition one of the following constraints must also be met:

- All events in the task are 'fast init'. Such events are executed once only before the periodic runnables hence there can be no conflict so global buffers can be directly accessed.
- Receivers can directly access the global buffer if the provider cannot preempt any receiver (in which case the receiver's data cannot be changed by the provider during its execution and the required data stability during runnable execution is preserved).
- The provider can directly access the global buffer if no requirer can preempt (and no receiver will see the value written to the global until the provider runnable terminates and thus the require implicit semantics are preserved).

## 26.4 Client-Server Communication

---

### 26.4.1 Direct Function Invocation

---

RTA-RTE includes an additional optimization in RTE phase that inlines the invocation of server runnable entities.

The optimization to direct invocation only occurs if the generated API function is suitable for in-lining. Optimization is therefore dependent on:

- All client-server communication with the server must be synchronous.

- The server is mapped as “pure” (see Section 8.6.1), or the client and server are mapped to the same task.

Optimization to direct function invocation occurs in RTE phase in both vendor and compatibility operating modes.

## 26.5 Function Elision

The application of the above optimizations is automatic when the RTE is generated—there are no specific optimizations to enable or disable optimizations. However in-lining functions does not, in itself, reduce the code size of the generated RTE since RTA-RTE is required to emit the functions to support object-code delivery of software-components.

Therefore to enable elision of function bodies from the generated RTE:

- Ensure the SWC is delivered as source code.
- Disable indirect API for all ports of the software-component.
- Disable ‘take address’ for the ports of the software-component.

### 26.5.1 Return Code Optimization

Many AUTOSAR RTE API functions return an Std\_ReturnType status using the function return value. When RTA-RTE optimizes the API to an inline macro it uses a C comma-expression to both perform the function’s actions and to set the return value as the final value in the expression.

If we again consider the simple simple Rte\_Write API:

```
Std_ReturnType Rte_SWCa_Write_p1_d1(uint8 data)
{
    Rte_000000 = data;
    return RTE_E_OK;
}
```

However now assume that port “p1” is unconnected and therefore the Write API is defined as discarding the data. The generated API function is now simply:

```
Std_ReturnType Rte_SWCa_Write_p1_d1(uint8 data)
{
    return RTE_E_OK;
}
```

The RTE generator can still inline the function but now uses a simpler comma-expression containing a single element:

```
#define Rte_Write_p1_d1(data) ( (Std_ReturnType)RTE_E_OK )
```

The inline comma-expression can again be used in the same way as a call to the generated RTE function but without the overhead of the function call. However when the macro replacement occurs the statement is now a C null-statement and the compiler may now complain about a “...statement has no effect.”. To disable this warning/error from the compiler ensure that one of the following occurs:

1. The return value is assigned to a variable.
2. The return value is tested (e.g. for equality) in a condition.
3. The return value is cast to void if not used.

For example, if the caller is not interested in the return value use a cast to void:

```
(void)Rte_Write_p1_d1(5u);
```

## 26.6 Init Runnables

The AUTOSAR mode mechanism provides ModeSwitchEvents that can be used to trigger runnable execution when an application mode is entered. This feature can be used to provide “init” runnables that are executed when the RTE starts and perform necessary initialization for a SWC instance.

However for a runnable that only runs once at system start the infrastructure code created by RTA-RTE to support the AUTOSAR mode mechanism is superfluous. Therefore RTA-RTE provides a mechanism to optimize activation for specified ModeSwitchEvents by mapping them to specified *FastInit* tasks. The use of a *FastInit* task both simplifies the activation code necessary for the runnable associated with the event as well as removing the normal infrastructure code created by RTA-RTE to support the AUTOSAR mode mechanism.



*An event that is specified as FastInit will not be activated by RTA-RTE during a user-triggered mode switch. Instead the FastInit task must be activated by user code at the appropriate time.*

The command-line option `--fast-init` specifies an absolute reference to either a SWC type or to a ModeSwitchEvent within an internal behaviour. A reference to a ModeSwitchEvent causes that event to be activated by a non-AUTOSAR compliant mechanism (a function call from a task body where the task is activated externally from the RTE). This avoids the complexity inherent in AUTOSAR compliant activation for mode switch activations. A reference to an SWC type causes all ModeSwitchEvents in the type to be subject to FastInit.

