ETAS

RTA-RTE V6.8.0

Toolchain Integration Guide

DRIVING | EMBEDDED EXCELLENCE

Copyright

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

©Copyright 2019 ETAS GmbH, Stuttgart.

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

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

Revision: 92501 [RTA-RTE 6.8.0]

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

Contents

1 Introduction **4**
1.1 About this Manual . . . 4
1.2 Who Should Read this Manual? . . . 4
1.3 Document Conventions . . . 4

2 Compiler Toolchain **6**
2.1 Adding a new Compiler Toolchain . . . 6
2.2 AUTOSAR Headers . . . 6
2.3 C Library . . . 8

3 Compiler Abstraction **9**
3.1 Memory Section Description File . . . 9
3.2 File Selection . . . 10
3.3 Example . . . 11

4 Dependencies and Outputs **12**
4.1 Contract Phase . . . 12
4.2 RTE Phase . . . 13
4.3 Compile-time . . . 15
4.4 Link-time . . . 17
4.5 Run-time . . . 17
4.6 Templates . . . 17

5 Legacy Systems **21**
5.1 Resource Reuse . . . 21
5.2 Fixed Task Set . . . 21

6 Operating System **23**
6.1 Overview . . . 23
6.2 AUTOSAR R4.0 OS support . . . 24
6.3 AUTOSAR R3.0 OS support . . . 26
6.4 AUTOSAR R1.0 OS support . . . 29
6.5 OSEK 2.2.3 support . . . 31
6.6 OS Abstraction Layer . . . 33

7 Build Environment **38**
7.1 Option sub-files . . . 38
7.2 Make . . . 38
7.3 Incremental Build . . . 39
7.4 Error reporting . . . 39

8 Contact, Support and Problem Reporting **42**

1 Introduction

1.1 About this Manual

This guide provides a reference for the integration of RTEs generated by RTA-RTE with multiple operating systems and third-party build environments such as “make” or integrated development environments (IDEs).

- Chapter 2 describes how to adapt RTA-RTE to a new compiler toolchain by providing versions of four AUTOSAR header files.
- Chpater3 describes how the compiler abstraction used by RTA-RTE can be adapted for specific project requirements using a *Memory Section Description File*.
- Chapter 4 describes the inputs and outputs of the RTE generator.
- Chapter 5 describes how RTEs generated by RTA-RTE can be integrated within legacy systems that have a fixed task set.
- Chapter 6 details how multiple operating system (both AUTOSAR compliant and non-AUTOSAR operating systems) are supported in the generated RTE and how OS selection affects the requirements of compilation of generated code.
- Chapter 7 describes RTA-RTE’s error reporting mechanism and how this can be customized to accommodate the requirements of third-party build environments such as make or with third-party IDEs.

1.2 Who Should Read this Manual?

The *RTA-RTE Toolchain Integration Guide* is intended for the software engineer who understands the concepts and general techniques of developing an RTE-based application and needs to know key technical detail about the use of RTA-RTE within both existing and new applications.

It is assumed that the reader is familiar with the *RTA-RTE User Guide* and *RTA-RTE Reference Manual*.

Related Documents

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

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

1.3 Document Conventions

Applicable version

> Information, e.g. an XML fragment, that is applicable to a subset of AUTOSAR revisions appears within a framed paragraph. The range of applicable revisions is displayed in the frame header.

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

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

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

2 Compiler Toolchain

2.1 Adding a new Compiler Toolchain

The output of the RTE generator is standard ANSI C and therefore RTA-RTE does not include command-line options to adapt the generation for a particular compiler toolchain. The sole exception is an option to define the non-standardized pre-processor directive used to issue a warning.

When targeting a new compiler toolchain it is necessary to provide AUTOSAR header files that define the AUTOSAR compiler abstraction and platform types. Depending on the toolchain, the header files can either be obtained directly from AUTOSAR or can be adapted from exemplar files included with RTA-RTE.

2.1.1 Errors and Warnings

Code generated by RTA-RTE includes conditional code that can include both compile time errors and warnings that are raised using C pre-processor directives embedded in the output.

ANSI standardizes the `#error` pre-processor directive but no equivalent standardization exists for a issuing "warning" message and therefore the compiler vendor is free to do whatever they so choose – common variants are `#warn`, `#warning` and `#pragma message`.

RTA-RTE includes an option, `--warn-directive`, to enable the default `#pragma` message directive to be replaced with a toolchain specific variant. For example, the option:

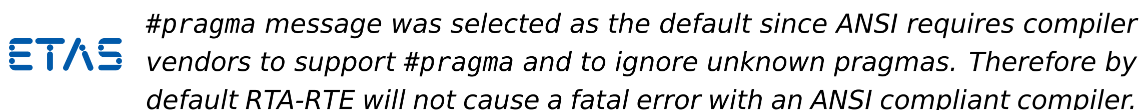
```
-warn=warning
```

causes RTA-RTE to generate:

```
#warning "Warning text"
```

instead of the default:

```
#pragma message "Warning text"
```



#pragma message was selected as the default since ANSI requires compiler vendors to support #pragma and to ignore unknown pragmas. Therefore by default RTA-RTE will not cause a fatal error with an ANSI compliant compiler.

2.2 AUTOSAR Headers

RTA-RTE generated code does use target-specific AUTOSAR header files that may require modification when targeting a new toolchain

The following generic AUTOSAR header files are used by the generated RTE and task bodies:

- Std_Types.h – defines standard types for AUTOSAR basic software.

  A copy of Std_Types.h file is included with distributions of RTA-RTE in {install-fldr}\External\Inc.

- Os.h – defines OS API[1] The selected OS file can be overridden with a command-line option.

  Your distribution of RTA-RTE includes a version of Os.h suitable for RTA-OSEK 5.0 in the file {install-fldr}\External\Inc. If you are not using RTA-OSEK please consult your OS documentation for information on the correct header file.

- Com.h – defines the COM API. (used only when inter-ECU communication is configured).

  RTA-RTE does not include AUTOSAR COM and therefore this file is not included with your distribution.

In addition to the generic header files, generated code uses four target specific header files that may need to be modified when targeting a new toolchain:

- Compiler.h – defines a mapping from the AUTOSAR compiler abstraction to toolchain specific keywords.
- Compiler_Cfg.h – Configuration file for memory and pointer classes.
- Rte_MemMap.h – Configures mapping for variables, constants and code of AUTOSAR modules to individual memory sections.
- Platform_Types.h – defines target and toolchain dependent types.

The AUTOSAR header files must be present on the compiler's include path when generated RTE code is compiled.

Your distribution of RTA-RTE includes exemplar AUTOSAR header files for the following targets:

- RTA-OSEK_Task17x6 – RTA-OSEK 5.0 for Tasking C Compiler v2.2r3 and Infineon TriCore 17x6.
- RTA-OSEK_VRTA – MinGW 3.4.2/Visual Studio and Microsoft Windows using RTA-OSEK 5.0 for PC.

Distributions that include supported for AUTOSAR OS R3.0 and above also include support for the following targets:

- RTA-OS30_TriCoreHighTec – RTA-OS3.0 v2.0 for HighTect C Compiler v3.4.5.2 and Infineon TriCore 17x6.

[1] This file is only used by the RTA-OSEK5.0 OS when the RTE generator is in compatibility mode only.

- RTA-OS30_VRTA – MinGW 3.4.2/Visual Studio Microsoft Windows using RTA-OS3.0 v2.0 for PC.

The exemplar AUTOSAR header files can be used as a reference when the RTE is retargeted for a new toolchain.

2.3 C Library

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

Alternatively, RTA-RTE will use the C library's memcpy function if the symbol RTE_LIBC_MEMCPY is defined when compiling the RTE library and generated code.

The standard function from the C Library may be preferred for certain targets, for example, when the compiler supports a built-in version of the memcpy function that compiles to inline optimal assembler.

3 Compiler Abstraction

RTA-RTE uses the AUTOSAR compiler abstraction macros when declaring or defining global data and generated API functions. Unfortunately the use of the compiler abstraction by an RTE generator has historically been poorly specified by AUTOSAR and this has led to incompatibilities between RTE generators when different implementation decisions lead to different usage of the compiler abstraction, for example, RTA-RTE uses the AUTOSAR `CONSTP2VAR` compiler abstraction macro when declaring `INOUT` and `OUT` function parameters since it is not required to modify where the reference points however other RTE generators use the P2VAR macro.

Attempting to use software components that assume one convention with an RTE generated that assumes a different convention can lead to errors during compilation. Therefore RTA-RTE permits the use of the compiler abstraction to be modified if the RTA-RTE default is not the desired behaviour.

The use of the compiler abstraction can be changed either globally or on a per-project basis using a *Memory Section Description File*.

