Elektrobit

EB tresos® AutoCore OS safety application guide

for ASIL-B applications

Date: 2021-04-28, Document version 1.3, Status: RELEASED

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

Confidential information.

Table of Contents

1. Document history

The revisions of the document as a whole are documented here. Each revision contains many smaller changes to the individual parts that go into the document, often by different contributors. The exact change history for each part is maintained in a revision control system.

Version	Date	State	Description
0.1	2017-10-27	PROPOSED	Initial draft
0.2	2018-02-20	PROPOSED	▶ Corrected findings from TE review. ▶ Added coverage trace to SAR (not visible in PDF). ▶ Added the following verification criteria, along with explanatory text: ▶ Api.MaxRuntime.NoErrorHook ▶ Implementation.UserGetStackInfo.PresetStackPointer ▶ Implementation.UserGetStackInfo.Parameter ▶ Implementation.GetTaskStack.Parameter ▶ Implementation.Resource.AlwaysAcquire ▶ Implementation.Spinlocks.OtherApi ▶ Implementation.OptimizedSchedule ▶ Implementation.UnexpectedReturn ▶ Implementation.AbsoluteCounterValue ▶ Implementation.TimeStamp ▶ Implementation.TimeStampElapsed
1.0	2018-03-08	PROPOSED	Rework following walkthrough.
1.0	2018-03-12	RELEASED	Verification report: [ASCOS_SAG_VR_1].
1.1	2019-03-06	PROPOSED	Removed the section about the GPT driver.
1.2	2021-04-27	PROPOSED	▶ Removed the OS fast locking section and its related APIs. ▶ Renamed OS_UserGetStackInfo to OS_GetStackInfo. ▶ Added the following verification criteria, along with explanatory text: ▶ Api.InterruptSourceRegisters ▶ Api.RefParam.GetApplicationState ▶ Implementation.ScheduleApi ▶ Added the following APIs to the OS services list:

Version	Date	State	Description
			▶ ActivateTaskAsyn ▶ ClearPendingInterrupt ▶ ControlIdle ▶ DisableInterruptSource ▶ EnableInterruptSource ▶ GetApplicationState ▶ GetCurrentApplicationID ▶ GetNumberOfActivatedCores ▶ SetEventAsyn ▶ Added ASIL-B recommended methods. ▶ Corrected typographical errors.
1.3	2021-04-28	RELEASED	Verification report: [ASCOS_SAG_VR_2].

Table 1.1. Document history

2. Document information

2.1. Objective

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

2.2. Scope and audience

This guide describes the use of EB tresos AutoCore OS in system applications which have safety allocations up to ASIL-B. It is valid for all projects and organizations which use EB tresos AutoCore OS 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 shall have the following knowledge and abilities:

- C-programming skills, especially in embedded systems.
- Experience in programming AUTOSAR-compliant ECUs.

Experience with safety standards and software development in safety related environments is assumed.

2.3. Motivation

This guide provides information on how to use EB tresos AutoCore OS correctly in safety-related projects. If EB tresos AutoCore OS is used differently, the resulting system might not comply with the assumed safety requirements of EB tresos AutoCore OS. These requirements are defined ins Chapter 5, “Assumed requirements“. You must take appropriate actions to ensure that your own safety requirements are fulfilled.

2.4. Structure

Chapter 2, “Document information“ (this chapter) gives a brief introduction to this document and its structure and typographical conventions.

Chapter 3, “About EB tresos AutoCore OS“ describes EB tresos AutoCore OS.

Chapter 4, “Using EB tresos AutoCore OS safely“ describes how to use the EB tresos AutoCore OS safely.

Chapter 5, “Assumed requirements“ describes the assumed requirements.

Chapter 6, “Required configuration settings“ lists the criteria that you must use to verify that EB tresos AutoCore OS is configured correctly for safe use. Safe use can only be determined for the *item* in which EB tresos AutoCore OS is used. You must perform a hazard analysis and risk assessment for the item. See [ISO26262-3_1ST], clause 7.

Appendix A, “Reserved identifiers“ lists the program identifiers that are reserved for use by EB tresos AutoCore OS.

Appendix B, “Safety of OS services“ lists the API services provided by EB tresos AutoCore OS and their availability for use in systems with safety-related functionality.

The chapters Chapter 4, “Using EB tresos AutoCore OS safely“ and Chapter 6, “Required configuration settings“ contain text describing the activities that need to be performed during the safety analysis of the item. The text contains the rationale behind the activites. The mandatory verification of your configuration and use of EB tresos AutoCore OS *that you must perform* is presented in the following style:

[Unique.Identifier]
<Optional conditional clause> Verify that <description of verification step>.

2.5. Typography and style conventions

The signal word *WARNING* indicates information that is vital for the success of the configuration.

WARNING	**Source and kind of the problem** What can happen to the software? What are the consequences of the problem? How does the user avoid the problem?

The signal word *NOTE* indicates important information on a subject.

NOTE **Important information**

Gives important information on a subject

The signal word *TIP* provides helpful hints, tips and shortcuts.

TIP **Helpful hints**

Gives helpful hints

Throughout the documentation, you find words and phrases that are displayed in **bold**, *italic*, or `monospaced` font.

To find out what these conventions mean, see the following table.

All default text is written in Arial Regular font.

