Elektrobit

EB tresos® AutoCore Generic RTE safety application guide

for ASIL-B applications

Date: 2021-03-11, Document version 1.1, Status: DRAFT

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

https://www.elektrobit.com/support

Legal disclaimer

Table of Contents

1. Document history

Version	Date	Author	State	Description
0.1	2018-03-01	Elektrobit Automotive GmbH	DRAFT	Initial version.
1.0	2018-06-27	Elektrobit Automotive GmbH	RELEASED	▶ Prepared document. ▶ Incorporated review findings. ▶ Set document state to released.
1.1	2021-03-11	Elektrobit Automotive GmbH	DRAFT	Reworked verification and review criteria chapter.

Table 1.1. Document history

2. Document information

2.1. Objective

The objective of this document is to provide you with all the information necessary to ensure that *EB tresos AutoCore Generic RTE* is used in a safe way in your project.

2.2. Scope and audience

This guide describes the use of EB tresos AutoCore Generic RTE in *system applications* which have safety allocations. It is valid for all projects and organizations which use EB tresos AutoCore Generic RTE in a safety-related environment.

The intended audience of this guide consists of:

- software architects
- safety engineers
- application developers
- software *integrators*

The persons with these roles are responsible for performing the verification methods defined in this guide.

They should at least have the following knowledge and abilities:

- C-programming skills
- experience in programming AUTOSAR-compliant ECUs

Furthermore, advanced knowledge of the following topics is recommended:

- AUTOSAR *RTE*
- AUTOSAR meta-model
- experience with safety standards
- experience with software development in safety-related environments

2.3. Motivation

This guide provides information on how to correctly use EB tresos AutoCore Generic RTE in projects for safety-related environments. If EB tresos AutoCore Generic RTE is used differently, the generated `Rte` code might not comply with the assumed safety requirements of EB tresos AutoCore Generic RTE. These requirements are defined in Section 4.1, “Assumed safety requirements”. You must take appropriate actions to ensure that your own safety requirements are fulfilled.

2.4. Structure

This document contains the following chapters:

Chapter 2, “Document information“
: Provides a brief introduction into this document and its structure.

Chapter 3, “About EB tresos AutoCore Generic RTE“
: Introduces EB tresos AutoCore Generic RTE and the environment in which it can be used.

Chapter 4, “Application safety requirements, assumptions, limitations“
: Describes safety-related characteristics of EB tresos AutoCore Generic RTE.

Chapter 5, “ EB tresos AutoCore Generic RTE best practices“
: Describes best practices to correctly use EB tresos AutoCore Generic RTE.

Chapter 6, “Using EB tresos AutoCore Generic RTE safely“
: Describes how to use EB tresos AutoCore Generic RTE in a safe way.

Chapter 7, “Verification and review criteria“
: Describes the verification criteria.

3. About EB tresos AutoCore Generic RTE

The AUTOSAR Run-Time Environment (RTE) together with the OS, AUTOSAR COM and other basic software modules is the implementation of the Virtual Functional Bus concepts. The RTE implements the AUTOSAR Virtual Functional Bus interfaces and thereby realizes the communication between AUTOSAR software components.

3.1. Architecture of the surrounding system

EB tresos AutoCore Generic RTE is intended to be implemented in an AUTOSAR 4.0.3 environment. For more information on AUTOSAR, see [ASRWEB].

For additional assumptions on the software that runs on the same ECU like EB tresos AutoCore Generic RTE, see Section 4.2.2, "Assumptions".

3.2. Description of EB tresos AutoCore Generic RTE

EB tresos AutoCore Generic RTE consists of the following components:

- *EB tresos AutoCore Generic 8 RTE*
- User documentation that consists of the EB tresos AutoCore Generic RTE documentation [ACGRTERELNOTES] and this safety application guide

EB tresos AutoCore Generic 8 RTE is the implementation of an AUTOSAR `Rte` module. It is a code generator used to generate the implementation of an ECU-specific `Rte`, based on the provided configuration and system description.

This guide contains review instructions for the generated `Rte` code. These review instructions address generic programming faults which could lead to a violation of the *freedom from interference*. The review shall be performed for the generated `Rte` code. For the review instructions, see Chapter 7, "Verification and review criteria".

The basic activities required to safeguard the generated `Rte` are:

1. Verify the input of the EB tresos AutoCore Generic 8 RTE Generator.
2. Perform the reviews defined in Chapter 7, "Verification and review criteria".

3.3. Robustness of EB tresos AutoCore Generic RTE

3.3.1. What EB tresos AutoCore Generic RTE does not do

EB tresos AutoCore Generic RTE does not validate the provided AUTOSAR model. It assumes that the provided model is correct. It is the responsibility of the integrator to ensure this behavior.

EB tresos AutoCore Generic RTE does not verify the linker file of the system application.

EB tresos AutoCore Generic RTE does not use any safety mechanism within the generated code. Instead, the generated EB tresos AutoCore Generic 8 RTE is verified independently to ensure that it behaves as expected.

3.3.2. Robustness against hardware faults

EB tresos AutoCore Generic RTE is not robust against hardware faults. It is assumed that the hardware uses fault detection mechanisms such as dual-redundant processing (e.g. lockstep mode), and memory with error detection and error-correcting codes (ECC).

3.3.3. Robustness against systematic software errors

The generated `Rte` code of EB tresos AutoCore Generic RTE is verified in accordance with [ISO26262_1ST] to detect systematic software errors.

There are no means implemented in EB tresos AutoCore Generic RTE which make it robust against systematic errors in the calling software.

3.3.4. Robustness against configuration errors

EB tresos AutoCore Generic RTE is sensitive to configuration errors. It is based on the assumption that the provided configuration and system model are correct and therefore only performs basic configuration checks. The integrator is responsible for the correctness of the configuration and the system model.

3.3.5. Robustness against resource conflicts

EB tresos AutoCore Generic RTE uses AUTOSAR `Os` services and tasks to implement its functionality and to protect critical sections. It relies on a correct `Os` configuration and the reliable implementation of the corresponding services as well as task handling by the `Os`.

EB tresos AutoCore Generic RTE uses the AUTOSAR Communication Stack to perform *inter-ECU communication*. Therefore you must make sure to use end-to-end protection for safety-relevant inter-ECU communication.

EB tresos AutoCore Generic RTE does not use other software or hardware resources beyond those mentioned above in the features which are supported for safety-related environments. For all other features, the integrator must handle potential resource conflicts.