3.1 Memory Section Description File

The *Memory Section Description File* is an XML file that describes how RTA-RTE should format the compiler abstraction macros in a number of scenarios:

Scenario Name	Description
`FUNC_ARG_IN`	An IN function argument used by a generated RTE API function.
`FUNC_ARG_INOUT`	An INOUT function argument used by a generated RTE API function.
`FUNC_ARG_OUT`	An OUT function argument used by a generated RTE API function.
`FUNC_RETURN`	The return type of a generated RTE API function.
`RAM`	A global variable defined in `Rte.c`.
`ROM`	A global constant defined in `Rte.c`.
`FUNC_LOCALS`	A function-local variable defined in `Rte.c`.
`TYPECAST`	A typecast in RTE generated code.

Each scenario is represented by a `<MemClass>` element within the *Memory Section Description File*, for example:

```
<MemClass>
    <name>ROM</name>
    <typeDef>volatile_ref VAR(type, RTE_DATA)</typeDef>
    <typeDefRef>volatile_ref P2VAR(type, RTE_DATA, ptrclass)</typeDefRef>
```

```
        <typeDefRef2Const>volatile_ref P2CONST(type, RTE_DATA, ptrclass)</
            typeDefRef2Const>
        <constQualifier>false</constQualifier>
        <volatileQualifier>false</volatileQualifier>
        <category>Internal</category>
        <description>Constant data</description>
    </MemClass>
```

The <name> defines the scenario. RTA-RTE then uses the <typeDef>, <typeDefRef> and <typeDefRef2Const> elements to define how to format the compiler abstraction macros for declarations, references to variables and references to constants respectively. The other elements within the <MemClass> are not used by RTA-RTE.

Within a <typeDef>, <typeDefRef> and <typeDefRef2Const> element RTA-RTE performs token replacement to form the final compiler abstraction as follows:

Token	Description
type	Expands to the data type name.
memclass	Expands to the memory class, e.g. RTE_CODE.
ptrclass	Expands to the pointer class, e.g. RTE_APPL_DATA. RTA-RTE uses the pointer class for the reference elements <typeDefRef> and <typeDefRef2Const> and refers to the pointed-to object.
volatile_ref	Expands to volatile if necessary, otherwise the empty string.

The <MemClass> element can be specified multiple times within the *Memory Section Description File*; all such elements are encapsulated within a <MemClasses> element which is itself within the root <MemClassConfig> element.

A full example can be found in the following file:

```
{install-fldr}/Examples/Configuration/memsect.xml
```

3.2 File Selection

The *Memory Section Description File* can be specified either globally for a RTA-RTE back-end processor or selected for each build.

Global Selection

The global default location of the *Memory Section Description File* is specified in the RTA-RTE configuration file:

```
[ Options ]
MemSectFile=memsect.xml
```

The MemSectFile option is specified either as an absolute path or relative to the bin folder of the relevant RTA-RTE backend processor. Therefore the default global location of the *Memory Section Description File* for backend processor is <backend> is:

```
{install-fldr}\bin\<backend>\memsect.xml
```

The *Memory Section Description File* is optional – if not present in the global location then **no error is raised** by RTA-RTE and the default compiler abstraction macros are used.

Command-line Selection

Alternatively the location of the *Memory Section Description File* can be set for a particular run of the RTE generator using the --memory-sections command-line option. For example:

```
--memory-sections:memsect.xml
```

The MemSectFile specified using the --memory-sections command-line option is either an absolute path or relative to the current folder when RTA-RTE is invoked. The setting on the command-line overrides any global setting in the RTA-RTE configuration file.

3.3 Example

Consider a SWC with a required sender-receiver port “r1” containing a data element “array” that uses an array data type. Using the default compiler abstraction the OUT parameter of the Rte_Read API uses the CONSTP2VAR macro:

```
Rte_Read_swcA_r1_array(CONSTP2VAR(unsigned char, AUTOMATIC,
    RTE_APPL_DATA) data);
```

However if a *Memory Section Description File* containing the following XML fragment to redefine how RTA-RTE generates compiler abstraction macros for the FUNC_ARG_OUT scenario:

```
<MemClass>
  <name>FUNC_ARG_OUT</name>
  <typeDefRef>P2VAR(type, AUTOMATIC, ptrclass)</typeDefRef>
  ...
</MemClass>
```

Then the compiler abstraction generated for the OUT parameter uses the P2VAR macro:

```
Rte_Read_swcA_r1_array(P2VAR(unsigned char, AUTOMATIC, RTE_APPL_DATA
    ) data);
```

4 Dependencies and Outputs

The required inputs and generated outputs depend on the operating phase of RTA-RTE.

RTA-RTE uses command-line arguments (see the *RTA-RTE Reference Manual*) to select between “Contract” and “RTE” phases and to specify the set of input files.

4.1 Contract Phase

Contract phase is selected using the `--contract` option. The option takes a single parameter; a reference to the SWC type for which contract phase generation is required.

The SWC type reference must be specified using an absolute reference.

It is not possible to create Contract phase outputs for multiple SWC types in a single execution of the RTA-RTE’s RTE generator.

4.1.1 Inputs

When invoked for Contract phase generation, RTA-RTE expects the following input files:

- The SWC description according to the AUTOSAR Software-Component Template.

Information present in the input but not required by RTA-RTE is ignored. Thus it is not necessary to strip out all non-relevant information when invoking RTA-RTE’s RTE generator.

The SWC description can be included in one or more XML files each of which can contain one or more AUTOSAR packages. Note that individual elements within a package cannot be split across packages, thus, for example, all constituent elements of an application software-component element must be specified within the same package.

4.1.2 Outputs

When invoked for Contract phase generation, RTA-RTE generates the following output files:

- A C header file, `Rte_Type.h`, containing all type definitions encountered within the input files.
- An application header file, `Rte_<name>.h` where `<name>` is the short name of the selected SWC, containing declarations of the RTE API for the SWC type. The generated application header file is used when compiling the SWC either for pre-delivery test purposes or to enable delivery as object-code.

The default location of output files is the same as the folder in which RTA-RTE’s RTE generator is invoked. The `--output` option can be used to explicitly set an output folder.

4.2 RTE Phase

RTE phase is selected using the `--rte-phase` option. The option takes a single parameter; a reference to the ECU instance for which RTE phase generation is required.

The ECU instance reference must be specified using an absolute instance reference. See the *RTA-RTE Reference Manual* for details on how to specify an ECU instance on the command-line

It is not possible to create RTE phase outputs for multiple ECU instances in a single execution of RTA-RTE's RTE generator.

4.2.1 Inputs

When invoked for RTE phase generation, RTA-RTE expects the following input files:

- The SWC descriptions according to the AUTOSAR Software-Component Template for all SWC types used within the input.

  This input defines the software components, their ports, internal behaviour and implementation characteristics and the interfaces provided and required by the ports
- The System description according to the AUTOSAR System Template for the ECU instance for which RTE phase is being generated.

  This input defines things like the network topology, how inter-ECU communication is mapped to the physical network etc.
- The ECU configuration according to the AUTOSAR ECU Description for the ECU instance for which the RTE is being generated.

  This input defines what tasks are present, how runnable entities are mapped to tasks, etc. and the mapping of AUTOSAR signals to COM signals for inter-ECU sender-receiver communication.

Information present in the input but not required by RTA-RTE is ignored. Thus it is not necessary to strip out all non-relevant information when invoking RTA-RTE's RTE generator.

As with Contract phase, the input can be included in one or more XML files each of which can contain one or more AUTOSAR packages. There is no requirement, for example, for all system information to be present in one file and all SWC information to be in another file.

4.2.2 Outputs

When invoked for RTE phase generation, RTA-RTE produces considerably more output than when invoked for Contract phase:

- A C source file, `Rte.c`, containing the generated RTE API and supporting data structures.

- A C header file, `Rte_Type.h`, containing all type definitions within the input files. For any particular SWC the type definitions generated will be the same as those generated at Contract phase.
- One or more application header files, `Rte_<name>.h` where `<name>` is the short name of an SWC type, containing declarations of the RTE API for the SWC type.

  During RTE phase, RTA-RTE has complete information about the mapping of all SWC instances to the ECU instance(s). Therefore, the application header file generated during RTE phase contains additional optimizations over and above those possible during Contract phase. For example, sender-receiver communication may be mapped to direct read/write depending on whether or not runnable entity mapping to tasks means that explicit data consistency are not required.
- A C header file, `Rte_Cbk.h`, containing prototypes for all call-back functions created within the generated RTE.
- A C header file, `Rte_Const.h`, containing RTE configuration constants.

  This file defines constants derived from the configuration that are used to optimize the compilation of the RTE library.

  All RTE library modules must be recompiled whenever the generated file `Rte_Const.h` changes.

  See the *RTA-RTE Reference Manual* for details on the generated configurations constants and for information on the RTE library.