Font	Description	Example
Arial italics	Emphasizes new or important terms	The *basic building blocks* of a configuration are module configurations.
Arial boldface	GUI elements and keyboard keys	1. In the **Project** drop-down list box, select Project_A. 2. Press the **Enter** key.
Monospaced font (Courier)	User input, code, and file directories	The module calls the `BswM_Dcm_RequestSessionMode()` function. For the project name, enter `Project_Test`.
Square brackets [ ]	Denotes optional parameters; for command syntax with optional parameters	`insertBefore [<opt>]`
Curly brackets {}	Denotes mandatory parameters; for command syntax with mandatory parameters	`insertBefore {<file>}`
Ellipsis …	Indicates further parameters; for command syntax with multiple parameters	`insertBefore [<opt>…]`
A vertical bar |	Indicates all available parameters; for command syntax in which you select one of the available parameters	`allowinvalidmarkup {on|off}`

3. About EB tresos AutoCore OS

EB tresos AutoCore OS is an implementation of the operating system module from the AUTOSAR standard. Apart from the documented deviations, EB tresos AutoCore OS is a conformant implementation of the AUTOSAR Os module. You can use EB tresos AutoCore OS in projects where a safety integrity level up to ASIL-B is demanded.

3.1. Architecture of the surrounding system

The *surrounding system* is defined as the hardware and software that comprises the ECU, excluding EB tresos AutoCore OS.

The safety of EB tresos AutoCore OS does not depend on the architecture of the surrounding hardware. EB tresos AutoCore OS uses features of the microcontroller to provide its specified functionality.

EB tresos AutoCore OS is independent of the software architecture of the surrounding system.

3.2. Description of EB tresos AutoCore OS

EB tresos AutoCore OS provides an implementation of the AUTOSAR operating system module suitable for use up to ASIL-B as defined in [ISO26262_1ST].

To find a list of all the OS services defined by EB tresos AutoCore OS and their implementation status, integrity category etc., see Appendix B, “Safety of OS services“.

EB tresos AutoCore OS has some memory protection features that you might be able to use as part of your freedom from interference arguments. However, EB tresos AutoCore OS does not provide any spatial freedom from interference guarantees. For details about the memory protection features see [ASCOS_USERGUIDE].

EB tresos AutoCore OS has timing protection features such as arrival rate monitoring and execution budget monitoring. These features do not provide temporal freedom from interference. EB tresos AutoCore OS does not guarantee that any specific component executes on time, nor that a component gets enough access to the CPU to perform its functions on time. If you need to detect or prevent interference in the time domain, you must use a timing protection module in combination with an independent time source, such as an external watchdog monitor. EB tresos AutoCore OS has support for measuring time intervals that can be used by such a module.

EB tresos AutoCore OS does not provide freedom from interference in the communication domain. If you transmit safety-related information over unsafe channels, you must use checksums or similar mechanisms to detect corruption and data loss.

3.3. Interaction with the surrounding system

EB tresos AutoCore OS manages the executable software components of the surrounding system by means of task and ISR objects as specified by AUTOSAR.

In this document, task and ISR objects are referred to collectively as *executable objects* or simply as *executables*. Each executable can use the features of the microcontroller subject to defined privileges and restrictions.

Each executable is bound to a *core* that is specified by the configuration. The executables that are bound to the same core execute concurrently, but not simultaneously, subject to the priority rules defined by AUTOSAR. Executables that are bound to different cores can execute simultaneously regardless of their relative priorities.

Apart from the startup and shutdown phases, EB tresos AutoCore OS is modeless. The state of the system at any time is defined by the states of the executables that are active. The surrounding system is responsible for maintaining any operating modes that are required for the system. The surrounding system also switches the system to a safe state if a safety-critical error is detected.

The software components can interact with EB tresos AutoCore OS by using the public API as defined in the EB tresos AutoCore OS documentation [ASCOS_USERGUIDE].

EB tresos AutoCore OS reports errors and protection faults to the surrounding system by means of three hook functions defined by the AUTOSAR specification: `ErrorHook()`, `ProtectionHook()` and `ShutdownHook()`. Each of these hook functions is called directly by the OS and therefore runs in privileged mode without memory protection. EB tresos AutoCore OS calls the hook function *on the core on which the error is detected.* This means that the hook functions can execute simultaneously on two or more cores. Your implementation of these hook functions must therefore use suitable mutual exclusion mechanisms to ensure the consistency of common data. Interrupt locks are not effective for this purpose.

In addition to the global hook functions, each *Os application* can provide its own hook functions. The OS calls these functions subject to the rules specified by AUTOSAR.

3.4. Outstanding anomalies

The EB tresos AutoCore OS release notes [ASCOS_RELNOTES] contains deviations from the AUTOSAR standard. You can find information about all known issues for a particular version of EB tresos AutoCore OS in EB tresos AutoCore OS known problems [ASCOS_KNOWNPROBLEMS] corresponding to your release. It is available from EB's delivery server and is updated regularly.

3.5. Change control

Contact the EB Product Support and Customer Care Team to initiate a change request. You can find contact details on page 2 of this document.

3.6. Identification of files in EB tresos AutoCore OS

The header files of EB tresos AutoCore OS begin with the prefix `Os_`. The hardware-independent source files of EB tresos AutoCore OS that are compiled and linked as part of the application are also prefixed with `Os_`. The hardware-independent files that are intended to be placed into OS libraries are prefixed with `kern-` or `lib-`. The hardware-dependent files are prefixed with an identifier representing the CPU family.

EB tresos AutoCore OS ships with a number of board-specific functions, for example to initialize the clock generator. These functions are intended as example implementations. They are sufficient to permit EB tresos AutoCore OS to be tested on an evaluation board provided by the hardware vendor.