3.3.6. Robustness against interrupt overload

EB tresos AutoCore Generic RTE does not handle task scheduling and interrupts. Therefore, it is not robust against interrupt overload. The integrator must make sure to configure and build the system so that interrupt overload cannot happen or is handled without jeopardizing the safety-relevant parts.

3.3.7. Robustness against input errors

EB tresos AutoCore Generic RTE must be used in accordance with the AUTOSAR specification and the EB tresos AutoCore Generic RTE user's guide. Input errors caused by incorrect usage of `Rte` API functions must be avoided by verifying the *application software* according to its safety integrity level.

3.3.8. Robustness against data starvation

EB tresos AutoCore Generic RTE does not expect any data input from external sources, so it cannot be starved of data.

3.4. Assessment of change log, limitations, new features

For each EB tresos AutoCore release, all changes to a module are documented in the module release notes. It is important to review these changes because the behavior of a module may change in a release. The release notes contain the following sub-chapters:

- Change log

  The change log lists the changes between different versions. Changes can be typically categorized as bug fixes, new features, and improvements.

- New features

  This sub-chapter lists features that were recently introduced to EB tresos AutoCore RTE. Caution should be taken when using features that have not yet been proven in use.

- EB-specific enhancements

  For some features specified in the AUTOSAR software specifications, EB tresos AutoCore RTE provides additional functionality to improve the usability of the provided services. In many cases, the content of this chapter may be relevant for the safe usage of the `Rte` because the behavior between `Rte` implementations may be different.

- Deviations

  This sub-chapter lists all deviations from the applicable AUTOSAR software specifications. It might occur that the required functionality of a safety software component is not implemented by the `Rte`. It is therefore important to perform an impact analysis of these deviations for each project.

- Limitations

  In some cases, it is not possible to realize the behavior of an `Rte` API as specified in the AUTOSAR software specifications. Thus the behavior of the generated `Rte` may differ from the required functionality of a safety software component. It is therefore important to perform an impact analysis of these deviations for each project.

3.5. Field monitoring

Periodically, the EB tresos AutoCore known issues document is automatically created for each delivery under maintenance and is provided to the customer via the EB Command platform. The document contains the following information:

- related release version
- list of published known issues (bugs) with the following information:
  - unique ID of issue
  - affected module and version
  - defect description
  - workaround (if applicable)

To access the EB tresos AutoCore known issues, you need a login and password for EB Command.

3.6. Backward compatibility

EB tresos AutoCore Generic RTE is developed for a specific EB tresos AutoCore Generic release. Compatibility to other EB tresos AutoCore releases is not guaranteed.

4. Application safety requirements, assumptions, limitations

This chapter describes the following characteristics of the EB tresos AutoCore Generic RTE:

- the definition of the assumed safety requirements
- the assumptions made by EB tresos AutoCore Generic RTE on the environment
- the limitations of the feature set

4.1. Assumed safety requirements

This section defines the assumed safety requirements of EB tresos AutoCore Generic RTE. The main use case of EB tresos AutoCore Generic RTE is the integration with safety-related SWC applications in one ECU.

Id:	RTE.Interference
Doctype:	`requirement`
Status:	APPROVED
Version:	1
Description:	EB tresos AutoCore Generic RTE shall not interfere with any SWC application.
Safety rationale:	[ISO26262-6_1ST] 7.4.11 requires a freedom from interference.
Needs coverage of:	-
Comment:	The freedom from interference criterion, according to ISO26262, is achieved by a safety analysis.

Id:	RTE.Verification
Doctype:	`requirement`
Status:	APPROVED
Version:	1
Description:	EB tresos AutoCore Generic RTE shall be verified according to ISO26262 First Edition.
Safety rationale:	EB tresos AutoCore Generic RTE shall be capable of being integrated in ECUs together with safe SWC applications.
Needs coverage of:	-

4.2. Assumptions for safe usage of RTE

This section defines all assumptions on the environment which are necessary to use EB tresos AutoCore Generic RTE in a safe way.

NOTE

Fulfillment of the assumptions
The integrator must ensure that the assumptions named in this chapter apply to the project.

4.2.1. Defining assumptions

Assumptions must be defined using the following scheme:

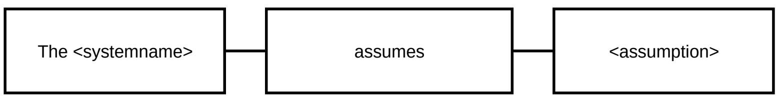

Figure 4.1. Scheme for defining assumptions

4.2.2. Assumptions

[RTE.Assumption.ValidatedTools]

EB tresos AutoCore Generic RTE assumes that the integrator takes appropriate measures to ensure the confidence in the used software tools, e.g. via tool qualifications.
Note: The used software tools include e.g. build environment, compiler, linker, and device flasher.

[RTE.Assumption.ReviewedToolOutput]

EB tresos AutoCore Generic RTE assumes that the integrator reviews the output of each used tool and takes appropriate measures if warnings or errors exist.
Note: The used tools are e.g. EB tresos Studio, the compiler, and the linker.

[RTE.Assumption.CompilerUsage]

EB tresos AutoCore Generic RTE assumes that the version and options of the used compiler are identical to the compiler specified in [ACGQSRTE] or are approved separately by Elektrobit Automotive GmbH.
Note: If any other compiler or options are chosen, then the user has to perform additional verification measures that are adequate and sufficient to ensure that there are no faults of EB tresos AutoCore Generic RTE caused by the chosen compiler or options. The used compiler or linker must at least issue a warning or error in the following cases:

- Inclusion of a non-existing file
- Usage of an undeclared or undefined identifier (function, variable, macro, or type)

- Implicit declaration of a function
- Implementation of functions with different parameter types than specified in the declaration of that function
- Assignment of a void return value of a function to a variable
- Assignment of incompatible data types
- Calling function with incompatible parameter data types
- Declaration or definition of duplicated identifiers (function, variable, macro, or type)

[RTE.Assumption.CorrectAutosarModel]

EB tresos AutoCore Generic RTE assumes that the AUTOSAR model used to generate the `Rte` is correct. That means it correctly reflects the requirements of the integrator and is valid according to AUTOSAR as well as to the restrictions of the configured modules.

[RTE.Assumption.ValidatedSoftware]

EB tresos AutoCore Generic RTE assumes that the software running on the ECU is validated according to the safety integrity level of its partition.