The following output files are only created when RTA-RTE is operating in “vendor” mode:

- A C source file, `<Taskname>.c`, containing the generated task body for each task containing runnable entities. Note that in compatibility mode the task bodies are created within the generated RTE file itself.

The following files are optional – whether or not they are generated depends on the contents of your distribution of RTA-RTE.

- `<OS Name>.<suffix>`, an OS configuration file for the AUTOSAR/OSEK Operating System <OS Name>. The file name will have an appropriate extension.
- `<COM Name>.oil`, an OIL configuration file for AUTOSAR/OSEK COM <COM Name>.

4.2.3 Optimization

The RTE generator can modify the generated RTE depending on command-line options.

Atomic Assignment

The `--atomic-assign` option is used to pass target specific knowledge about AUTOSAR platform types to the RTE generator. The option takes as a parameter a comma-separated list of platform types (`uint8`, `uint16`, etc.), e.g.:

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

The option indicates that assignment of the specified types is atomic and therefore no special mechanisms are required to be inserted into generated code by the RTE generator.

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`

4.3 Compile-time

The output of RTA-RTE is standard ANSI C and therefore RTA-RTE requires no command-line options to adapt it for a particular compiler toolchain. However, when compiling code generated by RTA-RTE the following dependencies should be observed.

4.3.1 C Library

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

However when `RTE_LIBC_MEMCPY` is defined when compiling the RTE library and generated code then RTA-RTE will use the C library’s `memcpy` function.

4.3.2 Include Path

The RTA-RTE distribution includes a number of include folders that must be present on the compiler’s include path when compiling generated code:

- `{install-fldr}\Inc` – Core RTA-RTE include files including Rte.h.
- `{install-fldr}\External\Inc` – Non-target specific AUTOSAR include files, for example `Os.h` and `Std_Types.h`.

  The AUTOSAR header files can be used with any operating system, compiler and microcontroller combinations.

- {install-fldr}\External\Inc\TGT – Target specific AUTOSAR include files including Compiler.h. and Compiler_Cfg.h. These files adapt the generated RTE to specific operating system, compiler and microcontroller combinations.

Your distribution of RTA-RTE includes exemplar target specific files. For additional files please contact either AUTOSAR or ETAS GmbH.

The recommended order on the include path is Inc then External\Inc and finally External\Inc\TGT. Please consult your compiler documentation on how to set additional include folders.

4.3.3 Pre-include

The generated RTE includes a pre-include mechanism that permits a user-specified header file to be read before any other header files are included.

To enable the pre-include mechanism define RTE_REINCLUDE when compiling Rte.c. For example, to include the file bob.h:

```
-D RTE_PREINCLUDE="\"bob.h\""
```

The defined pre-include header file must be encapsulated in either double quotes or angle brackets.

4.3.4 RTE Initialization Check

The generated RTE includes a check to ensure that any inter-ECU communication that occurs before the RTE is initialized is ignored. This check can be disabled if, for example, it is known that no such communication can occur, by defining RTE_OMIT_UNINIT_CHECK when compiling Rte.c.

If RTE_OMIT_UNINIT_CHECK is not defined when the RTE is compiled the check for an un-initialized RTE occurs within each COM call-back.

4.3.5 OS Environment

RTA-RTE includes support for multiple operating systems. Different selections of OS modify the set of OS objects created, the OS API used within the RTE library and, if applicable, the form of the generated OS configuration.

The OS support assumes the existence of compile-time definitions that define the OS environment on which the generated code will be run. For example, the OS support may generate APIs compliant to the AUTOSAR OS standard but the compiled code still needs to know which OS implementation is providing the execution environment in order to include implementation specific header files.

The OS environment used must be defined on the command-line. Full details of OS integration, and the OS environments supported by this release of RTA-RTE, are contained in Chapter 6.

4.4 Link-time

When linking an application containing RTA-RTE generated code the following dependencies should be observed.

4.4.1 RTE Library

RTA-RTE uses a library that must be compiled and linked with the generated code and the application code to form the final executable.

The RTE library, `Rte_Lib.c`, is generated based on template files. The location of the generated file can be controlled using the `--output` command-line option.

4.4.2 AUTOSAR modules

RTA-RTE uses the AUTOSAR COM and OS modules. Therefore these modules must be built and linked with the application.

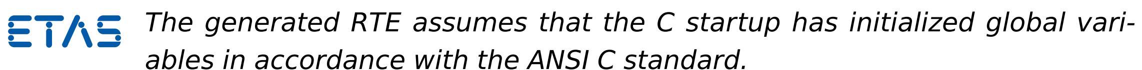
The interaction of RTA-RTE generated code with COM can be influenced with the following command-line options:

- `--com-symbolic-sigs` – When specified, RTA-RTE generated code references COM signals by their name rather than their handle id. The R2.0 backend also supports the deprecated `--symbolic-sigs` command-line option.
- `--com-version` – Specifies the COM API in use. The option takes a single parameter that specifies the COM version, e.g. `-cv=2.0`.

4.5 Run-time

4.5.1 Startup

The `Rte_Start` API must be used to start the RTE. The function ensures that runnable entities triggered by “Timing” RTEEvents are started and activates those triggered by “ModeSwitch” RTEEvents connected to the ENTRY of the initial mode.



The generated RTE assumes that the C startup has initialized global variables in accordance with the ANSI C standard.

4.6 Templates

RTA-RTE uses templates to define the RTE library and associated header files. The template files are supplied with RTA-RTE as plain text and can, if required, be modified to conform to specific user requirements.

4.6.1 Template Scripting Language

RTA-RTE templates use a markup language ‘embedded’ into templates which are then parsed by RTA-RTE, for example:

```
% $outputfile='Rte_Lib.c'
/** @file         <%= $outputfile %> */
```

```
#include "Rte.h"

% if $rbglobals['RTE_IMODEINIT'] > 0
extern FUNC(void, RTE_CODE) Rte_IModeInit(void);
% end
```

This shows a fragment of a template that could represent the `Rte_Lib.c` output file.

Points to note:

- The tag `/*%` signals the start of a section of template script. Matching tag `%*/` signals the end of that section.
- The tag `//%` at the start of a line signals that the entire line is a section of template script.
- Text that is not in a script section is simply passed to the output.
- Text that is in a script section gets executed in the sequence that it appears.
- A start tag `//%#` is treated as a comment. No script code gets executed.

4.6.2 Template Processing

All templates are located within a single folder and are processed after the RTE has been generated. The default location for templates is defined within the RTE's INI file but this can be overridden using the `--template-path` command-line option.

A template file must

- Begin with `Rte`
- End with `_*`, which is converted into the extension of the output file. For example `_c` will be converted to `.c`
- Have the extension `.template`.

A template file with a name that matches `Rte*_h.template` defines a generated header file that is processed in both RTE and contract phases.

The order in which template files are processed is not defined.

4.6.3 Template Global Definitions

The template parsing code that executes in RTA-RTE templates runs in the context of the RTE generator and thus certain global definitions are available for use within the template, e.g.:

```
//% if ( RTE_CALPRM_INITRAM == 0)
```

The following definition parameters are defined within RTA-RTE:

Parameter Name	Type	Notes
RTE_WOWP_EVENTS	Integer	Number of “wake up on wait point” events defined within the generated RTE. This value is used within the Rte_Lib.c template to enable/disable WOWP event based processing.
RTE_MSITABLE_SIZE	Integer	Number of entries in the “minimum start interval” table. This value is used within the Rte_Lib.c template to enable/disable MSI based processing.
RTE_NUM_ALARMS	Integer	Number of alarms. This value is used within the Rte_Lib.c template to enable/disable alarm processing.
RTE_NULL_SCHEDULE	Integer	Non-zero if RTA-RTE has **not** generate a schedule table. This value is used within the Rte_Lib.c template to enable/disable schedule table processing.
RTE_OS_EVENTS	Integer	Number of OS events used to activate runnable entities. This value is used within the RTE library template to enable/disable code that manipulates OS events.
RTE_CALPRM_INITRAM	Integer	Non-zero if calibration uses the “init-ram” method.
RTE_CALPRM_INITRAM_DEFAULTS	Integer	Non-zero if the RTE library, Rte_Lib.c, must initialize Rte_CalprmInitRAM from Rte_CalprmInitROM during Rte_Start. This value is used within the RTE library template to include/exclude code that performs the initialization.

RTE_IMODEINIT	Integer	Non-zero if RTA-RTE has created the Rte_IModeInit function. This value is used within the Rte_Lib.c template to enable/disable code that uses the function.
RTE_USE_GETCURRENTTASKALARM	Integer	Non-zero if the Rte_Lib.c must include the Rte_GetCurrentTaskAlarm function.
RTE_COMMS_ERROR	String	RTE error code for a communications error. This value is used within templates to provide different definitions based on the AUTOSAR release.