The board functions are not part of EB tresos AutoCore OS itself and are not developed to satisfy any particular project requirements. If you choose to use the board functions in either unmodified form or as a basis for your own implementation, you must ensure that the functions fulfill your requirements at the desired integrity level.

3.7. Robustness of EB tresos AutoCore OS

To understand how to use EB tresos AutoCore OS correctly, you must understand its limitations as well as its features. This section describes what EB tresos AutoCore OS does not do and provides information about its robustness in various circumstances.

3.7.1. What EB tresos AutoCore OS does not do

EB tresos AutoCore OS does not provide freedom from interference. If you use the features of EB tresos AutoCore OS as part of your freedom-from-interference arguments, you must verify that the features are configured correctly for your application and that they actually provide the freedom from interference that you claim.

EB tresos AutoCore OS does not guarantee that an interrupted executable will ever be resumed. Termination of an executable or a complete Os application by the `ProtectionHook()` and a shutdown caused by the detection of a serious fault are among the reasons why the EB tresos AutoCore OS makes no guarantee of resumption.

If any executable erroneously makes an unintentional but formally valid service request, then the request will be performed by EB tresos AutoCore OS. An example of such an erroneous request is a task activation that is permitted, but takes place at the wrong time. The OS has no way to distinguish between intentional and unintentional service requests. Such unintentional service requests can be detected by the application using external control flow monitoring or similar mechanisms.

3.7.2. Robustness against hardware faults

EB tresos AutoCore OS is susceptible to hardware faults. If your requirements specify robustness against hardware faults you must choose hardware with suitable fault detection mechanisms for the required safety integrity level such as dual redundant processors (e.g. *lockstep*), memory with *error detection and correction codes* (ECC) etc.

3.7.3. Robustness against systematic software errors

EB tresos AutoCore OS is susceptible to systematic software errors in its own code. Although EB tresos AutoCore OS contains some checks of the validity of its internal variables, it cannot be guaranteed that a systematic error in the EB tresos AutoCore OS is detected.

To guard against this risk, EB tresos AutoCore OS is developed using the practices required for Automotive SPICE.

3.7.4. Robustness against configuration errors

EB tresos AutoCore OS is sensitive to configuration errors. EB tresos AutoCore OS accepts a wide variety of possible configurations and has only limited support to detect invalid configurations. In particular, many configurations that are incorrect for a given system may be valid from the EB tresos AutoCore OS's point of view.

EB tresos AutoCore OS has some configuration parameters that disable error checking features and thus affect its robustness to errors. The required configuration of these parameters is given in Chapter 6, “Required configuration settings“.

3.7.5. Robustness against resource conflicts

EB tresos AutoCore OS manages shared resources of the processor (including its internal registers) such that no conflicts take place, provided that all of your software complies with the hardware or compiler's ABI with respect to factors such as register use, saving, and restoring. For functions that are written in C, this is the responsibility of the compiler and you must use appropriate compiler options to ensure compliance. You must verify that functions written in other languages, in particular your own assembly language functions, comply with the C calling conventions.

Unless otherwise stated in the [ASCOS_ARCHNOTES], the use of floating point computations is only supported in task contexts.

EB tresos AutoCore OS requires exclusive use of the parts of the interrupt controller that control the operation of interrupts on the cores on which EB tresos AutoCore OS is configured to run. You must ensure that your application does not directly access these parts of the interrupt controller.

If you share a peripheral such as a timer with EB tresos AutoCore OS you must ensure that the other components that use the peripheral cannot adversely affect EB tresos AutoCore OS.

3.7.6. Robustness against interrupt overload

EB tresos AutoCore OS itself is resistant to interrupt overload. Its nested, priority levelling design means that it can only process one request at a given priority level at any time. Provided that sufficient stack is configured for each ISR and hook function nested interrupts cannot cause an overflow of EB tresos AutoCore OS's stack.

This robustness, however, does not provide any protection for the system as a whole from interrupt overload. Although EB tresos AutoCore OS implements the arrival rate limiting feature of AUTOSAR, it is still possible for rapidly-occurring interrupts to prevent EB tresos AutoCore OS from giving processor time to lower-priority tasks and ISRs. It is assumed that any safety implications resulting from such situations are detected by an external watchdog.

3.7.7. Robustness against input errors

EB tresos AutoCore OS does not accept any data input from external sources.

Incorrect parameter values and other errors resulting from calls to EB tresos AutoCore OS services are detected by the OS and reported by calling `ErrorHook()`. The error checking for pointer parameters is limited; you must verify that your software always passes the address of a valid variable of the correct type to APIs that accept pointer parameters. See Section 4.4.2.1, “Services with RefType parameters” for more information.

4. Using EB tresos AutoCore OS safely

Before you use EB tresos AutoCore OS with a specific processor, you must:

- Read and understand the reference manuals for the microcontroller family and the MCU that you use, including the document errata list if applicable. Pay particular attention to those sections that are relevant for the hardware features that you use.
- Read and understand the hardware vendor's safety manual for the MCU that you use, including the document errata list if applicable.
- Fulfill all safety verification criteria that are stated in the safety manual and are relevant for the safe operation of your product. Safety verification measures that are often requested include:
  - Reading back peripheral registers after writing.
  - When an interrupt is signalled, verifying that the peripheral has requested the interrupt.
  - Detecting delayed and missing interrupt signals.