The command-line option `--fast-init` can be specified multiple times to apply *FastInit* to multiple ModeSwitchEvents.

RTA-RTE places no restrictions on the mapping of ModeSwitchEvents within *FastInit* tasks other than events subject to normal AUTOSAR-compliant activation cannot be mapped to the same task. As a consequence it is possible to map all “init” runnables from all mode instances to the same *FastInit* task.





*It is permitted to map ModeSwitchEvents for different mode transitions (e.g. both entry to "init\_mode" and entry to "run\_mode") to the same FastInit task. When this is done care must be taken since all runnables will be activated when the task is dispatched irrespective of the current mode.*

## 26.7 Data Consistency

---

The RTE generator uses RTE\_ATOMICxxx macros to apply data consistency mechanisms to ensure that assignments are atomic. Since the RTE generator has no knowledge of target capabilities it is not possible for it to exploit platform specific knowledge to eliminate use of RTE\_ATOMICxxx macros. However the RTE generator supports two mechanisms where the integrator can apply such knowledge to improve efficiency of generated core.

### 26.7.1 Atomic assignments

---

The `--atomic-assign` can be used to specify one or more platform types that are always subject to atomic assignment. When specified the RTE generator will not generate RTE\_ATOMICxxx macros for the types which can then subsequently enable further optimizations of the API mapping.

The following types can be specified with the `-aa` option:

- `uint8`, `uint16` and `uint32` (includes the AUTOSAR `char` and opaque metatypes and the `Std_ReturnType`).
- `sint8`, `sint16` and `sint32`.
- `boolean`.
- `float32` and `float64`

### 26.7.2 Macro elision

---

After RTE generation individual RTE\_ATOMICxxx macros can be elided by appropriate definitions when compiling RTE generated code (including application header file).

For example, defining `RTE_16BIT_ATOMIC` forces the `RTE_ATOMIC16` macro to have no run-time effect.

For more information on eliminating the effect of RTE\_ATOMIC macros please consult the *RTA-RTE Toolchain Integration Guide*.

### 26.7.3 Inline Functions

---

RTA-RTE can use either an OS resource or interrupt blocking to implement an exclusive area. RTA-RTE can optimize access to the OS resource or the locking of interrupts in the RTE Enter/Exit API call by inlining the API mapping in the application header file. When this inlining occurs the applications access to exclusive area is as efficient as possible since RTA-RTE imposes no run-time overhead.

However the inlining of the access to the OS resource or the locking of interrupts means that the application header file now includes OS API calls within the API mapping and therefore the SWC application code must include `Os.h`.

**ETAS** *When an Enter/Exit API call is inlined within the API mapping the SWC implementation must include `Os.h`.*

## 26.8 Tips

---

The optimization potential of the generated RTE is maximized by:

- Use 1:1 sender-receiver communication.
- Use primitive types for explicit sender-receiver communication.
- Map provider and requirer to the same task or to tasks that cannot preempt.
- Use source-code delivery with disabled indirect API when possible.
- Wherever possible, mark a server as “pure”.
- Avoid software-components delivered as object-code.
- Where possible uses single-instantiated software-components.

## 27 RTE Architecture

---

The RTE itself is a passive entity and thus does not have any of its own threads of control. RTE code is only executed when an external entity—such as a software component—invokes generated code in the RTE.

Software components are given access to API calls through the software component's application header file which is custom generated for each software component type. All API calls use the form:

```
Rte_<Call_Name>_<Discriminator>(instance, args)
```

Where <Discriminator> can be either the port and data item/operation name or, where this is not appropriate, an identifier extracted from the input. The application header file performs an important role for the RTE because it binds the API call names onto specific implementations depending on how component prototypes are mapped to an ECU.

The implementations of the RTE API calls themselves are custom generated by the RTE generator for each software component. The functionality of the RTE is provided mainly by the code written to the `Rte.c` file at generation time.