The following definition parameters are defined within RTA-RTE as a result of command-line options:

Parameter Name	Type	Notes
OPT_OS_TASK_AS_FUNCTION	Integer	Contains the value of the --os-task-as-function command-line option. If the option is not specified this definition parameter is defined as zero.

5 Legacy Systems

RTA-RTE includes support for the integration of generated RTE code into a legacy system that has a fixed set of tasks to which RTE tasks cannot be added.

5.1 Resource Reuse

AUTOSAR manages mutual exclusive access within a SW-C via exclusive areas declared within the SW-C type. A different implementation strategy can be declared for each exclusive area and for those areas using the “OsResource” implementation strategy RTA-RTE ensures that each SW-C instance accesses a different OS resource.

In normal usage RTA-RTE creates resources as required in the generated OS configuration file using the following name convention:

```
Rte_<type>_<swci>_<ea>
```

Where `<type>` indicates whether the OS resource is “standard” or “internal”, `<swci>` is an RTA-RTE identifier for the SW-C instance and `<ea>` is the exclusive area name.

RTA-RTE also supports an extension to the AUTOSAR standard that permits a predefined OS resource to be used instead of the RTA-RTE created resource.

The specification of which resource to use for an exclusive area instance is RTA-RTE specific.

The AUTOSAR `ExclusiveAreaImplementation` container in the ECU configuration has been extended with an optional reference to the OS resource to use.

```
<REFERENCE-VALUE>
  <DEFINITION-REF DEST='REFERENCE-PARAM-DEF'>/RTARTE/Rte/
      SwComponentInstance/ExclusiveAreaImplementation/
      ExclusiveAreaOsResourceRef</DEFINITION-REF>
  <VALUE-REF DEST='PARAM-CONF-CONTAINER-DEF'>/pkg/Os/resource</VALUE-REF>
</REFERENCE-VALUE>
```

The declaration of the referenced OS resource must be consistent with the expectations of RTA-RTE – in particular whether the resource is “standard” or “internal”. In the event of a mismatch, RTA-RTE will produce a warning and the specified resource will be ignored.

RTA-RTE ensures that all generated tasks are declared as accessing the specified resource however the resource itself is not redeclared within the generated OS configuration file.

5.2 Fixed Task Set

The option `--os-task-as-function` is used to enable and disable generation of task bodies as functions. For example, the option `--os-task-as-function=1` causes RTA-RTE to emit generated task bodies as functions rather than using the OS `TASK()` macro. The generated functions can then be invoked from the existing task bodies as required. The name of the generated task body function is `Rte_task_<name>` where `<name>` is

the task name.

When building tasks as functions the tasks must still be declared in the input so that they are visible to RTA-RTE. The pseudo-tasks must be specified with the same priority as the legacy tasks from which they will be invoked.

For example, consider a legacy system that contains 10ms, 50ms and 100ms tasks and a runnable entity `re1` triggered by a timing event with period 50ms and mapped to task `taskA`. When the task-as-function option is not used RTA-RTE defines the task body using the following definition:

```
TASK(task)
{
  ...
```

However when `--os-task-as-function=1` is specified, the task body definition is altered to:

```
FUNC(void, RTE_CODE)
Rte_task_taskA()
{
  ...
```

The generated function can then be invoked from whatever legacy task is appropriate – in this case the 50ms task.

When generating tasks as functions the mapping of runnable entities to tasks controls the number of generated functions. It is thus possible to define multiple "task functions" to arbitrarily split the execution of runnable entities to fit the requirements of the legacy system.

5.2.1 Interaction with RTE Mechanisms

The use of task bodies as functions is incompatible with the following RTE mechanisms:

Minimum Start Intervals – When a runnable entity specifies a minimum start interval the RTE must be able to control runnable invocation at run-time to permit it to "hold-off" execution until the minimum start interval has expired.

6 Operating System

6.1 Overview

6.1.1 OS Selection on the command line

RTA-RTE supports several embedded operating systems, selected with the command-line option `--operating-system=`*parameter*, where *parameter* is one of:

- `autosar40`
- `autosar30`
- `autosar10`
- `osek223`

If not supplied, the default is `autosar40` for AUTOSAR 4.x projects and `autosar30` for AUTOSAR 3.x projects.

The choice of OS influences how RTA-RTE generates OS API calls, can limit the available RTE features (for example, the availablility of Category 2 Runnables), and writes some constants the `Rte_Const.h` to enable RTE library code to be adapted at compile time.

- `RTE_OSAPI_`*xxx* defines the OS API used, e.g. OSEK or AUTOSAR 4.0
- `RTE_OSCFG_`*xxx* defines the format of the created Os Needs file if present.

6.1.2 Os Implementation preprocessor definition

If the specific OS implementation is known to RTA-RTE, the generated code may be able to take advantage of certain implementation details that cannot be relied upon to be available in all implementations of the standard. RTA-RTE protects such implementation-specific code with a preprocessor conditional based on the following symbols:

- `OSENV_RTAOS40` indicates that ETAS RTA-OS is being used as the execution environment with the AUTOSAR 4.0 API. Note that the 40 refers to the AUTOSAR revision, not the version of RTA-OS. This symbol is supported for AUTOSAR 4.0 projects.
- `OSENV_RTAOS30` indicates that ETAS RTA-OS is being used as the execution environment with an AUTOSAR 3.x API. This symbol is supported for AUTOSAR 3.x and AUTOSAT 4.0 projects.
- `OSENV_RTAOSEK` indicates that RTA-OSEK 5.0 is being used as the execution environment. This symbol is suported in projects with `--operating-system=osek223`. RTA-OSEK 5.0 supports the AUTOSAR 1.0 OS API.

- `OSENV_UNSUPPORTED` indicates that the used AUTOSAR Operating System is not known to RTA-RTE. This symbol is supported by all RTA-RTE projects and avoids code that takes advantage of any implemntation-specific knowledge. RTA-RTE will emit code conforming with the OS API specified by the `--operating-system` option.

It is mandatory to define exactly one of the above symbols at compile time whenever RTE-generated header files are involved in the compilation unit. The appropriate `OSENV` definition can either be supplied on the compiler command-line when compiling the generated RTE, or specificed via the `--os-define-osenv` option to RTA-RTE when generating the RTE. In this last case, RTA-RTE will write the required definition to `Rte_Const.h`.

6.2 AUTOSAR R4.0 OS support

RTA-RTE can interact with a standards-conformant AUTOSAR 4.0 Operating System and has been tested with ETAS RTA-OS. This section explains how the RTE and OS generation tools can interact and how to integrate the generated C code.

6.2.1 OS Needs

RTA-RTE implements the AUTOSAR concept of OsNeeds, that is, given an *upstream OS configuration* and an RTE configuration, RTA-RTE will generate a file `osNeeds.arxml` containing information that completes the OS configuration.

At the time of writing, for SC3 and SC4 configurations, `osNeeds.arxml` does not by itself complete the configuration. It is necesssary to manually add in the ownership and access rights of the OsApplications on the various Os objects in OsNeeds.

The option `--os-file` can be used to specify a different name for the osNeeds file.

The default namespace URI used in `osNeeds.arxml` is `http://autosar.org/schema/r4.0`. The option `--os-xml-namespace` can be used to set a different namespace URI.

The command-line options `--os-file` and `os-xml-namespace` only modify the filename or namespace declaration and do not affect the contents of the XML.

Merge RTE generated and Input XML

By default, or when `--os-output-param=changed`, RTA-RTE does not write task parameters (i.e. priority, activation limit and schedule) to the generated `osNeeds.arxml` if they are present and acceptible in the upstream OS Configuration. This makes it possible to simply provide the generated `osNeeds.arxml` to the OS Generator along with the upstream OS configuration to generate the OS.

This method is illustrated in Figure 6.1.

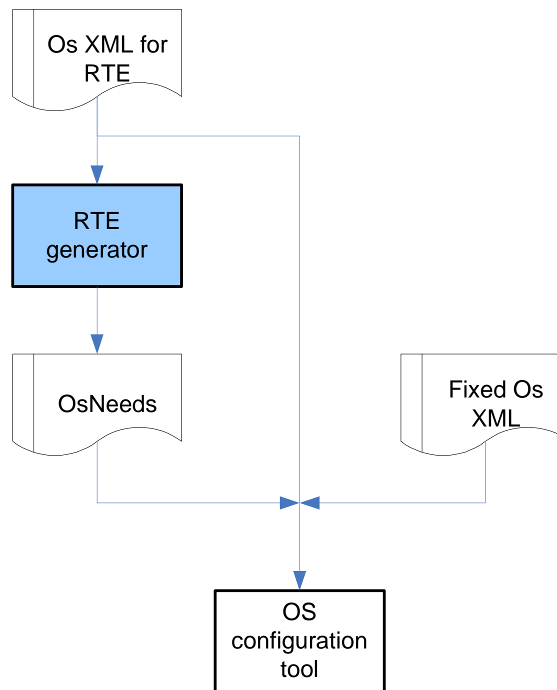


Figure 6.1: Interaction of RTA-RTE and Os configuration files when using option `--os-output-param=changed`.