  The above list is not exhaustive.
- Read and understand the hardware errata lists. Analyze each erratum to determine how it affects the safety of your product and whether there are mitigations that you need to apply.

The hardware features used by EB tresos AutoCore OS are listed in [ASCOS_ARCHNOTES].

Each release of EB tresos AutoCore OS is qualified for use with a particular MCU derivative. The exact identifier and revision of the derivative is stated in the EB tresos AutoCore OS quality statement [ASCOS_QSTATEMENT]. Do not use a given release of EB tresos AutoCore OS with any derivative other than specified.

4.1. Applicability of this document

This document does not refer to a specific version of EB tresos AutoCore OS. It is applicable for EB tresos AutoCore OS versions 6.1.x.

4.2. Prerequisites

Before you use EB tresos AutoCore OS you must partition the software of the surrounding system into *Os objects* (*tasks* and *ISRs*), that shall be scheduled according to the AUTOSAR standard. The configuration of EB tresos AutoCore OS depends heavily on your partitioning decisions.

4.3. Using EB tresos Studio

This section describes how to configure EB tresos AutoCore OS using EB tresos Studio.

4.3.1. Configuring the operating system module

To configure the Os module, see the EB tresos AutoCore OS documentation [ASCOS_USERGUIDE].

- Ensure that you assign the correct priorities to your tasks.
- Ensure that you assign the correct interrupt levels to your ISRs.
- Ensure that you select the `OsTaskSchedule` attribute appropriately for each task.
- Ensure that you configure the correct resources for each task and ISR that uses resources. This is particularly important for internal resources because they are automatically acquired when a task runs. This means that an error check for missing resources at run-time is not possible.

[Configuration.Specification]
Verify that you configure EB tresos AutoCore OS to implement your system requirements correctly.

4.3.2. Generating the operating system module

To create a set of generated source and header files for EB tresos AutoCore OS from your configuration, follow the EB tresos AutoCore OS documentation [ASCOS_USERGUIDE].

4.4. Implementing your system

The details on how to implement the additional software in your system are beyond the scope of this document. There are rules that you must follow to ensure that you use EB tresos AutoCore OS safely. The rules are given in the following sections.

4.4.1. Avoid symbols reserved by EB tresos AutoCore OS

All symbols that start with `OS_`, `Os_`, `os_`, `E_OS_`, `OSServiceId_`, `OSError_`, `OSTICKSPERBASE_`, `OSMAXALLOWEDVALUE_` and `OSMINCYCLE_` are reserved for the exclusive use of EB tresos AutoCore OS.

In addition to these symbol prefixes, the symbols listed in Table A.1, “Identifiers reserved for use by AUTOSAR Os” are reserved for use by EB tresos AutoCore OS as specified by [ASR_19_11SPEC]. The symbols listed in Table A.2, “Identifiers reserved for use by EB tresos AutoCore OS” are reserved for implementation by EB.

You must not define any of the symbols mentioned above in your application. This prohibition includes the naming of objects in your configuration, because the identifiers for the configuration objects are defined as C macros in the generated files.

[Api.ReservedWords]
Verify that your application does not define or declare identifiers that are reserved by EB tresos AutoCore OS.

4.4.2. EB tresos AutoCore OS services

The table in Appendix B, “Safety of OS services“ lists all the services provided by EB tresos AutoCore OS that have been analysed for the purpose of this safety application guide. Many of these services are defined by the AUTOSAR standard, but the list includes some EB vendor-specific extensions.

The second column of this table indicates whether you can call the service in an application with a required safety integrity level.

EB tresos AutoCore OS contains many internal functions that are not intended to be called directly by tasks or ISRs. You must not directly call any function that is not listed in Appendix B, “Safety of OS services“.

[Api.ASIL]
Verify that your application only calls services that are specified in Appendix B, “Safety of OS services“ as safe to use for your application's safety integrity level.

[Api.Undocumented]
Verify that your application does not directly call functions or macros defined in EB tresos AutoCore OS that are not listed in Appendix B, “Safety of OS services“.

4.4.2.1. Services with RefType parameters

EB tresos AutoCore OS checks the values of the parameters that are passed to system services whenever possible. If the error checking is not disabled, out-of-range parameters are reported by means of the `ErrorHook()` callout.

There are some APIs that require a reference to a variable in which the API shall place the requested information. It is not possible to verify the validity of these parameters. Only minimal validity checks are possible for non-trusted applications. You must verify that a correct parameter has been passed to every invocation of these APIs.

Our experience showed that the AUTOSAR convention of declaring `XxxRefType` to be a type that is a reference to a variable declared as `XxxType` is a common cause of programming errors such as inappropriate typecasting and endianness. It is important to declare a variable `XxxType foo;` and to pass its address

(`&foo`) to the function. Declaring a variable of the type `XxxRefType` and passing it directly is an error unless the variable was previously initialized with the address of a `XxxType` variable.

The recommended practice is to declare a local automatic variable of the correct type. If you use a variable with global lifetime, you must ensure that it is not subject to race conditions caused by concurrent access.

[Api.RefParam.GetAlarm]
For every call to `GetAlarm(Id, Tick)` in your application, verify that the parameter `Tick` is the address of a variable that is declared with the type `TickType`.

[Api.RefParam.GetAlarmBase]
For every call to `GetAlarmBase(Id, Info)` in your application, verify that the parameter `Info` is the address of a variable that is declared with the type `AlarmBaseType`.

[Api.RefParam.GetCounterValue]
For every call to `GetCounterValue(Id, Value)` in your application, verify that the parameter `Value` is the address of a variable that is declared with the type `TickType`.