The RTE API has been designed so that calls can be made efficiently at runtime. This is achieved by providing an API mapping for each software component that binds the component's API to permit the direct invocation or even mapping calls to managed reads and writes to global data.

The following sections outline how these mechanisms work.

### 27.1 Component Data Structure

---

The component data structure defines the RTE API calls that a particular component instance can invoke based on the component type's declaration (i.e. ports) and internal behavior (per-instance memories, exclusive areas, etc.).

The declaration of the component data structure data type is created by the RTE generator and is internal to the RTE—application software should only access members within the structure via RTA API definitions in the component's application header file.

The component data structure data type is the same for all instances of a component type. Hence the structure can be uniquely named using only the name of the component type irrespective of the number of times the component is actually instantiated, e.g. for SWC type `MyComponent` the component data structure data would be `Rte_CDS_MyComponent`.

```
typedef struct  
{  
    /* Structure definition for MyComponent */  
} Rte_CDS_MyComponent;
```

The component data structure includes all information necessary for the generated RTE API mapping within the application header file to invoke the correct function for the particular instance. Information within the component data structure includes a function table used for indirect invocation of generated custom functions and definitions of state that is not shared between component prototypes.

The component data structure function table is used for indirect invocation of generated custom functions (see Section 27.3) for details of the API mapping).

When an instance of the component data structure is declared in software component header file, the generated function table is initialized to point to the correct generated custom functions for the component. The functions themselves and the instance of the component data structure are generated by RTA-RTE in the `Rte.c` file.

The component data structure is instantiated within `Rte.c` once for each component instance that is mapped to the ECU instance for which the RTE is being generated. For a software-component that can be multiple instantiated, the component data structure instances are named using an RTA-RTE generated instance name as follows:

```
Rte_Instance_<InternalComponentInstanceName>
```

However, for a software-component that can only be instantiated once per ECU instance, AUTOSAR requires that the single instance of the component data structure created within `Rte.c` is named as follows:

```
Rte_Inst_<SWCTypeName>
```

The names of software-component types must be globally unique to avoid type conflicts within the generated code.

## 27.2 Component Instance Handle

---

When multiple prototypes of the same software component type are mapped to the same ECU, then the RTE must be able to disambiguate access to their individual state. When the RTE is generated in the RTE phase, each prototype is assigned a unique RTE instance handle that is used to identify the component prototype data structure.

Each component prototype that is mapped to an ECU instance has an instance of the component data structure written to `Rte.c`.

All activity within a software component is handled by runnable entities and in each case the RTE can identify the correct instance of the relevant component data structure and pass the appropriate handle to the runnable entity when it is activated.

When runnable entities make calls back to the RTE they pass in their “self” parameter that identifies the prototype (i.e. the current `Rte_Instance` handle) to the generated RTE API functions. The handle identifies the instance to the RTE and enables selection of the correct functionality.

## 27.3 The RTE API Implementation

The `Rte.c` file contains the implementations of all custom RTE API calls required by component instances mapped to the ECU instance. Each call takes the form:

```
Rte_<call>_<prototype>_<port>_<data/op>(<args>)
```

Where `<call>` is an RTE API name (e.g. `Call`, `Send`), `<prototype>` is the component prototype, `<port>` is the port on which the call operates and `<args>` are the interface arguments.

### 27.3.1 API Mapping

The RTE API names are based on a root names, such as `Rte_Call`, combined with both the port name and data item or operation name as suffixes. The RTE API names do not include the component type or instance name. This omission permits the component to be developed before its deployed (and hence the instance name) is known.

Two different components can define identically named ports (as well as identically named operations or data items) and thus have the same RTE API names for accessing the data items or operations. However, the generated RTE API functions for the different components must be unique since they can be radically different, for example, one may be intra-ECU and highly optimized whereas another may be inter-ECU and use the communication service. The RTE API functions are created during the “RTE” phase of RTE generation at which point the component deployment, and hence instance name, as been determined. The RTE generator can therefore use the component instance name to ensure that generated functions can be uniquely identified.