Replace Input with RTE-generated XML

When `--os-output-param=all` is supplied then RTA-RTE outputs all task parameters (i.e. priority, activation limit and schedule) and the RTE resource declaration in the generated `osNeeds.arxml` irrespective of whether or not RTE generation has caused them to change.

In this mode it is likely that there will be duplicate or conflicting information between the input Os configuration and the generated `osNeeds.arxml`. It is unlikely that the two files can be merged by the OS Generator in this case.

This method is illustrated in Figure 6.2.

6.2.2 AUTOSAR R4.0 OS Interaction

AUTOSAR R4.0 introduced mechanisms to associate all RTE objects, such as timing events and exclusive areas, with existing OS objects manually. When this is done then the generated `osNeeds.arxml` can be discarded, as illustrated in Figure 6.3.

When you use R4.0 OS Interaction mechanisms it is strongly recommended to use command-line option `--strict-config-check=true` to ensure that all required OS objects are present in the input.

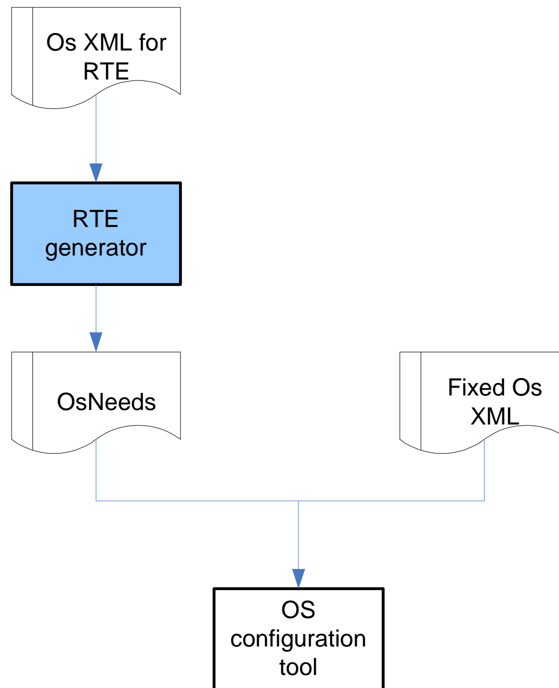


Figure 6.2: Interaction of RTA-RTE and Os configuration files when using option `--os-output-param=all`.

6.2.3 Generated C

When RTA-RTE generates an RTE expecting to use an AUTOSAR 4.0 OS, it writes `#define`s for the following symbols to `Rte_Const.h`:

Symbol	Meaning
`RTE_OSAPI_AUTOSAR_R40`	The API used by the generated RTE is compatible with AUTOSAR OS R4.0. A schedule table is used for runnable entities activated by time.
`RTE_OSCFG_AUTOSAR_R40`	The generated OS configuration fragment uses AUTOSAR R4.0 XML.

6.2.4 ETAS RTA–OS v5.x and v4.0

RTA-OS 4.0 and later are capable of generating an AUTOSAR 4.0 conformant API for use with RTA-RTE. The OS environment macro `OSENV_RTAOS40` must be defined when RTE is used with ETAS RTA-OS, for example:

```
cc --DOSENV_RTAOS40 Rte.c
```

The 40 in `OSENV_RTAOS40` relates to AUTOSAR 4.0, not the product version of RTA-OS.

6.2.5 Other Operating Systems

To use RTA-RTE with another AUTOSAR 4.0 conformant OS product, define the macro `OSENV_UNSUPPORTED` to ensure that no implementation-spefic details are used, for ex-

Definition	Notes
RTE_OSAPI_AUTOSAR_R30	The API used by the generated RTE is compatible with AUTOSAR OS R3.0. A schedule table is used for runnable entities activated by time.
RTE_OSCFG_AUTOSAR_R30	The generated OS configuration fragment uses AUTOSAR R3.0 XML.

6.3.1 RTA–OS

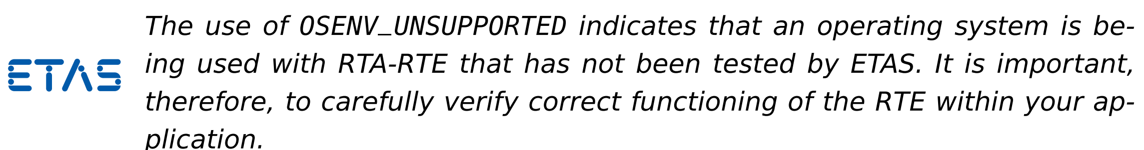
The OS environment `OSENV_RTAOS30` must be defined when RTA-OS is used with the AUTOSAR R3.0 OS support. For example:

```
cc --DOSENV_RTAOS30 Rte.c
```

6.3.2 Other Operating Systems

Operating systems other than RTA-OS can be used with the AUTOSAR R3.0 OS support if the `OSENV_UNSUPPORTED` environment is declared. For example:

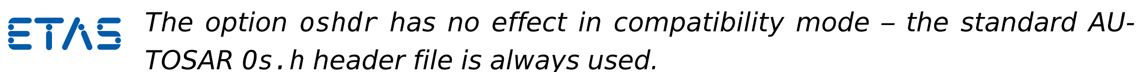
```
cc --DOSENV_UNSUPPORTED Rte.c
```



The use of `OSENV_UNSUPPORTED` indicates that an operating system is being used with RTA-RTE that has not been tested by ETAS. It is important, therefore, to carefully verify correct functioning of the RTE within your application.

6.3.3 OS Header File

When operating in vendor mode, the AUTOSAR R3.0 OS support includes an option, `oshdr`, to define the OS header file used within the generated RTE and the generated task bodies.



The option `oshdr` has no effect in compatibility mode – the standard AUTOSAR `Os.h` header file is always used.

As an example, the option `--oshdr=name` causes RTA-RTE to use `#include` <`name`> when including the OS Header file in generated code instead of the default `#include` <`Os.h`>.

6.3.4 OS Configuration File

By default the AUTOSAR R3.0 OS support creates an OS configuration file fragment name `osNeeds.arxml`.

If the AUTOSAR R3.0 OS support is used with other operating systems, the `osfile` option can be used to rename the generated XML file fragment.

The option `osfile` only renames the generated file and has no effect on its contents.

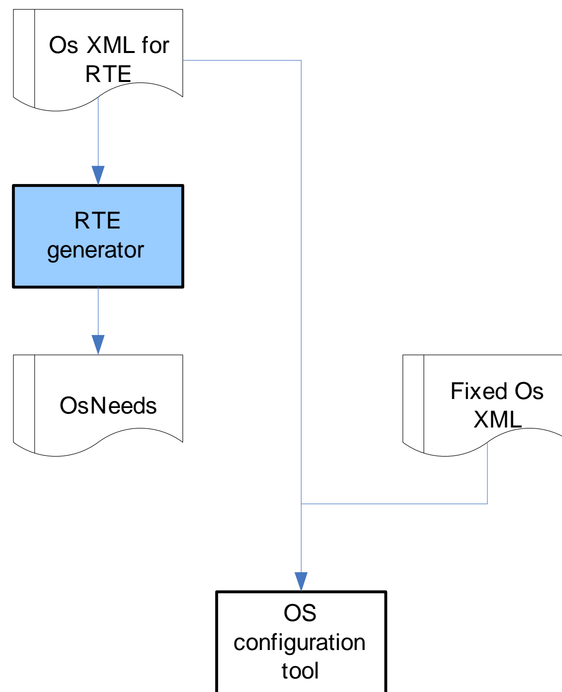


Figure 6.3: Interaction of RTA-RTE and Os configuration files when using R4.0 OS Interaction.

The default namespace URI used within `osNeeds.arxml` is `http://autosar.org/3.0.2`. The option `--os-xml-namespace` can be used to set a different namespace URI, for example:

```
RTEGen --os-xml-namespace=http://autosar.org/3.1.0 ...
```

The option `osxmlns` only modifies the namespace declaration within the generated OS configuration file and has no effect on its contents.

6.3.5 OS Parameters

The `--os-output-param` option supports two methods of working with the `osNeeds.arxml` created by RTA-RTE where it is used either instead of, or merged with, the input XML when generating the OS.

Merge

When the `--os-output-param` command-line option is used with the parameter changed then RTA-RTE does not redefine task parameters (i.e. priority, activation limit and schedule) within the generated `osNeeds.arxml` unless RTE generation causes them to change. This means that the input Os configuration and the generated `osNeeds.arxml` can be merged with other Os configuration files when passed to the OS generator.