[Api.RefParam.GetElapsedValue]
For every call to `GetElapsedCounterValue(Id, PreviousValue, Value)` or `GetElapsedValue(Id, PreviousValue, Value)` in your application, verify that the parameters `PreviousValue` and `Value` are both addresses of variables that are declared with the type `TickType`.

[Api.RefParam.GetEvent]
For every call to `GetEvent(Id, Event)` in your application, verify that the parameter `Event` is the address of a variable that is declared with the type `EventMaskType`.

[Api.RefParam.GetScheduleTableStatus]
For every call to `GetScheduleTableStatus(Id, out)` in your application, verify that the parameter `out` is the address of a variable that is declared with the type `ScheduleTableStatusType`.

[Api.RefParam.GetTaskID]
For every call to `GetTaskID(Id)` in your application, verify that the parameter `Id` is the address of a variable that is declared with the type `TaskType`.

[Api.RefParam.GetTaskState]
For every call to `GetTaskState(Id, State)` in your application, verify that the parameter `State` is the address of a variable that is declared with the type `TaskStateType`.

[Api.RefParam.GetApplicationState]
For every call to `GetApplicationState(Id, State)` in your application, verify that the parameter `State` is the address of a variable that is declared with the type `ApplicationStateType`.

In addition to the AUTOSAR services listed above, EB tresos AutoCore OS provides some vendor-specific services to obtain information about the system. These services are:

- `OS_GetIsrMaxRuntime()`
- `OS_GetTaskMakRuntime()`
- `OS_GetTimeStamp()`

- `OS_DiffTime32()`
- `OS_TimeSub64()`
- `OS_GetStackInfo()`
- `OS_GetCurrentStackArea()`

When calling each of these services, if the parameter is of type `XXX *` (with or without a `const` qualifier), you must pass the address of a variable that you declared as `XXX`. For example:

```
os_timestamp_t currentTime;
OS_GetTimeStamp(&currentTime);
```

[Api.RefParam.BaseType]
For every call to an EB tresos AutoCore OS vendor-specific API that accepts one or more explicit pointer parameters, verify that you pass the address of a validly declared variable of the correct base type to each pointer parameter.

The APIs `OS_GetTaskMaxRuntime()` and `OS_GetIsrMaxRuntime()` are imlemented as simple functions that do not enter the OS. If they detect an error, the function's return value is the only indication. `ErrorHook()` is not called.

[Api.MaxRuntime.NoErrorHook]
If your application calls `OS_GetTaskMaxRuntime()` or `OS_GetIsrMaxRuntime()`, verify that your application ensures that the return value is `E_OK` before using the content of the `out` variable.

4.4.2.2. Trusted functions

The `CallTrustedFunction()` API accepts a parameter that it passes on to the trusted function. The specified type name for the parameter is `TrustedFunctionParameterRefType`, which means it is a pointer. However, there is no associated `TrustedFunctionParameterType` associated with it, because the data that is passed to a trusted function depends on the function itself. The actual type of `TrustedFunctionParameterRefType` is therefore `(void *)`.

The anonymous nature of the parameter means that the OS cannot perform any range checks on the value of the parameter. This in turn means that the trusted function is responsible for ensuring that the parameter is valid.

[Api.RefParam.CallTrustedFunction]
For every call to `CallTrustedFunction(FunctionIndex, FunctionParams)` in your application, verify that the parameter `FunctionParams` is the address of a variable that is declared with the type that is expected by your trusted function.

[Api.RefParam.CallTrustedFunction.Implementation]
For every trusted function `TRUSTED_<name>(FunctionIndex, FunctionParams)` in your application, verify that the parameter `FunctionParams` is checked for validity before it is dereferenced.

4.4.2.3. Fast interrupt locking

Here *fast interrupt locking* refers to the interrupt locking which uses a hardware mechanism directly.

Disabling all interrupts by directly toggling the processor's *interrupt enable* flag prevents the operating system from regaining control of the processor. This type of interrupt locking is used by the `EB_FAST_LOCK` mechanism that you can configure for `SchM` and `Rte` exclusive areas.

If you call an OS API while interrupts are disabled, there is no guarantee that interrupts remain disabled while the OS is handling your request. For example, if a higher-priority task becomes active during the API, it will execute *with interrupts enabled* before control returns to your calling task.

[Api.Use.EBFastLock]
If your application uses a direct hardware method for disabling and enabling interrupts, verify that you do not call OS APIs while interrupts are disabled. This prohibition includes the AUTOSAR interrupt locking services.

4.4.3. Other considerations

There are some characteristics of EB tresos AutoCore OS and of operating systems in general that can have unexpected results. It is impossible to know every possible use case, but this section describes issues that are commonly encountered.

4.4.3.1. Stack sizes

In EB tresos AutoCore OS, each task uses a stack that the OS reserves for the task when the task starts executing. When the task terminates its stack is no longer needed and the OS releases it so that another task can use it. The stack sharing is determined statically by the configuration. Only tasks that cannot preempt each other can share the same stack. Extended tasks never share their stacks with other tasks.

The OS APIs use the caller's stack (`OsTrappingKernel=FALSE`) or the kernel stack (`OsTrappingKernel=TRUE`). Interrupt handling and ISRs use the kernel stack on most hardware.

Stack overflow is a common problem that is not always detected during development. The reason for this is that the worst-case stack depth might only be encountered under rare circumstances. This is particularly true for the kernel stack, where the worst-case stack depth only occurs during the worst-case interrupt nesting. This characteristic means that empirical estimation of stack depth almost always yields too low a value.