Since the RTE API name as seen by the location neutral software component and the generated API function are different the RTE generator defines an API mapping in the component’s application header file from the RTE API to the names of the generated API functions defined in the RTE module.



*The API mapping written to the component’s application header file is created automatically by the RTE generator—the same RTE API is therefore available to a component irrespective of the number of instances deployed and whether or not source-code is available.*

All RTE API call names are formed from a root name and the port name and operation/-data item name as a suffixes. The RTE API mapping is responsible for mapping the RTE API name to the correct implementation of the RTE function in `Rte.c`. The API mapping permits an RTE generator to include targeted optimization as well as removing the need to implement functions that act as de-multiplexers from generic API calls to particular functions within the generated RTE.

The API mapping is implemented as a set of macros. The API mapping is uniquely generated for each component and written to the component header file.

There are typically two kinds of mappings:

- Indirect mappings
- Direct mappings

### 27.3.2 Indirect Mapping

---

Indirect is the most general form and is implemented in the API mapping by the RTE generator when optimizations cannot be applied; for example the instance name of a component is not known or a component may be multiple instantiated.

The RTE API mapping from API name to generated function is defined in the application header files that are created during the first “contract” phase of RTE generation will always invoke the generated API functions via the component’s instance handle. This provides the necessary level of indirection to support binary-code software components as well as supporting multiple component instances.

When indirect invocation is used, execution of the generated API functions occurs via the appropriate entry in the function table defined in the component data structure (see Section 20.1). The fields within the function table are initialized by the RTE generator.

Indirect invocation permits both binary-code software components and multiple instances of components to be supported since the component instance name does not need to be fixed when the component is compiled and the RTE can initialize a different function table for each instance therefore invoking different generated API functions for each instance.

An indirect mapping is used when only a multiple prototypes of a software component are mapped to a single ECU instance. The basic form of the mapping in this case is:

```
#define Rte_<Call>_<p>_<d/op> (self, data) \  
    ((self)-><Call>_<p>_<d/op>(data))
```

Where <Call> is the RTE API call type (Call, Send, Write etc.), <p> the port name and <d/op> is the port data item/operation name.

### 27.3.3 Direct Mapping

---

It is possible to perform a number of optimizations on the API mapping that can eliminate the run-time costs of indirect invocation when both the component instance name is known and it is known a priori that only a single instance of the component will be mapped to an ECU. Direct invocation also requires that the component is available in source-code form since the optimization can only be applied after component deployment has occurred.

With direct invocation it is possible to completely elide the instance handle and therefore impose zero run-time overhead. However, it remains in the RTE API signature for compatibility with the requirements of indirect invocation.

A direct mapping is used when only a single (source code) prototype of a software component is mapped to an ECU instance. The basic form of the mapping in this case

is:

```
#define Rte_<call>_<p>_<d/op>(self,data) \  
    Rte_<call>_<c>_<p>_<d/op>(data)
```

Where <call> is the RTE API call type (Call, Send, Write etc.), <p> the port name and <d/op> is the port data item/operation name

Note in the direct case that the call through the instance handle self has been elided since only a single instance is in use.

For intra-task communication made by “source-code” components the call may be further mapped to a direct write to an OS-managed queue or global data item.

# **Part VII**

# **Support**



## 28 **Contact, Support and Problem Reporting**

---

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries      [www.etas.com/en/contact.php](http://www.etas.com/en/contact.php)

ETAS technical support      [www.etas.com/en/hotlines.php](http://www.etas.com/en/hotlines.php)

The RTA hotline is available to all RTA-RTE users with a valid support contract.

[rta.hotline.uk@etas.com](mailto:rta.hotline.uk@etas.com)

+44 (0)1904 562624. (0900-1730 GMT/BST)

Please provide support with the following information:

- Your support contract number.
- Your AUTOSAR XML and/or OS configuration files.
- The command line that results in an error message.
- The version of the ETAS tools you are using.