[RTE.Assumption.RteUsage]

EB tresos AutoCore Generic RTE assumes that the `Rte` is used according to [ASRRTE403]. This means among others:

- The *application software* correctly calls the `Rte` API functions necessary to perform the intended functionality.
- The return values of `Rte` APIs are checked for error conditions. If error conditions occur, these are handled appropriately.
- Any software on the ECU only calls `Rte` API functions which are allowed in the corresponding context. For example, a *runnable entity* only calls `Rte` API functions which are configured for it.
- No software on the ECU, with exception of the `Rte` itself, directly accesses any `Rte` internals like functions or variables, even if they are visible outside the `Rte`. To use an `Rte` feature, the official APIs and interfaces are used.
- Runnable entities are exclusively triggered by tasks defined by the `Rte` or `Rte` APIs.

[RTE.Assumption.SignalValidation]

EB tresos AutoCore Generic RTE assumes that SWCs validate data received from SWCs with lower safety integrity level or take other appropriate measures before using the data if invalid data can compromise safety requirements.

[RTE.Assumption.CoverageAnalysis]

EB tresos AutoCore Generic RTE assumes that the integrator performs a test coverage analysis for the integration tests, e.g. modified condition/decision coverage (MCDC).
Note: `Rte` code which is not covered by the integration tests must be validated by the integrator.

[RTE.Assumption.InitialValue]

EB tresos AutoCore Generic RTE assumes that the integrator configures initial values for sender-receiver communication if undefined initial values violate the system's safety requirements.

[RTE.Assumption.OsInitialization]

EB tresos AutoCore Generic RTE assumes that an exclusive area API is only used after the OS is started.

[RTE.Assumption.ExclusiveAreaImplementationMechanism]

EB tresos AutoCore Generic RTE assumes that the configured implementation mechanism of an exclusive area provides appropriate interruption protection for all execution contexts that an *executable entity* may run inside. This means that the interruption protection is not too weak, e.g. the exclusive area can be interrupted. This also means that the interruption protection is not over-dimensioned, e.g. a weaker implementation mechanism can be used.

[RTE.Assumption.ReservedIdentifiers]

EB tresos AutoCore Generic RTE has it own namespace and assumes that all identifiers beginning with `Rte_` are reserved for `Rte` usage only.

[RTE.Assumption.MemorySections]

EB tresos AutoCore Generic RTE assumes that the used OS ensures correct assignment of global variables and functions to memory sections if memory protection is enabled.

4.3. Limitations

See section *Limitations* in chapter *Module release notes* in [ACGRTERELNOTES]. For safety-related projects, the features are limited to those features which are defined by the safety requirements in chapter Section 4.1, “Assumed safety requirements” and the corresponding review criteria.

5. EB tresos AutoCore Generic RTE best practices

5.1. Introduction

This chapter concentrates on methods and best practices that enable you to avoid faults which could violate the assumed safety requirements. The information provided here should help you to:

- avoid errors
- recognize errors when they occur

5.2. Avoiding importer errors

One common source of `Rte` generation errors is an invalid import of the project data model caused by missing, incomplete, or invalid ARXML files. This chapter provides guidance on how to avoid these categories of errors.

5.2.1. Resolving missing artifact errors

During the import of AUTOSAR files into EB tresos Studio, references that exist within the ARXML files are verified and resolved. Occasionally, ARXML files reference artifacts (via an AUTOSAR short-name path) that do not exist in the project's data model or the imported ARXML files.

Typically, the causes are:

- An ARXML file was not included in the importer file set.
- An older version of an ARXML file was imported.
- The AUTOSAR path of the referenced artifact contains errors.
- The ARXML file references platform-dependent artifacts generated by EB tresos Studio.

If the referenced artifact is not already present within the EB tresos Studio data model or not present in the importer file set, EB tresos Studio logs an error and ignores the original artifact. As a result, the generated `Rte` code may not contain the required API function, or a variant may be generated that the application software component does not expect.

You can avoid importer errors of this type by resolving all importer errors before generation of the `Rte`. Because importer errors may not be visible in the problems view or error log after importing, you should re-verify the imported data periodically.

5.2.2. Validating the XML schema

When importing ARXML files, EB tresos Studio verifies if the content of the ARXML files conforms to the AUTOSAR XML schema for that version. If the content of an imported file does not completely conform to the AUTOSAR XML schema, validation errors and warnings are logged after the import.

XML schema validation errors typically occur when:

- the wrong AUTOSAR version is set within the ARXML file header
- elements within an ARXML file are not complete
- sub-elements are defined in the wrong order

If XML validation issues occur during import, the imported data model may become invalid. In that case, the generated `Rte` may contain errors because the required data may not be available or lead to an internal error within the `Rte` Generator.

You should never disable XML schema validation. If validation errors or warnings occur, resolve them before generating the `Rte`.

5.2.3. Assuring data model consistency

After an ARXML file is imported into EB tresos Studio, it is not possible to trace from where the imported data model artifacts originated. In order to ensure that the source ARXML files are consistent with the project's data model, additional measures are required.

It is recommended to keep both the imported ARXML files and the EB tresos Studio data model under version control. When changes are made to the associated ARXML files, the files shall be immediately imported into EB tresos Studio. In addition, the updated project data model file `SystemModel2.tdb` shall be updated in the version control system.

5.2.4. Explicitly importing code generator data model artifacts

Wizards that are executed in EB tresos Studio can generate artifacts directly into the project's data model. If this happens, it is not possible to monitor changes to the data model. That means EB tresos Studio has no history for changes to the internal data model. It is possible, though, for the wizards to generate ARXML files instead of writing to the internal data model.

If supported, a wizard's output shall be generated to an ARXML file. This file shall be explicitly imported into the project's data model. In addition, the generated ARXML files shall be added to the project's version control system.

5.3. Monitoring code changes via version control

It is a common practice for version control systems to exclude generated files from version control. Before compiling the system, the generated files shall then be regenerated.

While this practice has a valid reasoning, keeping generated RTE code under version control has the following advantages:

- Changes to the `Rte` code can be checked in together with changes to the ECU configuration files and ARXML files. As a result, the effects of the input files can be seen in the changes to the `Rte` code.