A stack overflow might only cause a failure under special circumstances. For example, if a task overflows its stack and writes into the stack of a higher-priority task, a fault will only occur if the higher-priority task preempts the first task during the time that the excess stack is in use.

It is important to calculate the worst case stack depth for each function, taking into account the functions that it calls. This is a recursive process. You must program the calculated depth for each top-level function (task, ISR, and hook function) into your configuration. For more information, see Section 6.3, “Stack sizes”.

You must always enable the stack monitoring feature in EB tresos AutoCore OS. The stack monitoring provided by the OS has limitations. It is possible for a task to exceed its stack limit without overwriting the end marker. Depending on your system safety requirements, you might need to implement additional monitoring measures. For example, a background stack monitoring function can calculate the amount of unused stack for each stack and report a fault if the amount falls below a predefined threshold. You can use the EB tresos AutoCore OS APIs `OS_GetUnusedIsrStack()` and `OS_GetUnusedTaskStack()` to calculate the amount of unused stack for the kernel and task stacks.

`OS_GetStackInfo()` is normally used indirectly by means of one of the wrapper functions (`OS_GetUnusedTaskStack()` etc.). If you call `OS_GetStackInfo()` directly you must ensure that you call it correctly.

`OS_GetUsedTaskStack()` and `OS_GetUnusedTaskStack()` use `OS_GetStackInfo()` to obtain the information. If you call the task APIs but pass an out-of-range parameter, the API might return information that is not useful, but without reporting an error.

[Implementation.StackDepth]
Verify that your stack calculations take into account the worst-case stack depth. This is not necessarily the deepest function-call nesting.

[Implementation.Recursion]
Verify that your application contains no uncontrolled recursion and that your stack calculations take into account the worst-case recursion depth.

[Implementation.Monitoring]
Verify that your application implements stack level monitoring consistent with your system safety requirements.

[Implementation.GetStackInfo.PresetStackPointer]
If you call the API `OS_GetStackInfo()` directly, verify that you initialize the `stackPointer` field of the `out` structure before calling the API.

[Implementation.GetStackInfo.Parameter]
If you call the API `OS_GetStackInfo()` directly, verify that the parameter that you pass to identify the task is one of:

- A valid task ID.
- `OS_IsrToTOI(i)`, where `i` is a valid ISR ID.
- `OS_IsrToTOI(OS_NULLISR)`.

[Implementation.GetTaskStack.Parameter]
If you call `OS_GetUsedTaskStack()` or `OS_GetUnusedTaskStack()`, verifiy that the task ID parameter is a valid task ID.

4.4.3.2. Hook functions

EB tresos AutoCore OS supports a range of callout functions called *hook functions*. Except for the hook functions that you configure for non-trusted Os applications, all hook functions run with the same privileges as the OS.

If you use EB tresos AutoCore OS in a multi-core configuration, hook functions can execute concurrently on two or more cores. You must design and implement your hook functions so that they cannot interfere with themselves as a result of concurrent execution.

[Implementation.HookFunctions.ASIL]
Verify that you design and implement your hook functions according to the standards appropriate for the safety integrity level of your system.

[Implementation.HookFunctions.Concurrency]
Verify that your implementation of hook functions takes into account their concurrency characteristics.

4.4.3.3. Interrupt locks, resources and spinlocks

On a single-core system, interrupt locks and resources are often used for synchronizing tasks and ISRs to prevent data inconsistency caused by concurrent access.

AUTOSAR specifies that `GetResource()` shall report an error if the caller has a higher priority than the ceiling priority of the resource. If you configure a resource with a ceiling priority that is too low, the error check detects the configuration error. Calling `GetResource()` has no effect on the priority of the caller if the caller is confgured with the ceiling priority of the resource. It is therefore common to omit the calls to `GetResource()` from the highest-priority tasks. The omission defeats the error check and, if a configuration error is present, could result in inconsistent data or data corruption.

On a multi-core system, interrupt locks and resources only have a local effect on the core on which they are used. If data is shared between cores, you need to use an inter-core locking mechanism such as an AUTOSAR spinlock object. However, a pure spinlock does not prevent preemption on the local core. This means that a task can occupy a spinlock for the duration of its preemption, thus potentially blocking other cores. Therefore it is important to nest spinlock occupancy within a suitable local lock. If your operating system supports it, the spinlock parameter `OsSpinlockLockMethod` causes an interrupt lock to be acquired and released automatically. If this parameter is not available, you must implement the locking yourself.

Spinlocks can cause deadlock if not used correctly. Attempting to acquire a spinlock that is already occupied on the same core is avoided by using interrupt locking. EB tresos AutoCore OS supports the `OsSpinlockLockMethod` so you can set it to `LOCK_ALL_INTERRUPTS` to fulfill this requirement. Deadlock between cores can be avoided by:

- Avoiding nested acquisition of spinlocks.

- Enforcing a strict order on the nested acquisition of spinlocks.

AUTOSAR specifies a method to enforce the order of spinlock acquisition. This is configured by means of the `OsSpinlockSuccessor` parameter. The method does not guarantee freedom from deadlocks if a task holding a spinlock can be preempted.

AUTOSAR does not generally forbid calling OS APIs while the caller occupies a spinlock. If the API does not detect an error for another reason, the API might cause a higher-priority task or ISR to execute, which in turn might use a spinlock in a way that could cause deadlock. Even if no error occurs, the spinlock could be occupied for longer than expected and could cause a task on another core to be blocked for longer than expected.