This method is illustrated in Figure 6.4.

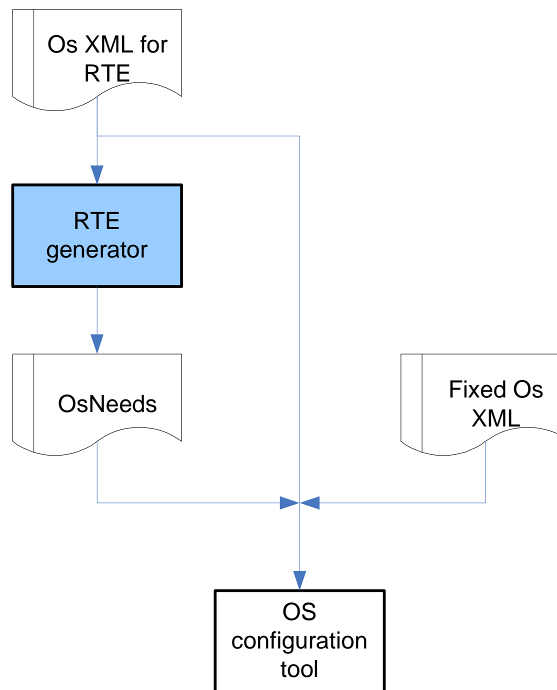


Figure 6.4: Interaction of RTA-RTE and Os configuration files when using option `-osparam=changed`.

Replacement

When the `--os-output-param` command-line option is used with the parameter `all` then RTA-RTE outputs all task parameters (i.e. priority, activation limit and schedule) and the RTE resource declaration within the generated `osNeeds.arxml` irrespective of whether or not RTE generation has caused them to change.

In this mode the input Os configuration and the generated `osNeeds.arxml` cannot be merged with other Os configuration files, such as the input Os configuration, when passed to the OS generator.

This method is illustrated in Figure 6.5.

RTA-OS does not permit parameters to be defined in multiple files and therefore if `-osparam=all` is used the input XML file and generated `osNeeds.arxml` file cannot be merged.

6.4 AUTOSAR R1.0 OS support

The AUTOSAR R1.0 OS support has been designed to integrate with RTA–OSEK 5.0 in AUTOSAR SC1 compatibility mode but can also be used with any AUTOSAR R1.0 OS.

Definition	Notes

RTE_OSAPI_AUTOSAR_R10	The API used by the generated RTE is compatible with AUTOSAR OS R1.0. A schedule table is used for runnable entities activated by time.
RTE_OSCFG_RTAOSEK	The generated OS configuration fragment uses AUTOSAR OIL with extensions for RTA–OSEK.

RTA–OSEK 5.0 extensions in the generated OIL will be ignored by third–party OSs.

6.4.1 RTA–OSEK 5.0

The OS environment `OSENV_RTAOSEK` must be defined when RTA-OSEK 5.0 is used with the AUTOSAR R1.0 support. For example:

```
cc --DOSENV_RTAOSEK Rte.c
```

When `OSENV_RTAOSEK` is defined, and RTA-RTE is operating in vendor mode, the RTA-OSEK per-task header files are used to optimize the generated task bodies.

6.4.2 Floating Point

The RTA-RTE AUTOSAR10 OS support includes RTA-OSEK specific OIL extensions to define whether or not tasks (and hence runnable entities invoked by the task) use floating point. The default is to assume that the tasks use floating point but this can be changed with the `osfp` option.

When set to 0 the option `--os-fp` disables floating point usage by generated tasks. Setting the option to 1 enables floating point support.

6.4.3 Other Operating Systems

Operating systems other than RTA-OSEK can be used with the AUTOSAR R1.0 OS support if the `OSENV_UNSUPPORTED` environment is declared, for example:

```
cc --DOSENV_UNSUPPORTED Rte.c
```

The use of `OSENV_UNSUPPORTED` indicates that an operating system is being used with RTA-RTE that has not been tested by ETAS. It is important, therefore, to carefully verify correct functioning of the RTE within your application.

6.4.4 OS Header File

When operating in vendor mode, the AUTOSAR R1.0 support includes an option, `oshdr`, to define the OS header file used within the generated RTE and the generated task bodies.

The option `oshdr` has no effect in compatibility mode – the standard AUTOSAR `Os.h` header file is always used.

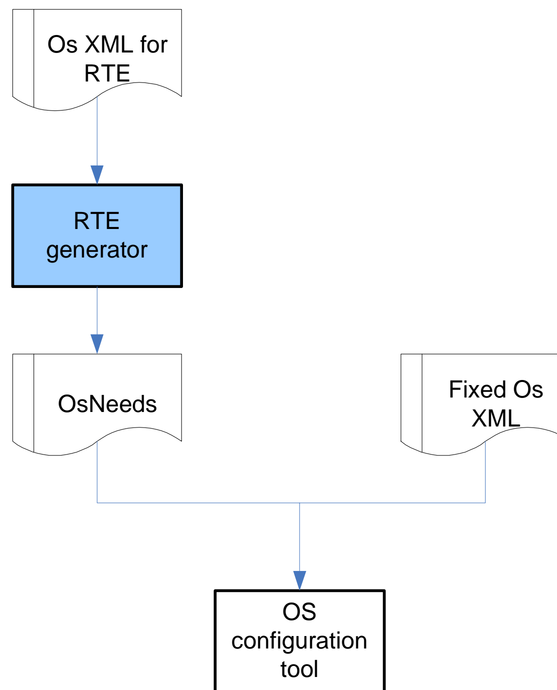


Figure 6.5: Interaction of RTA-RTE and Os configuration files when using option `-osparam=all`.

As an example, the option `--oshdr=name` causes RTA-RTE to use `#include <name>` when including the OS Header file in generated code instead of the default `#include <Os.h>`.

6.4.5 OS Configuration File

By default the AUTOSAR R1.0 OS support creates an OS configuration file fragment name rta-osek.oil.

ETAS *The generated OS configuration fragment includes RTA-OSEK 5.0 specific OIL++ statements to configure RTA-OSEK specific functionality. The OIL++ statements will be ignored by third–party OIL parsers.*

If the AUTOSAR R1.0 OS support is used with other operating systems, the `osfile` option can be used to rename the generated OIL file fragment.

The option `osfile` only renames the generated file and has no effect on its contents. In particular, renaming the generated file does not remove the RTA-OSEK 5.0 OIL++ statements.

As an example, the option `--osfile=os.oil` causes the RTA-RTE AUTOSAR R1.0 OS support to create a file `os.oil` containing the generated OS configuration fragment.

6.5 OSEK 2.2.3 support

The OSEK 2.2.3 OS support has been designed to integrate with a generic OSEK OS such as RTA-OSEK 5.0.

Definition	Notes
RTE_OSAPI_OSEK	The API used by the generated RTE is compatible with OSEK 2.2.3. Cyclic alarms are used for runnable entities activated by time.
RTE_OSCFG_OSEK	The generated OS configuration fragment uses OSEK OIL.

6.5.1 RTA-OSEK 5.0

The OS environment `OSENV_RTAOSEK` must be defined when RTA-OSEK 5.0 is used with the OSEK 2.2.3 support. For example:

```
cc --DOSENV\_RTAOSEK Rte.c
```

When `OSENV_RTAOSEK` is defined, and RTA-RTE is operating in vendor mode, the RTA-OSEK per-task header files are used to optimize the generated task bodies.

6.5.2 Other Operating Systems

Operating systems other than RTA-OSEK can be used with the OSEK 2.2.3 OS support if the OSENV_UNSUPPORTED environment is declared. For example:

```
cc --DOSENV\_UNSUPPORTED Rte.c
```

The use of OSENV_UNSUPPORTED indicates that an operating system is being used with RTA-RTE that has not been tested by ETAS. It is important, therefore, to carefully verify correct functioning of the RTE within your application.

6.5.3 OS Header File

When operating in vendor mode, the OSEK 2.2.3 OS support includes an option, `oshdr`, to define the OS header file used within the generated RTE and the generated task bodies.

The option `oshdr` has no effect in compatibility mode – the standard AUTOSAR `Os.h` header file is always used.

As an example, the option `--oshdr=name` causes RTA-RTE to use `#include` <name> when including the OS Header file in generated code instead of the default `#include` <`Os.h`>.

6.5.4 OS Configuration File

By default the OSEK 2.2.3 support creates an OS configuration file fragment name `osek.oil`.

If the OSEK 2.2.3 support is used with other operating systems, the `osfile` option can be used to rename the generated OIL file fragment.

The option `osfile` only renames the generated file and has no effect on its contents.

As an example, the option `--osfile=os.oil` causes the RTA-RTE OSEK 2.2.3 support to create a file `os.oil` containing the generated OS configuration fragment.