## Index

---

### Symbols

C++, [124](#)  
--atomic-assign, [265](#)  
--deviate-shared-mode-queue, [120](#)  
--fast-init, [264](#)  
--have-64bit-int-types, [177](#), [179](#)  
--implicit-use-global-buffers, [262](#)  
--mode-policy, [121](#)  
--operating-system, [223](#)  
--optimize, [260](#)  
--os-file, [223](#)  
--output, [171](#)  
--strict-config-check, [224](#)  
--task-recurrence, [242](#), [243](#)  
--toolchain-significant-len, [241](#)

### A

A2L, *see also* [McSupportData](#), [171](#)  
Acknowledgement request, [78](#)  
Activation of Runnable Entity, [92](#)  
Alive timeout, [81](#)  
API  
    Direct, [124](#), [270](#)  
    Explicit, [133](#)  
    Implicit, [133](#)  
    Indirect, [108](#), [109](#), [124](#), [270](#)  
    Mapping, [269](#)  
    Parameter passing, [123](#)  
Application Header, [124](#)  
    API Mapping, [269](#)  
    Contract Phase, [38](#)  
ApplicationArrayDataType  
    Example of, [49](#)  
ApplicationPrimitiveDataType  
    Example of, [47](#), [48](#)  
ApplicationRecordDataType  
    Example of, [50](#)  
ApplicationValueSpecification  
    Example of, [67](#)  
Array Type  
    As function parameter, [57](#)  
ArrayValueSpecification  
    Example of, [67](#)  
Assembly Connectors, [156](#)  
    Port Mapping, [160](#)  
Asynchronous Server Call Returns Event, [84](#), [99](#), [130](#)

Atomic Assignment, [265](#)

AUTOSAR, [10](#)

Basic Software, [146](#)

Data Types, [46](#)

Development Process, [42](#)

Element, [32](#)

NVRAM, [140](#)

Package, [28](#)

## B

Background Event, [85](#)

Background event, [87](#)

Base type

Native Declaration, [53](#)

BSW, [10](#)

API Name generation, [152](#)

Background event, [148](#)

Connections, [181](#)

External Trigger event, [148](#)

Instantiation, [180](#)

Internal Trigger event, [149](#)

Mode Sender Policy, [113](#)

Mode Switch Ack event, [152](#)

Mode Switch event, [150](#)

Modules, [146](#)

Periodic event, [148](#)

Schedulable Entity, [147](#)

Scheduler, [181](#)

Task Mapping, [181](#)

BSW Phase, [41](#)

## C

C, [124](#)

C/S, [10](#)

Calibration, [136](#)

Access Point, [94](#)

export in McSupportData, [171](#)

Interface, [73](#)

Parameters, [170](#)

Prototype, [73](#)

Category

ARRAY, [59](#)

DATA\_REFERENCE, [60](#)

FUNCTION\_REFERENCE, [60](#)

HOST, [59](#)

STRUCTURE, [59](#)

TYPE\_REFERENCE, [59](#)

UNION, [59](#)

VALUE, [59](#)

Characteristic, *see also* Calibration Parameter

Client

Asynchronous, [97](#), [130](#)

Synchronous, [97](#)

COM

Module configuration, [34](#), [226](#)

Command-line option

samples, [42](#), [127](#)

Communication

Calibration, [20](#)

Client-server, [20](#), [135](#), [218](#), [249](#)

Explicit, [88](#)

Implicit, [89](#)

Inter-ECU, [210](#)

Modes, [111](#)

Sender-receiver, [18](#), [78](#), [81](#), [215](#), [248](#)

Complex Type

Inter-ECU, [216](#)

Component Data Structure, [267](#)

Component Instance Handle, [17](#), [268](#)

Composition

BSW, [180](#)

Connectors, [156](#)

Instantiation, [156](#)

SWCs, [155](#)

Type, [155](#)

CompuMethod

Example of, [47](#), [48](#), [65](#), [66](#)

CompuRationalCoeffs, [65](#)

Precision of, [65](#)

ConstantReference

Example of, [68](#)

ConstantSpecification

Example of, [68](#)

Contract Phase, [37](#)