[Implementation.Resource.AlwaysAcquire]
If you configure resource locks, verify that every task or ISR acquires the resource before it access the data that is protected by the resource, even if the resource has no effect on the priority of the task or ISR.

[Implementation.Spinlocks.NoPreempt]
Verify that your application locks interrupts on the local core for the duration of every spinlock occupancy.

[Implementation.Spinlocks.Deadlock]
Verify that your application uses spinlocks in a deadlock-free manner.

[Implementation.Spinlocks.OtherApi]
Verify that your application does not call OS APIs while occupying a spinlock. The exceptions to this rule are other locking APIs, `ShutdownOS` and `ShutdownAllCores()`.

4.4.3.4. Internal resources and non-preemptable tasks

The AUTOSAR resource mechanism prevents unexpected priority inversion by means of the *priority ceiling protocol* (PCP). The variety of PCP that AUTOSAR specifies requires that a task that acquires a resource inherits the ceiling priority at the point of aquisition and reverts to its former priority when it releases the resource. This blocks any other task that has a lower priority than the ceiling until the resource is released. Thus any task that also needs to acquire the resource is prevented from executing.

The theory behind PCP assumes that the low-priority task occupies the resource for the short time that is actually needed and releases immediately when the task is not needed anymore.

The disadvantage of using a resource rather than a direct hardware lock is the CPU time overhead. If blocking all interrupts is not desired, a resource is the only mechanism available.

A common way that is used to reduce the locking overhead is to configure an internal resource. A task that is configured to use an internal resource conceptually acquires the resource before it starts running and relinquishes the resource on termination and entering the waiting state. The task can also release the resource temporarily by calling `Schedule()`. By the time `Schedule()` returns to its caller, the resource was released and re-acquired.

In EB tresos AutoCore OS an internal resource has no run-time overhead. The disadvantage is that the resource remains occupied for the entire execution of the task, whether or not the resource is actually needed at any time. This mechanism essentially causes a priority inversion, where a low-priority task can prevent a high-priority task from running. You must analyze the timing and priorities in your system to ensure that this behavior does not have an adverse impact on a safety requirement.

A non-preemptive task (configured with `OsTaskSchedule=NON`) exhibits similar behavior. Conceptually, it acquires `RES_SCHEDULER` on entry. You must therefore treat non-preemptable tasks in the same way as tasks that use internal resources.

A task configured with `OsTaskCallScheduler=NO` shall not call `Schedule()` as it can lead to stack corruption.

[Implementation.InternalResource]
If you configured one or more tasks to use internal resources or as non-preemptable tasks, verify that the tasks so configured cannot cause a safety requirement to be violated.

[Implementation.ScheduleApi]
For each task that you configure with `OsTaskCallScheduler=NO`, verify that the task does not call `Schedule()` API.

4.4.3.5. Timely execution

EB tresos AutoCore OS does not have any features to monitor the timely execution of tasks. Faults in the timer hardware and the interrupt controller can cause task activations to be postponed or omitted completely. Unexpected behavior of the application, such as excessive interrupt or resource blocking, can delay the execution of tasks that have been activated.

The timing protection features of EB tresos AutoCore OS can help to ensure that there is enough time to execute the safety-relevant tasks. This protection does not extend to detecting lack of activity or delayed activity. You must implement measures to monitor time-triggered activites.

[Implementation.Watchdog]
Verify that your system has appropriate monitoring of execution, deadlines, and other time-related characteristics to ensure that your safety requirements are met.

4.4.3.6. Synchronization of schedule tables

EB tresos AutoCore OS implements the standard AUTOSAR APIs and mechanisms for synchronizing schedule tables with an external time source. These APIs and mechanisms have some characteristics to be aware of when designing and analyzing your system.

You are not required to synchronize your schedule tables, but if you do, you need to be aware of the potential problems.

The API `SyncScheduleTable()` assumes that the global time parameter passed by the caller is sufficiently accurate at the point in the code where the current schedule table time is calculated. Clearly, there is an error due to the execution time of the caller and the API, but this should be irrelevant in most cases.

If an interrupt or preemption occurs between obtaining the global time in the application and calculating the current local time in the API, a large error can be introduced, which can cause the schedule table to start adjusting itself unnecessarily. In extreme cases the error might force the schedule table into the asynchronous state.

It is necessary to prevent preemptions while calculating global time and calling `SyncScheduleTable()`. You can achieve this by acquiring a resource that is shared by a high-priority ISR. Alternatively, you can perform the whole calculation and synchronization in a high-priority ISR. Using OS interrupt locking APIs does not work because AUTOSAR forbids the use of APIs while interrupts are locked. Locking interrupts directly on the hardware is also not recommended because the OS does not guarantee to honour the lock during API execution. See Section 4.4.2.3, “Fast interrupt locking” for more information.

The synchronization mechanism works by shortening or lengthening the time between expiry points along the schedule table. For each expiry point, you can configure the amount of time that is available for adjustments. See Section 6.5, “Schedule table synchronization parameters” for information about configuring synchronization safely.

If you are using two schedule tables, only one of which is synchronized, the synchronization process might start temporal interference between the tasks that are activated by the schedule tables. This can occur when expiry points that occur with sufficient delay under normal circumstances drift close together as a result of minor differences between local and global time.

[Implementation.StSynchronization.Preemption]

If you use schedule table synchronization, verify that no preemption or interrupt can occur in the time between obtaining global time in your application and calculating local time in the `SyncScheduleTable()` API.