  This is not only helpful for understanding changes to the `Rte` code, but also for observing if unintended changes to the input files were made. For example, the `Rte` correctly implements a variant of an API function but that variant that does not realize the application software component's safety requirements.
- The changes to the implementation of a generated `Rte` function can be observed and documented. The following fault scenarios are possible:
  - A required functionality is removed, e.g. an event is no longer set after the operation is completed.
  - A required functionality is added, e.g. the function is blocking, and an additional function call is made.
  - A required functionality is modified, e.g. the sequence of an operation or the type of an operation is changed.

Recommended procedure in case of code changes:

1. Modify the data model or ECU configuration as required.
2. Regenerate the `Rte`.
3. Check in the changes.
4. Perform a delta review of the `Rte` code.

In this way, you can determine if required interfaces have been added, removed, or modified and if these changes are appropriate for the changes made to the data model and ECU configurations.

5.4. Configuring Rte ECU configuration parameters that impact safety

This section provides guidance with respect to the safe configuration of the `Rte` ECU configuration.

5.4.1. Configuring RteExclusiveAreaImplementation containers

For each exclusive area used by a software component prototype, the Rte Editor creates an **RteExclusiveAreaImplementation** configuration container. This container contains the parameters relevant to the configuration of the generated `Rte_Enter()` and `Rte_Exit()` API functions.

The `RteExclusiveAreaImplMechanism` configuration parameter is used to set the implementation mechanism for the specified exclusive area. The possible values are:

- ALL_INTERRUPT_BLOCKING
- COOPERATIVE_RUNNABLE_PLACEMENT
- OS_INTERRUPT_BLOCKING
- OS_RESOURCE
- EB_FAST_LOCK
- NO_LOCK
- USER_CALLOUT

The value EB_FAST_LOCK shall only be configured when it can be ensured that the calling software component never calls an `Os` or `Rte` API with the exclusive area enabled.

The value NO_LOCK shall only be configured when it can be ensured that the code protected by the exclusive area can never be concurrently executed, e.g. through call chain or timing analysis.

The value OS_RESOURCE shall only be configured when it can be ensured that the application software component can never be called in an interrupt context. For example, this may occur when the integration code executes a server runnable entity that is not mapped to a task.

5.4.2. Configuring Rte optimizations

This section provides the recommended configuration values of the `Rte` optimizations parameters.

`ComCbkNotInterruptable`

This parameter defines if `Com` callback functions are interruptible. If they are not interruptible (e.g. they run in non-interruptible ISRs or tasks), the `Rte` does not apply any data consistency mechanism within `Com` callbacks.

If the `Com` callback functions are unintentionally interrupted, the data received within the function may become corrupted. Therefore, it is recommended that this parameter is disabled.

`DataConsistencyMechanism`

This parameter defines the default data consistency mechanism. Possible options are: usage of `Os` resources (default) or interrupt blocking. The data consistency mechanism is applied to receive buffers/queues and inter-runnable variables if data corruption occurs.

The Rte Generator verifies if the default data consistency mechanism is sufficient to protect the associated data access. If the ECU extract is not complete (e.g. a runnable entity accesses a function associated to a port without declaring access to the port), the `Rte` may make false assumptions regarding the required data consistency mechanism. It is therefore recommended that this parameter is always set to interrupt blocking.

`DirectReadFromCom`

If this parameter is enabled, the `Rte` may do optimizations to save some RAM by directly reading values from `Com` instead of buffering them. In some scenarios where the software components poll `Com` values at a frequency higher than the update frequency, this optimization may cause an unwanted increase in run-time. If the I-PDU group is restarted after `Rte` startup, the `Rte` reads the initial values provided by the `Com` module. If disabled, the `Rte` always buffers values from `Com`. This may cause an increase in memory consumption. If the I-PDU group is restarted after `Rte` startup, the `Rte` keeps the last received value and does not read the initial value provided by the `Com` module.

The `Rte` verifies that the runnable entities that call the `Rte_Read` functions associated to a `Com` signal can never be called concurrently. If the ECU extract is not complete (e.g. a runnable entity accesses a function associated to a port without declaring access to the port), the `Rte` may make false assumptions regarding the concurrency of a `Rte_Read` function. It is therefore recommended that this parameter is disabled.

`DisablePartitionActiveChecks`

If set to true, the `Rte` does not fulfill requirements [SWS_Rte_02535], [SWS_Rte_02536], and [SWS_-Rte_02538] anymore. This means that the `Rte` does not check for each API function if the current partition is active. You have to ensure that no callbacks and no API functions are called before `Rte_Start()`/ after `Rte_Stop()` was executed. The effect of this optimization depends on the number of generated API functions and how often they are called by the application.

If multiple partitions are configured, it might be difficult to ensure that no API functions are called before `Rte_Start()` and after `Rte_Stop()`. This is because separate `Os` schedule tables are defined for each partition and they are not started at the same time. As a result, the scheduling of timing events in one partition may begin before the `Rte_Start()` function in another partition was completed. It is therefore recommended to disable this parameter.

`GenerateEmptyRteStartStopStubs`

If set to true, the `Rte` generates empty `Rte_Start()` and `Rte_Stop()` stubs. This is required when the `EcuM` operates from within a non-trusted partition and calls a trusted partition.

Because of the memory protection issue described above, it is recommended to enable this configuration parameter and call the partition-specific `Rte_Start_{partition}()` and `Rte_Stop_{partition}()` functions directly. The `EcuM`'s partition can then remain as non-trusted. This ensures that the basic software cannot corrupt the memory of partitions that safety application software components run on.

`InterruptBlockingFunction`

This parameter defines the functions which shall be used to block interrupts. Possible options are:

- `SuspendResumeAllInterrupts` (default): uses the standard AUTOSAR `Os` functions `SuspendAllInterrupts` and `ResumeAllInterrupts`

- `DisableEnableAllInterrupts`: uses the standard AUTOSAR `Os` functions `DisableAllInterrupts` and `EnableAllInterrupts`
- `TS_IntDisableEnable`: EB-specific functions from the EB `Base` module
- `Rte_UserDefinedIntLockUnlock`: user-defined functions/macros which have to be declared in a header file called `Rte_UserDefinedIntLock.h`

If `TS_IntDisableEnable` is configured, the `Os` is not notified that the interrupts are locked. As a result, calls to `Os` functions may not work correctly. This optimization shall only be used when it can be assured that the generated `Rte` code never calls another module with the interrupts locked.

6. Using EB tresos AutoCore Generic RTE safely

If EB tresos AutoCore Generic RTE is used in an unintended way, faults may occur. The purpose of this chapter is to describe the scenarios of such faults.