6.6 OS Abstraction Layer

The final element in RTA-RTE support for multiple operating systems is the OS abstraction layer which provides support mechanisms for implementation dependent characteristics of operating systems.

6.6.1 Rationale

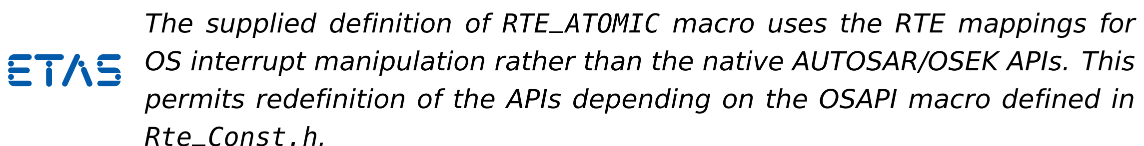
The AUTOSAR operating system and OSEK 2.2.3 from which it is derived define standard types used by OS APIs, e.g. `TaskType`, but do not define the implementation of these types. The RTA-RTE OS abstraction layer provides the necessary mechanisms for RTA-RTE generated code to be adapted to different OS implementations of the fundamental OS types.

The RTA-RTE OS abstraction layer is defined in `Rte_Intl.h`. This file should not be modified directly; instead either the underlying template should be changed or each element of the abstraction mechanism can be overridden using definitions on the command–line when RTA-RTE generated code is compiled.

6.6.2 Atomic Code

RTA-RTE uses the `RTE_ATOMIC` macro to encapsulate short regions (typically single expressions) of atomic code:

```
#define RTE_ATOMIC(op) (Rte_SuspendOSInterrupts(),\
                        (op),                      \
                        Rte_ResumeOSInterrupts() )
```



The supplied definition of RTE_ATOMIC macro uses the RTE mappings for OS interrupt manipulation rather than the native AUTOSAR/OSEK APIs. This permits redefinition of the APIs depending on the OSAPI macro defined in `Rte_Const.h`.

An alternative definition of `RTE_ATOMIC` can be supplied if the above definition is not suitable, for example, a compiler toolchain may offer an intrinsic function that exploits special features of a particular CPU, for example:

```
#define RTE_ATOMIC(op) (_atomic(op))
```

The definition of `RTE_ATOMIC` must satisfy the following constraints:

- Preemption by any OS controlled entity (task or ISR) must be impossible for any code within `RTE_ATOMIC`.
- “op” is an expression that may be comma-separated.

- RTE_ATOMIC must be an expression and not a statement.

6.6.3 TaskType

When generating code, RTA-RTE can support task handles that are implemented either as global variables or as a fixed value such as an integer or the address of an element in an array.

The default mechanism within RTA-RTE assumes that a task is defined as an integer or fixed value suitable for direct use within a C static initializer. When this is the case one might expect to see the following definition within OS configuration code:

```
#define task1 (/* os specific */)
```

The alternative mechanism, that of a TaskType as a global variable, can also be used. In this case one might expect to see the following definition within generated OS configuration code:

```
TaskType task1 = (/* os specific */);
```

The use of a global variable rather than a fixed value is enabled then any of the following are defined:

- OSENV_RTAOSEK
- RTE_TASKTYPE_IS_GLOBALCONST

Whatever the OS mechanism selected for the declaration of task handles RTA-RTE needs to be able to include them within C static initializers. This is achieved through the RTE_TASKREFTYPE definition of which can be modified depending on the selected OS definition of TaskType to perform the correct action.

The default definition of RTE_TASKREFTYPE when task handles are fixed values is straightforward since these values can be included within a static initializer without modification:

```
#define RTE_TASKREFTYPE  TaskType
```

However when a TaskType is a global variable then RTE_TASKREFTYPE is defined as the *address of* the variable since the address can be included within a static initializer:

```
#define RTE_TASKREFTYPE  \
         P2CONST(TaskType,AUTOMATIC,RTE_OS_CDATA)
```

If required, an alternative definition of RTE_TASKREFTYPE can be given on the command-line which will then override the default definition when RTA-RTE generated code is compiled.

RTA-RTE includes macro definitions RTE_TASK_FROM_REF and RTE_REF_FROM_TASK to convert a TaskType to and from a reference type. The definitions of these macros

is modified depending on whether a task handle is a fixed value or a global variable. If required, alternative definitions of RTE_TASK_FROM_REF and RTE_REF_FROM_TASK can be given on the command-line when compiling generated RTE code which will then override the default definition when RTA-RTE generated code is compiled.

Finally, the value RTE_TASK_REF_NO_TASK is defined as a *null task* reference, i.e. a value suitable for use within a static initializer that can never itself be a valid task reference. For example, when a task handle is a global variable then the following definition is used since 0 can never be a valid address of a real variable:

```
#define RTE_TASK_REF_NO_TASK            (0)
```

6.6.4 EventMaskType

When generating code, RTA-RTE can support OS event masks that are implemented either as global variables or as a fixed value such as an integer or the address of an element in an array.

The default mechanism within RTA-RTE assumes that an event mask is defined as a fixed value suitable for direct use within a C static initializer. When this is the case one might expect to see the following definition within generated OS configuration code:

```
#define event1 (/* os specific */)
```

The alternative mechanism, that of a EventMaskType as a global variable, can also be used. In this case one might expect to see the following definition within generated OS configuration code:

```
EventMaskType event1 = (/* os specific */);
```

The use of a global variable rather than a fixed value is enabled then any of the following are defined:

- OSENV_RTAOSEK
- RTE_EVENTTYPE_IS_GLOBALCONST

Whatever the OS mechanism selected for the declaration of event masks RTA-RTE needs to be able to include them within C static initializers. This is achieved through the RTE_EVENTREFTYPE definition of which can be modified depending on the selected OS definition of EventMaskType to perform the correct action.

The default definition of RTE_EVENTREFTYPE when event masks are fixed values is straightforward since these values can be included within a static initializer without modification:

```
#define RTE_EVENTREFTYPE  EventMaskType
```

However when a EventMaskType is a global variable then RTE_EVENTREFTYPE is defined as the *address of* the variable since the address can be included within a static initializer:

```
#define RTE_EVENTREFTYPE  \
         P2CONST(EventMaskType,AUTOMATIC,RTE_OS_CDATA)
```

If required, an alternative definition of RTE_EVENTREFTYPE can be given on the command-line which will then override the default definition when RTA-RTE generated code is compiled.

RTA-RTE includes macro definitions RTE_EVENT_FROM_REF and RTE_REF_FROM_EVENT to convert a EventMaskType to and from a reference type. The definitions of these macros is modified depending on whether an event mask is a fixed value or a global variable. If required, alternative definitions can be given on the command-line when compiling generated RTE code which will then override the default definition when RTA-RTE generated code is compiled.

6.6.5 AlarmType

When generating code, RTA-RTE can support Os alarms where AlarmType is implemented either as global variables or as a fixed value such as an integer or the address of an element in an array.

The default mechanism within RTA-RTE assumes that an alarm is defined as a fixed value suitable for direct use within a C static initializer. When this is the case one might expect to see the following definition within generated OS configuration code:

```
#define alarm1 (/* os specific */)
```

The alternative mechanism, that of a AlarmType as a global variable, can also be used. In this case one might expect to see the following definition within generated OS configuration code:

```
AlarmType alarm1 = (/* os specific */);
```

The use of a global variable rather than a fixed value is enabled then any of the following are defined:

- OSENV_RTAOSEK
- RTE_ALARMTYPE_IS_GLOBALCONST

Whatever the OS mechanism selected for the declaration of alarms RTA-RTE needs to be able to include them within C static initializers. This is achieved through the RTE_ALARMREFTYPE definition of which can be modified depending on the selected OS definition of AlarmType to perform the correct action. If required, an alternative definition of RTE_ALARMREFTYPE can be given on the command-line which will then override the default definition when RTA-RTE generated code is compiled.

RTA-RTE includes macro definitions RTE_ALARM_FROM_REF and RTE_REF_FROM_ALARM to convert an AlarmType to and from a reference type. The definitions of these macros is modified depending on whether an event mask is a fixed value or a global variable.

If required, alternative definitions can be given on the command-line when compiling generated RTE code which will then override the default definition when RTA-RTE generated code is compiled.

Finally, the value RTE_NULL_ALARM_REF is defined as a *null alarm* reference, i.e. a value suitable for use within a static initializer that can never itself be a valid alarm reference. For example, when a alarm handle is a global variable then the following definition is used since 0 can never be a valid address of a real variable:

```
#define RTE_NULL_ALARM_REF           (0)
```

If required, an alternative definition of RTE_NULL_ALARM_REF can be given on the command-line when compiling generated RTE code which will then override the default definition when RTA-RTE generated code is compiled.