Conventions, [9](#)

## **D**

Data Conversion

Compatible versus Convertible, [176](#)

Concepts, [62](#), [173](#)

Modelling Application Types, [61](#)

Precision of, [178](#)

Data Element, [69](#)

Data Receive Error Event, [84](#)  
Data Received Event, [84](#), [128](#)  
Data Send Completed Event, [85](#)  
Data Write Completed Event, [85](#)  
Delegation Connector, [158](#)  
Development Process, [42](#)

## E

### ECU

- Instantiation, [199](#)
- Mapping Services, [202](#)
- Mapping SWCs, [200](#)
- Ports, [198](#)
- Type, [198](#)

ECUC, [10](#)

ECUC Description, [33](#)

Exclusive Area, [102](#), [137](#)

- Configuration, [102](#)
- Explicit, [103](#)
- Implementation Method, [104](#)
- Implicit, [103](#)

extended task, [244](#)

External Trigger Occurred Event, [85](#), [101](#)

## F

FactorSiToUnit, [63](#)

FactorSiToUnit

- Precision of, [178](#)

Fast Init, [264](#)

Filter, [81](#)

FlatInstanceDescriptor, [167](#)

FlatMap, [167](#)

Flatmap, [167](#)

forced-basic semantics, [245](#)

## I

Identifier Length, [241](#)

Initial Value, [79](#)

Inline functions, [260](#), [265](#)

Inter-ECU

- Behavior, [253](#)
- Configuration, [210](#)
- Signal Mapping, [215](#), [218](#)

Inter-runnable Variable, [106](#), [136](#)

Interface, [18](#), [69](#)

- Calibration, [73](#)
- Client-server, [72](#)

- Mode group, [71](#), [113](#)
- Mode-Switch, [71](#)
- Nv-Data, [70](#)
- Sender-receiver, [69](#)
- Trigger, [74](#)
- Internal Behavior, [83](#)
  - Events, [84](#)
  - Exclusive Area, [102](#)
  - Inter-runnable Variable, [106](#)
- Internal Trigger Occurred Event, [85](#), [100](#)
- Invalidation, [80](#)
- IOC, [10](#)

## **M**

- McSupportData, [171](#)
  - Structure of, [172](#)
- McSupportFile
  - Location of, [171](#)
- Measurement
  - export in McSupportData, [171](#)
- Memory Abstraction
  - Runnable, [110](#)
- Minimum Start Interval, [86](#)
- MISRA
  - 6.3, [57](#), [60](#)
- Mode
  - Access Point, [107](#), [113](#)
- Mode Dependency, [117](#)
- Mode Graph, [118](#)
- Mode Instance, [118](#)
- Mode Manager, [112](#)
- Mode Switch
  - Asynchronous, [114](#)
  - Synchronous, [114](#)
- Mode Switch Event, [85](#), [116](#)
- Mode Switched Ack Event, [85](#)
- Mode User, [112](#)
- Modes, [111](#)
  - Configuration, [111](#)
  - Distributed Mode Users, [121](#)
  - Fast init, [119](#)
  - Queue length, [112](#)
  - Shared Mode Queue, [120](#)
  - Switch management, [115](#)

## **N**

- Namespace, [122](#)

NumericalValueSpecification

Example of, [67](#)

Nv-Block, [140](#)

Nv-Data Element, [70](#)

NVRAM

Configuration, [189](#)

GetMirror, [193](#)

NvMNotifyInitBlock, [194](#)

NvMNotifyJobFinished, [194](#)

SetMirror, [193](#)

NVRAM mirror, [141](#)

## O

Object-code delivery, [39](#)

OffsetSiToUnit, [63](#)

OffsetSiToUnit

Precision of, [178](#)

Operation, [72](#)

Call, [95](#)

Inter-ECU, [218](#)

Operation Invoked Event, [85](#), [94](#), [128](#)

Optimization, [260](#)

OS

Category 2 Tasks, [205](#), [244](#)

Counters, [224](#)

Module configuration, [34](#), [224](#)

Schedule Table, [225](#)