6.1. Using the Smc in a multi-core environment

The Shared Memory Communicator (`Smc`) module supports data consistency mechanisms for inter-core communication (spinlocks). However, it does not support hardware-specific cache handling.

To verify that the `Smc` can be used in a multi-core environment, check that caches are disabled and that a shared memory approach between different cores is applicable. If this is not the case, use *Mixed* as the inter-partition communication mechanism. This mechanism still uses the `Smc` for the hardware-independent inter-partition intra-core communication and the `Ioc` module for the hardware-dependent inter-partition inter-core communication.

6.2. The basic software might not run correctly on the ECU

To ensure that the basic software always runs correctly on the ECU, set the implementation mechanism of an exclusive area that is associated with a basic software module entity to `All Interrupt Blocking`. If you set the implementation mechanism to any other value, the basic software might not run correctly on your ECU.

6.3. The Rte ignores incoming transmissions via the Com module

Any `Rte` functionality is only available after the `Rte_Start()` function was called first. If `Rte_Start()` was not called, the `Rte` ignores incoming transmissions via the `Com` module. If the application depends on ignored incoming `Com` data, the application is not initialized correctly.

To avoid that the `Rte` ignores all incoming transmissions via the `Com` module, call `Rte_Start()` before the `Com` module starts receiving transmissions.

6.4. Blocked runnable entities cause operating system errors

If a category 2 runnable entity has implicit access to an exclusive area and then gets blocked, the runnable entity causes operating system errors.

To avoid operating system errors, ensure that category 2 runnable entities do not have implicit access to exclusive areas. For details on runnable entities categories, see section *Executable entity categories* within [ACGRTEUSRDOC].

6.5. Blocked runnable entities cause wrong timing behavior

If multiple runnable entities are mapped to the same task and at least one of these runnable entities is of category 2, the Rte Generator reports a warning. If you ignore this warning, the category 2 runnable entity blocks the execution of the other executable entities and thus influences the overall timing behavior.

To avoid wrong timing behavior, map category 2 runnable entities to a separate extended task.

6.6. Declaration of runnable entity access to a port or exclusive area

An `Rte` API may only be used by the runnable that describes its usage. If a runnable entity uses an `Rte` API without declaring access to it, the data consistency mechanism implemented by the `Rte` may provide the required protection.

6.7. Uncontrolled behavior of the software components

The base type property settings are highly dependent on the microcontroller target, the derivative, the compiler, and the compiler settings. Thus, the default properties may only be used for the default:

- target,

- derivative,
- compiler, and
- compiler settings.

If you use the default base type properties for a target, derivative, compiler or for compiler settings that differ from the default, the software components behave in an uncontrolled way. The same applies if you change the default base type properties for the default target, derivative, compiler, and compiler settings.

To avoid uncontrolled behavior of the software component, check and adapt the default properties accordingly if you want to use the `Rte` in a different environment (different derivative, compiler, compiler settings etc.).

6.8. Atomic access in the context of data structures

The `Rte` uses structure types for complex data buffers, e.g. to hold status and value information for a data element. Atomic access to structure elements is compiler-dependent and therefore cannot be guaranteed by the `Rte`.

If the access to structure elements is not atomic, you should set the atomic access attributes to false even if atomic access for single variables of a certain base type is possible.

6.9. Using linear conversion in sender-receiver communication

If you use data conversion, you risk to introduce undefined or unspecified behavior. Use this feature only if you know the effects on the generated code.

6.10. Disabling partition active checks

After disabling the partition active checks, you must ensure that no runnables are triggered before `Rte_Start()` or after `Rte_Stop()` is called.

The partition active check is implemented to fulfill the requirements [SWS_Rte_02535], [SWS_Rte_02536], and [SWS_Rte_02538] of [ASRRTE422]. If this check is not generated, it can happen that buffers are modified or runnables are started even if `Rte_Start()` was not called before. If `Rte_Stop()` is called, it can still happen that runnables, which are in execution, trigger further runnables.

For example, by calling `Rte_Call()`, the `Rte` activates the server runnable even if the partition is already stopped. For the `Com` callbacks, it might be possible that incoming data triggers runnables or modifies buffers although the `Rte` was not started or was already stopped. You must ensure that no further runnables are triggered or that such activations do not interfere with any `Rte` functionality.

6.11. Using blocking `Rte` APIs

If an AUTOSAR interface defines a wait point, the `Rte` generates a blocking API. There are two scenarios where a blocking API can be problematic:

- No time-out value is configured. In this case, the task blocks forever when the associated operation fails (or until shutdown).
- The API is used by a runnable entity that is mapped to a task with other `Rte` events. As a result, the other `Rte` events cannot be handled until the blocking operation is completed.

The following is recommended when wait points are involved:

- Map blocking events to separate tasks.
- Always define a time-out value for wait points.

6.12. Defining initial values

Without an initial value, the variable data prototype might be in an invalid state until the first value is sent.

Therefore, always define a valid initial value. This ensures that the sender runnables are started before the receivers. Alternatively, you can define a constant that indicates that the current value is not set yet.

6.13. Avoiding unconnected ports

If a software component port is not connected at generation time, the `Rte` generates a stub implementation of the corresponding API function. That means that the software component can use the API. However, the software component receives a constant value instead of the required information. As a result, the behavior of the software component may not be as expected.

If a software component is not configured for multiple instantiation, ensure that all ports are connected before generating the `Rte`.

6.14. Avoiding queuing of variable data

If a queue is defined, a queue overflow can always occur. Then the receiver may consume data that is no longer valid. For example, the queue quickly fills up and new data is discarded.

To avoid a queue overflow, define a queue length that is large enough to handle high load periods. Alternatively, configure the priority of the receiver tasks with a higher value than the sender tasks.

6.15. Accessing INOUT or OUT parameters

If an `Rte` API with INOUT or OUT parameters fails, it may not update one or more of the output parameter values. Therefore, the software component must always evaluate the return value before using output parameter values. The return value of an `Rte` API shall never be cast to void. You should always analyze compiler warnings to determine if a return value was not handled.

7. Verification and review criteria

This chapter outlines the verification and reviews to be performed against faults which could lead to a violation of assumed safety requirements for the RTE, see Section 4.1, “Assumed safety requirements”

7.1. Verification according to ISO 26262 First Edition

For every EB tresos AutoCore Generic release, the EB tresos AutoCore Generic Quality and Metric report [ACGQMR] is generated. The methods and metrics recommended by [ISO26262_1ST] for all EB tresos AutoCore Generic modules including `Rte` can be verified in this report.