6.6.6 ResourceType

When generating code, RTA-RTE can support Os alarms where ResourceType is implemented either as global variables or as a fixed value such as an integer or the address of an element in an array.

- OSENV_RTAOSEK
- RTE_RESOURCETYPE_IS_GLOBALCONST

When either of the above definitions is defined, RTA-RTE assumes that ResourceType is a global constant and defines RTE_RESOURCE_FROM_REF and RTE_REF_FROM_RESOURCE appropriately. If required, alternative definitions can be given on the command–line when compiling generated RTE code which will then override the default definition when RTA-RTE generated code is compiled.

7 Build Environment

The RTA-RTE generation tool is designed to be integrated into a third-party development environment, such as Eclipse, provided some simple requirements are met.

The primary requirement necessary to integrate RTA-RTE is that the build environment must be capable of invoking an external Win32 executable at the required point in the build process. When this requirement is met RTA-RTE's error formatting can, if required, be modified to enable the invoking build tool to detect and correctly locate errors in the input files passed to RTA-RTE.

7.1 Option sub-files

To overcome limitations of build environments related to the length of the command-line RTA-RTE includes the `--file` option[1] to read additional command-line options from a file.

See the *RTA-RTE Reference Manual* for more details on the `-f` option.

The option can be used recursively; a file read using the `-f` option can include other files if required.

7.2 Make

RTA-RTE is designed to be easy to integrate into make (or make-like) build environments.

7.2.1 Return Code

RTA-RTE's RTE generator executable, `RTEGen.exe`, sets the error code depending on whether or not RTE generation was successful:

- 0 : Success – the RTE Types and application headers (RTE and Contract phase) and/or other files (RTE phase only) were generated without detected error.

  The "success" return does not consider warnings as errors.
- Non-zero : Failure – at least one error was reported.

7.2.2 Build Dependency Information

The `--makedep` command-line option causes RTA-RTE to emit dependency information for generated files to the specified file. The option takes one parameter; the name of the file to which dependency information should be written.

The generated dependency information is designed to be directly included within Makefiles and therefore created as a set of rules each of which has the form:

```
<target>:       <dependencies>
```

[1] The R2.0 backend also supports the `--subfile` command-line option.

The <`target`> of the rule defines the file for which the dependencies are defined. The emitted information always contains exactly one <`target`> per rule.

The <`target`> may be either a C header file or an object file. To maintain compiler independence the <`target`> uses a reference to the make variable `$(OBJ)` rather than assuming a specific object file suffix.

The <`dependencies`> list contains all files upon which the <`target`> depends; if any dependency is modified then the target must be rebuilt.

As an example, the following emitted rule defines the dependencies for the object file created by compiling Rte.c:

```
Rte.$(OBJ):     Com.h Os.h Rte.h Rte_Cfg.h
                Rte_Const.h Rte_Intl.h
                Rte_Main.h Rte_Type.h Rte.c
```

7.3 Incremental Build

By default, RTA-RTE creates all output files. However when incremental file generation is enabled using the `--incremental-build` command-line option then previously generated output files are only modified if their contents has changed.

Incremental build examines the previously generated before deciding whether it should be overwritten with the newly generated file. Thus when incremental build is enabled the location of a previously generated file and the corresponding newly created file must be the same.

When incremental build is enabled the generation of timestamps in output files is disabled. Also when enabled the list of generated files includes an annotation, `[skipped]` or `[updated]`, that indicates whether or not incremental build overwritten an existing file.

Incremental build does not affect the dependency information created by the `--makedep` command-line option.

The error output file created by the `--error-report` command-line option is not subject to incremental build – it is always created irrespective of the setting of the `--incremental-build` option.

7.4 Error reporting

Uniquely raised fatal, error, warning and information messages are collected by the RTA-RTE core and passed to the selected error reporter which is then responsible for:

- Formatting errors for output.
- Error display.

The RTA-RTE distribution includes three selectable error reporters;

- “file” writes all errors to the file `Rte.err`.
- “console” writes errors to the system error stream.
- “xml” writes errors as XML to the file `RteErr.xml`.

The `--error-report` option is used to select the required error reporter. If the option is omitted the “console” error reporter is used as the default.

7.4.1 XML Output

The “xml” error reporter included with RTA-RTE writes error messages as XML to the file `RteErr.xml`.

The XML root node is <RTEGEN> and all error messages are written within the sub-node <MESSAGES> in the order they were raised. For example:

```
<RTEGEN>
  <VERSION-INFO>
    <RTEGen></RTEGen>
  </VERSION-INFO>
  <MESSAGES>
    <MESSAGE>
      ...
    </MESSAGE>
    <COUNT>
      <ERROR>0</ERROR>
      <WARNING>1</WARNING>
      <INFO>0</INFO>
    </COUNT>
  </MESSAGES>
</RTEGEN>
```

Each <MESSAGE> element includes five sub-nodes:

- <CODE> – The message code including the error class (`F`, `E`, `W` or `I`), the two-digit container identifier and the four digit message identifier.
- <FILE> – File containing the error (if appropriate, if no file is relevant this element will be empty).
- <LINE> – Line within the file (if appropriate, if no file is relevant this element will be empty).
- <TEXT> – Message text.
- <LOCATION> – RTE internal location where error message was generated (if not available this element will be empty).

For example:

```
<MESSAGE>
  <CODE>W14-1205</CODE>
  <FILE>swcA.arxml</FILE>
  <LINE>8</LINE>
  <TEXT>Generation warning -- ...</TEXT>
  <LOCATION/>
</MESSAGE>
```

The <MESSAGES> element also includes a <COUNT> element that defines the total number of error, warning and information messages.

8 Contact, Support and Problem Reporting

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries	`www.etas.com/en/contact.php`
ETAS technical support	`www.etas.com/en/hotlines.php`

The RTA hotline is available to all RTA-RTE users with a valid support contract.

rta.hotline.uk@etas.com
+44 (0)1904 562624. (0900-1730 GMT/BST)

Please provide support with the following information:

- Your support contract number.
- Your AUTOSAR XML and/or OS configuration files.
- The command line that results in an error message.
- The version of the ETAS tools you are using.

Index

Symbols

OSENV_macros, 34–37
OSENV_ macros, 26, 27, 30, 32
--atomic-assign, 15
--com-symbolic-sigs, 17
--com-version, 17
--contract, 12
--error-report, 39, 40
--file, 38
--incremental-build, 39
--makedep, 38, 39
--memory-sections, 11
--operating-system, 23, 24
--os-define-osenv, 24
--os-file, 24
--os-fp, 30
--os-output-param, 24–26, 28, 29
--os-task-as-function, 20, 21
--os-xml-namespace, 24, 28
--oshdr, 26
--output, 12, 17
--rte-phase, 13
--strict-config-check, 25
--subfile, 38
--symbolic-sigs, 17
--template-path, 18
--warn-directive, 6

A
Application header file, 12, 14
Atomic
    Assignment, 15
    Code, 33

B
Build Dependency Information, 38

C
C Library, 8, 15
C warnings, 6
C-Startup, 17
Com.h, 7
Compiler.h, 7
Compiler_Cfg.h, 7
Contract Phase, 12
    Inputs, 12
    Outputs, 12

Conventions, 4

E
Error reporting, 39

I
Include path, 15
Incremental Build, 39
Initializtion check, 16

L
Legacy Systems, 21

M
Make tool, 38
memcpy, 8, 15
MemMap.h, *see* Rte_MemMap.h
Memory Section Description File, 9

O
OS Abstraction, 33
  AlarmType, 36
  Atomic code, 33
  EventMaskType, 35
  ResourceType, 37
  TaskType, 34
OS environment, 16
Os.h, 7
OSEK 2.2.3, 31
  Configuration, 32
  Header file, 32

P
Phase
  Contract, *see* Contract Phase
  RTE, *see* RTE Phase
Platform_Types.h, 7

R
Resource reuse, 21
Return code, 38
RTA-OS 4.0, 24
  Configuration, 24
  Header file, 26
RTA-OSEK 5.0, 29
  Configuration, 31
  Floating point, 30
  Header file, 30

RTE Phase, 13
  Inputs, 13
  Outputs, 13
RTE-OS3.0, 26
  Configuration, 27
  Header file, 27
  Parameters, 28
Rte.c, 13
RTE_ATOMIC, 33
Rte_Cbk.h, 14
Rte_Const.h, 14
RTE_LIBC_MEMCPY, 15
Rte_memcpy, 8, 15
Rte_MemMap.h, 7
RTE_OMIT_UNINIT_CHECK, 16
Rte_Type.h, 12, 14

S
Std_Types.h, 7

T
Target
  Task17x6, 7
  TriCoreHighText, 7
  Virtual PC, 7, 8
Task-as-function option, 21
Templates, 17
  Definitions, 18
  Folder, 18
  Language, 17

V
Vendor mode, 14
vendor mode, 26