Tasks, [41](#), [203](#)

Os Interaction

Alarm, [242](#)

Schedule Point, [243](#)

Schedule Table, [242](#)

Task Recurrence, [242](#)

## P

Package, [28](#)

Merge, [29](#)

Partial Record Mappings, [166](#)

PDU

Instantiation, [214](#)

Type, [212](#)

Per-instance Memory, [137](#)

periodic RteEvents, [244](#)

PhysicalDimension

Example of, [64](#)

Port

Defined Arguments, [108](#)

- Options, [108](#)
- Port Mapping, [160](#)
- pre-scalers, [244](#)
- Precision
  - CompuRationalCoeffs, [65](#)
  - FactorSiToUnit, [178](#)
  - OffsetSiToUnit, [178](#)
  - Unit, [178](#)
- Primitive Type
  - Inter-ECU, [216](#)
- Provided Port, [76](#), [88](#)
  
- Q**
- Queue
  - Configuration, [80](#)
  
- R**
- RecordValueSpecification
  - Example of, [68](#)
- References, [30](#)
  - Absolute, [30](#)
  - Instance, [32](#)
  - Relative, [31](#)
- Required Port, [76](#), [90](#)
- ResultingProperties, [172](#)
- Role
  - NvMNotifyInitBlock, [194](#)
  - NvMNotifyJobFinished, [194](#)
- RootSwCompositionPrototype, [167](#)
- RTA-OS, [10](#)
- RTA-OSEK, [10](#)
- RTE, [10](#)
  - Architecture, [267](#)
  - Module configuration, [34](#)
  - Namespace, [122](#)
  - Output, [40](#)
  - Start, [139](#)
  - Stop, [139](#)
- RTE Phase, [39](#)
- RteEvents
  - periodic, [244](#)
  - sporadic, [244](#)
- Runnable entity, [86](#)
  - Activation, [92](#)
  - Arrival rate, [86](#)
  - Category, [22](#)
  - Configuration, [127](#)



- Direct Trigger, [100](#)
- Implementation, [24](#), [126](#)
- Initialization, [115](#), [264](#)
- Sample, [42](#), [127](#)
- Symbol, [86](#), [127](#)
- Task Mapping, [203](#)

## S

- S/R, [10](#)
- SchM, [10](#)
- Server
  - Concurrent, [95](#)
  - Invocation by client, [95](#)
- Service Connector, [159](#)
- Services, [25](#)
  - Mapping, [202](#)
- sporadic RteEvents, [244](#)
- SWC
  - Behavior, [21](#), [83](#)
  - Implementation, [122](#)
  - Implementation selection, [201](#)
  - Initialization, [115](#), [264](#)
  - Instantiation, [16](#), [39](#), [109](#), [125](#), [156](#)
  - Mapping to ECUs, [200](#)
  - NvBlock, [75](#), [140](#)
  - Type, [14](#), [75](#)
- Synchronization
  - Configuration, [183](#)
  - Mode Groups, [185](#)
  - Runnable Entities, [187](#)
  - Triggers, [186](#)
- System Signal, [210](#)

## T

- task
  - bodies, [244](#)
  - extended vs basic, [244](#)
- TEXTTABLE, [55](#)
- Timing Event, [84](#), [87](#), [225](#)
- Types
  - Application, [46](#)
  - Arrays, [48](#), [57](#)
  - Base, [50](#)
  - Conversion between, [173](#)
  - Enumerated, [55](#)
  - Implementation, [51](#)
  - Mapping, [60](#)

PhysicalDimension, [64](#)

Platform, [52](#)

Records, [49](#), [58](#)

Reference, [54](#)

Structures, [49](#), [58](#)

Types, modeling with CompuMethod, [64](#)

Types, modeling with Unit, [64](#)

## **U**

Unit

Example of, [64](#)

## **V**

VFB, [10](#), [13](#)

VFB Tracing, [228](#)

Enable, [228](#), [231](#)

Events, [228](#)

## **W**

Wake up of Wait Point, [93](#), [99](#)

## **X**

XML, [10](#), [26](#)