7.2. Review Objectives

This section defines review objectives that address generic programming faults which can lead to *freedom from interference* violation in the generated RTE code.

Freedom from interference violations occur whenever unintended memory regions are written. Consequences of such violations can be corrupted data or undefined behavior for other parts of the application. Writing outside of array borders or data types are examples for such violations.

7.2.1. Review tool

To assist the user in identifying the review objectives, a review tool was developed. This tool is part of the RTE and generates an HTML file from the generated source code where all assignments and function calls are highlighted. This report also contains justifications and/or additional detailed review instructions with a corresponding id that can be found in Section 7.2.2, “Review instructions”.

The report is generated if the RTE configuration parameter `/Rte/RteGeneration/GenerateRteFreedomFromInterferenceReviewInstructions` is set to `true`. In addition, the justification and detailed review instruction comments are generated in the regular source files.

If the report generation is enabled, the RTE generator decides for each generated expression if it contains a variable assignment or a function call. Those expressions are then categorized into one of two categories described in Table 7.1, “Highlighting colors” based on rules derived from Section 7.2.2, “Review instructions”.

Color	Description
`Rte_Status = RTE_E_TIMEOUT;`	The expression is analyzed by the review tool according to the reviews in Section 7.2.2, “Review instructions”. A comment is generated afterward containing the according review ids with justifications.
`Rte_Smc_Rte_<name>[tailIdx] = 9U;`	The expression is analyzed by the review tool according to the reviews in Section 7.2.2, “Review instructions”, but the expression contains some aspects that have to be reviewed manually. To help with this task a comment is generated afterward containing the according review ids with justifications and/or detailed review instructions.
`/* RTE.Review.<id>: ... */`	The review tool comments containing the justifications and detailed review instructions.

Table 7.1. Highlighting colors

The generated HTML files can be found in the `doc/rte_review_report` folder within the configured output directory. The files have the same name with `.html` extension as the generated header and source files.

The goal of the review tool is to help with performing the reviews in Section 7.2.2, “Review instructions”. The review tool is not meant as replacement for the manual reviews. The reviewer has to verify that the provided justifications are correct and has to follow the review instructions for generated review tool comments.

WARNING

Experimental proof-of-concept

The review tool feature of the RTE is currently an experimental proof-of-concept and is **not** recommended to be used in production.

7.2.2. Review instructions

The review instructions which are specified in the next chapters shall be applied to all sub-expressions of a C statement and not only to the C statement itself i.e. variable assignments or function calls that are nested into other expression have to be reviewed, too. The review instructions shall be applied from the innermost to the outermost expression. If for example the dereference operator is applied to a comma operator, the review for the dereference operator can only be checked after the reviews for the comma operator arguments were done and the resulting expression of the comma operator was determined. The resulting value will then serve as an input for the review of the pointer dereference.

For source code examples see Appendix B, “Review source code examples“.

7.2.2.1. Variable assignments

Variable assignment

For each expression on the left-hand side of an assignment:

- verify for each array access in the form of `arr[index]` that the **Array access** review is successful
- verify for each pointer dereference in the form `*ptr` that the **Pointer dereference** review is successful

Review Tool

If the left-hand-side of an assignment does not contain any array access or pointer dereference expression then the review tool will justify the assignment with the id `RTE.Review.VariableAssignment.Justification`.

Array access

Verify that:

- if the array name is a pointer, then it points to a *valid address* where enough bytes are reserved to write the specified element , and
- the array access index is smaller than the length of the referenced array

Review Tool

If the array name is a pointer, then a review instruction with id `RTE.Review.WriteAccess.PointerTypeAndNotNull` is generated to check that the address points to a valid address.

If the array name is not a pointer, then a justification with id `RTE.Review.WriteAccess.ArrayType.Justification` is generated.

A review instruction with id `RTE.Review.WriteAccess.ArrayElement` is generated to check the array index.

Pointer dereference

Verify that the pointer:

- points to a *valid address* (this also includes that if the pointer is casted, it must be verified that the cast is valid)

Review Tool

A review instruction with id `RTE.Review.WriteAccess.PointerTypeAndNotNull` and the expected pointer type is generated to check for a valid address.

If the dereference operation is applied to a cast expression, it must be checked that the cast is valid and the review tool generates the id `RTE.Review.WriteAccess.PointerCast`.

7.2.2.2. Function calls

Function call

Verify for each function call that:

- the called function is defined as `((void)0)` by a C preprocessor macro, or
- the called function has no arguments or only *IN arguments*, or
- For each *OUT argument* and *INOUT argument*:
  - verify that if the passed argument is a pointer to a payload and another argument specifies its size, then the **INOUT/OUT argument with size argument** review must be successful, or
  - verify that if the passed argument is a pointer to a variable or head of an array without a corresponding size argument, then the **INOUT/OUT argument without size argument** review must be successful

Review Tool

If all hook functions are disabled then they are defined as `((void)0)` and the review tool justifies the expression with id `RTE.Review.FunctionCall.HookDisabled.Justification`.

If a function has no arguments or only in arguments then the review tool justifies the expression with id `RTE.Review.FunctionCall.ReadOnly.Justification`.

INOUT/OUT argument without size argument

Verify that:

- the passed pointer argument points to a valid address, and
- the called function does not write outside of the boundaries of the referenced variable

Review Tool

A review instruction with id `RTE.Review.WriteAccess.PointerTypeAndNotNull` is generated to verify that the pointer is valid. If it can be ensured that the passed address is always valid e.g. if an array type is defined as a global or local variable and its address is passed, the justification `RTE.Review.WriteAccess.ArrayType.Justification` is added.

A review instruction with id `RTE.Review.FunctionCall.OutInoutArg` and with the expected number of bytes is generated to verify that the called function does not write outside of the boundaries.

If the review tool cannot determine the size, because the parameter is a void pointer, the id `RTE.Review.FunctionCall.OutInoutVoidArg` is generated.

INOUT/OUT argument with size argument

Verify that:

- the passed pointer argument points to a valid address, and
- the passed size is not bigger than the size of the payload, and

- the called function does not write more bytes to the payload than indicated by this argument

Review Tool

If the corresponding size argument is passed in the form `size(var)` and the pointer argument can point to `NULL` then a review instruction with id `RTE.Review.WriteAccess.PointerTypeAndNotNull` is generated.

If the corresponding size argument is passed in the form `size(var)` and the pointer argument cannot point to `NULL` (e.g. because the address-of operator is used), then a review instruction with id `RTE.Review.WriteAccess.PointerType` is generated to verify that the argument points to the type of the size argument.

If the corresponding size argument is passed in the form `size(var)` and the pointer argument cannot point to `NULL` (e.g. because the address-of operator is used) and the type of the size argument matches the type of the pointer argument, a justification with id `RTE.Review.WriteAccess.PointerType.Justification` is generated.

If the called function was already qualified (e.g. `TS_MemCpy(dest, src, size)`) to not write more bytes than indicated by the size argument, then the justification with id `RTE.Review.FunctionCall.OutInoutArgSizeIndicator.Justification` is generated.

7.2.2.3. C preprocessor macros

C preprocessor macro

For each C preprocessor macro:

- verify that if the defined expression is not *read-only* then the **Possible write access** review is successful

Review Tool

If the macro does not map to a *read-only* expression, then a review instruction with id `RTE.Review.PreprocessorMacro.PossibleWriteAcccess` is generated to ensure that the user of the macro does not write out of memory boundaries of the returned variable/dereferenced pointer.

If macro does map to a read-only value, then a justification with id `RTE.Review.PreprocessorMacro.ReadOnly.Justification` is generated.

Possible write access

Verify for all usages of this macro that:

- the macro is used on the left-hand side of an assignment and the **Variable assignment** review is successful, or
- the macro is used as function argument in a function call and the **Function call** review is successful, or
- the macro is used in other contexts

Appendix A. Reserved identifiers

See **[RTE.Assumption.ReservedIdentifiers]** for identifiers reserved for `Rte`. You must not use these identifiers as names for any user-defined object.

The prohibition extends to the names of the objects that you create in AUTOSAR module configurations in EB tresos Studio.

Appendix B. Review source code examples

This chapter contains general C source code examples with short explanations for the reviews in Section 7.2.2, "Review instructions".

B.1. Variable assignment examples

Example B.1. Variable assignments without further reviews

```
void function()
{
  short local_short;
  short* local_pointer;

  // valid, because the expression on the left-hand side is a not a pointer or array
  local_short = 10;

  // valid, because the expression on the left-hand side contains a pointer, but
  // is not a pointer dereference expression
  local_pointer = &local_short;
}
```

Example B.2. Array access

```
void function(int* array_argument, int array_argument_length)
{
  int local_array[3] = {0};

  // valid, because index < array length
  local_array[1] = 4;

  // invalid, because index >= array length
  local_array[3] = 5;

  // only valid if all calling function pass an array with at least two elements
  // as first function argument
  array_argument[1] = -1;
```

```
  if(array_argument_length > 1)
  {
    // only valid if all calling functions provide valid function arguments.
    // if valid arguments are provided, then the line is only executed if array
    // has at least 2 elements
    array_argument[1] = -1;
  }
}
```

Example B.3. Pointer dereference

```
void function(int* pointer_argument)
{
  short* local_pointer;
  int local_int = 0;
  char local_char = 'a';
  short local_short = 3;

  // only valid if all calling functions provide a valid function argument
  *pointer_argument = 2;

  // only valid if all calling functions provide the address of an array
  // with at least two elements as function argument
  *(pointer_argument += 1, pointer_argument) = 2;

  // invalid, because pointer is not initialized
  *local_pointer = 4;

  // valid, because pointer has valid address and sizeof(short) <= sizeof(int)
  *(local_pointer = (short*)(void*)&local_int, local_pointer) = 7;

  // only valid, if sizeof(short) <= sizeof(char) on the architecture
  *(local_pointer = (short*)(void*)&local_char, local_pointer) = 14;

  // invalid, because pointer has invalid address
  *(local_pointer = &local_short + 1, local_pointer) = -3;
}
```

B.2. Function call examples

Example B.4. Function calls without further reviews

```
extern void function_a(void);
extern void function_b(void const* data, int data_length);
#define function_c(data, data_length) ((void)0)

void function_e(int* array, int array_length)
{
  int* local_pointer;

  // valid, because function has no arguments
  function_a();

  // valid, because function has only IN arguments
  function_b(local_pointer, sizeof(int));

  // valid, because function_c is defined as ((void)0)
  function_c(local_pointer, sizeof(int));
}
```

Example B.5. Function calls with pointer arguments

```
extern void function_a(int* pointer_argument);

void function_b(void)
{
  int local_int = 5;
  int* local_pointer;

  // address is invalid => valid if function_a only reads the argument
  function_a((int*)0);

  // address is valid => valid if function_a writes <= sizeof(int) bytes
  // to local_int
  function_a(&local_int);

  // local_pointer is not initialized => valid if function_a only reads the argument
  function_a(local_pointer);
}
```

Example B.6. Function calls with data type size argument

```
void function_a(void* data, int type_size);

void function_b(void)
{
  int array[5] = {0};
  short value;

  // size argument = size of type => valid if function_a does not
  // write more than sizeof(array) bytes to array
  function_a(array, sizeof(array));

  // invalid, because size argument > size of type
  function_a(&array[2], sizeof(array));

  // sizeof(char) <= sizeof(short) => valid if function_a does not
  // write more than sizeof(char) bytes to value
  function_a(&value, sizeof(char));
}
```

Example B.7. Function calls with array length argument

```
extern void function_a(int* array, int array_length);

void function_b(void)
{
  int array[5] = {0};

  // array length argument = array length => valid if function_a does not write
  // more than 5 elements to array
  function_a(array, 5);

  // invalid, because length argument > length of passed array (= 4)
  function_a(&array[1], 5);

  // length argument < length of passed array (= 4) => valid if function_a does
  // not write more than 3 elements to array
  function_a(&array[1], 3);
}
```

B.3. Comma operator examples

Example B.8. Comma operator examples

```
extern void function_a(int const* pointer_argument);
extern void function_b(void* pointer_argument, int data_type_size_argument);
#define OK 0

int global_variable = 0;

int function_c(void)
{
  int status_variable;
  int* local_pointer;

  // - first operand is function call with only IN arguments
  // - last operand is a constant
  // => valid because reviews for all operands are successful
  status_variable = (function_a(&global_variable), OK);

  // invalid because local_pointer is not initialized but used
  // in pointer dereference expression on the left-hand side
  // of variable assignment
  *(function_a(local_pointer), local_pointer) = 5;

  // NULL_PTR is passed as first argument of function_b
  // => only valid if function_b does not write to first argument
  function_b((function_a(&global_variable), NULL_PTR), sizeof(int));

  return status_variable;
}
```

B.4. C Preprocessor macro examples

Example B.9. C preprocessor macros without further reviews

```
extern void function_a(int* array, int array_length);
int global_array[4] = {0};

// valid, because no expression is defined
#define MACRO_A

// valid, because defined expression is a constant
#define MACRO_B 0

// valid, because defined expression is a function identifier
#define MACRO_C function_a

// valid, because defined expression is a pointer to a constant
#define MACRO_D (int const*)&global_variable
```

Example B.10. C preprocessor macros with manual reviews

```
extern void function_a(int* array, int array_length);

int global_variable = 35;
int global_array[4] = {0};

// defined expression is a pointer to a variable
// => valid if all usages of the macro are valid
#define MACRO_E &global_variable

// defined expression is comma operator and last operand is an array
// => valid if reviews of individual statements are successful and
// all usages of the macro are valid
#define MACRO_F (function_a(global_array, 4), global_array)

// the defined expression is a compound statement
// => valid if reviews of individual statements are successful
#define MACRO_G \
do { \
  function_a(global_array, 4); \
} while(0)
```

Appendix C. Document configuration information

This document was created by the DocBook engine using the source files and revisions listed below. All paths are relative to the directory https://subversion.ebgroup.elektrobit.com/svn/EB_tresos/DocEBtresos/asc_-DocRTE/trunk/doc/public/binding/safety_application_guide.

Filename	Revision
..\..\fragments\safety_guide\bibliography\Bibliography.xml	3080
..\..\fragments\safety_guide\glossary\Glossary.xml	6229
About_the_RTE.xml	3080
appendix\ReservedIdentifiers.xml	3080
appendix\ReviewCodeExamples.xml	6229
assumptions\Assumptions.xml	6229
AutoCore_RTE_safety_application_guide.xml	6229
constraints\Assumed_Requirements.xml	3080
constraints\Index.xml	3080
constraints\RteFeatureSetLimitations.xml	3080
Document_information.xml	3080
History.xml	6229
RTE_BestPractices\Best_Practices.xml	3080
RTE_BestPractices\VerificationAndReviews.xml	6229

Glossary

`EB tresos AutoCore Generic RTE`	EB tresos AutoCore Generic RTE is the AUTOSAR RTE implementation. It provides the configuration interface and the code generator to generate the RTE code. EB tresos AutoCore Generic RTE generates the RTE target implementation based on a predefined input model. See *Runtime Environment (RTE)*.
`Application software`	The application software is the software implemented within the AUTOSAR application layer. Among others this includes the *AUTOSAR application software components*.
`Basic software module (BSWM)`	A basic software module is an AUTOSAR basic software module.
`Electronic control unit (ECU)`	An electronic control unit is an AUTOSAR ECU (compare [ASRGLOSSARY]).
`Executable entity`	An executable entity is either a *runnable entity* or an AUTOSAR *BSW schedulable entity*. In contrast to an AUTOSAR executable entity this term does e.g. not cover AUTOSAR BSW interrupt entities.
`Freedom from Interference`	Freedom from Interference implies that one software component does not hinder or interferes with the functioning of other software component. (see [ISO26262-6_1ST]).
`IN argument`	An IN argument is a value that is passed to an *IN parameter* of a called function.
`INOUT argument`	An INOUT argument is a value that is passed to an *INOUT parameter* of a called function.
`Integrator`	An integrator is a person who integrates the software on the ECU.
`Inter-ECU communication`	Inter-ECU communication refers to the communication between different ECUs, typically using the AUTOSAR Com module.
`OUT argument`	An OUT argument is a value that is passed to an *OUT parameter* of a called function.
`Parameter direction`	All function parameters fall into one of two classes: ▶ parameters that are strictly read-only (IN parameters), and

	- parameters whose value may be modified by the function (INOUT and OUT parameters). The parameter directions are specified in chapter "5.2.6.5 API Parameters" of [ASRRTE422]
`Read-only expression`	An expression is considered as read-only if: - it's a simple value (e.g. `5U`) - it's a constant variable - it's a pointer to a constant (`P2CONST`) - it's a function identifier - it's a function call that returns a read-only value - a comma expression where the last operand is read-only
`Runtime Environment (RTE)`	The Runtime Environment (RTE) or also AUTOSAR Runtime Environment is an AUTOSAR module. It is specified by [ASRRTE403].
`Runnable entity (RE)`	A runnable entity (RE) is an AUTOSAR runnable entity.
`BSW schedulable entity`	A BSW schedulable entity is a C-function of a *BSW module* which runs in the context of an OS task and is scheduled via an OS service, e.g. via a schedule table or an OS event.
`Software component (SWC)`	A software component (SWC) is an AUTOSAR software component.
`System application`	A system application is the software implementing the functionality of an *ECU*. In context of the EB tresos AutoCore Generic RTE this includes the *application software*, the *AUTOSAR RTE*, the AUTOSAR basic software and the AUTOSAR OS.
`Valid Address`	A memory address which is within the allocated memory of the module and can be written to without overwriting any other softwares memory.

Bibliography

[ACGQMR] *EB tresos AutoCore Quality And Metric Report*

[ACGQSRTE] *Quality Statement*

[ACGRTERELNOTES] *EB tresos® AutoCore Generic 8 RTE documentation*: ACG8 RTE release notes, product release 8.5

[ACGRTEUSRDOC] *EB tresos® AutoCore Generic 8 RTE documentation*: ACG8 RTE user's guide, product release 8.5

[ASRGLOSSARY] *Glossary*, Version 4.2.2 Final

[ASRRTE403] *Specification of RTE*, Version 3.2.0 Final, 2011-10-26

[ASRRTE422] *Specification of RTE*, Version 4.2.2 Final

[ASRWEB] *AUTOSAR website*, 2015-03-13
http://www.autosar.org

[ISO26262_1ST] *INTERNATIONAL STANDARD ISO 26262*: Road vehicles - Functional safety, 2011

[ISO26262-6_1ST] *INTERNATIONAL STANDARD ISO 26262-6*: Road vehicles - Functional safety - Part 6: Product development at the software level, 2011