[Implementation.StSynchronization.Interference]

If you use schedule table synchronization, verify that the synchronization of a schedule table cannot cause temporal interference with the unsynchronized activities in your system.

4.4.3.7. Optimized schedule API

EB tresos AutoCore OS provides an optimized interface to the `Schedule()` API to avoid the overhead of an unnecessary system call. The full error handling takes place if the `Schedule()` API is called. However the wrapper functions make use of the current task pointer without verifying that it is valid. If you call the optimized API from any context other than a task, the API might dereference a NULL pointer in the context of the task.

[Implementation.OptimizedSchedule]

If you use the `OS_ScheduleIfNecessary()`, `OS_ScheduleIfWorthwhile()`, `OS_IsScheduleNecessary()`, or `OS_IsScheduleWorthwhile()` APIs, verify that you only call them from a task context.

4.4.3.8. API returns to caller unexpectedly

The AUTOSAR APIs `TerminateTask()`, `ChainTask()`, `TerminateApplication(<self>,...)`, `ShutdownOS`, and `ShutdownAllCores()` do not return to the caller if they are successful. This means that a return value of `E_OK` is not possible.

If the API detects an error it does not perform the operation. Instead it returns an error code to the caller. In most cases the error is caused by a programming error. In the case of `ChainTask()` the error code could be `E_OS_LIMIT`, which is caused by the state of the system.

You must ensure that any code that is executed after a return from one of these APIs cannot cause a safety requirement to be violated.

In many cases, returning from the top-level function of the task or ISR is the only way to handle this type of error, because the OS is then required to terminate the task or ISR and to free any locks that were acquired.

[Implementation.UnexpectedReturn]

Verify that your application cannot violate a safety requirement if one of the following APIs unexectedly returns to its caller:

- `TerminateTask()`
- `ChainTask()`
- `TerminateApplication()`
- `ShutdownOS()`
- `ShutdownAllCores()`

4.4.3.9. Using absolute counter values

The AUTOSAR APIs `SetAbsAlarm()` and `StartScheduleTableAbs()` cause the first expiry of the alarm or schedule table to take place at a specified value of the underlying counter. If the counter has already counted past that value, the first expiry is delayed for a full counter round. Depending on the hardware and the configuration, the delay could be anything from a few milliseconds to many years.

It is usually preferable to use `SetRelAlarm()` or `StartScheduleTableRel()`. However, there are some valid use cases for the absolute variants.

There is an Rte parameter that configures an `ABSOLUTE` first expiry.

[Implementation.AbsoluteCounterValue]

If your application calls `SetAbsAlarm()` or `StartScheduleTableAbs()`, either:

- verify that the counter can never tick past the specified first expiry before the alarm gets set, or
- verify that the application can tolerate a delay of one full wrap-around of the counter.

4.4.3.10. Using the timestamp feature

EB tresos AutoCore OS provides a timestamp feature that you can use to measure the passage of time. The feature is implemented using a high resolution 64-bit counter that can never overflow. If the hardware provides a 64-bit counter, the OS uses it. If not, the OS constructs a 64-bit timer from one of the available hardware counters.

The no-overflow condition only holds if the counter starts at or near zero after a cold reset. If a hardware timestamp is used, the OS does not initialize it at startup, thus permitting continuous measurement of time over a software-triggered reset. If the timer is not in a defined state after a reset, you must ensure that it is initialized before starting the OS.

If the timestamp is derived from a hardware counter this property is ensured by the initialization to zero of the variables, so you do not need to initialize the timer.

The APIs `OS_TimeSub64()` and `OS_DiffTime32()` produce unreliable results if the "previous" time is greater than the "now" time. The APIs do not check for errors.

[Implementation.TimeStamp]

If your application uses the timestamp feature, verify that the value of the timestamp at start-up is sufficiently low that it cannot overflow during the longest permitted up-time of the system.

[Implementation.TimeStampElapsed]

If your application uses the timestamp difference APIs, verify that the `oldTime` parameter is less than or equal to (i.e. was obtained chronologically earlier than) the `newTime` parameter.

4.4.3.11. Accessing interrupt controller registers

EB tresos AutoCore OS requires exclusive use of the parts of the interrupt controller that control the operation of interrupts on the cores on which EB tresos AutoCore OS is configured to run.

[Api.InterruptSourceRegisters]

Verify that the application does not access the interrupt controller registers which are being used by OS.

4.5. Implications of the compiler settings

The compiler, the version, and the exact options that EB supports for a particular release are specified in the corresponding EB tresos AutoCore OS quality statement [ASCOS_QSTATEMENT]. EB tresos AutoCore OS is tested extensively using these options.

WARNING **Tested compilers**

Ensure that you use the same compiler at the same version and with the same settings that EB used for verification. You can find this information in the quality statement.

If you want to use a different compiler, a different version of the compiler or other options than specified in the quality statement, sufficient and adequate verification measures must be performed. Contact EB Product Support and Customer Care Team in this case.

[Qualification]

Verify that the version of EB tresos AutoCore OS that you are using is qualified

- as "ready for manufacturing" (RfM)
- with the microcontroller derivative that you use
- with the compiler version and settings that you use

4.6. Steps outside the scope of EB

The architecture, design, implementation, verification, and validation of your system are outside the scope of this guide. It is the system designer's responsibility to ensure that the system as a whole is designed in accordance with the requirements and that the configuration of EB tresos AutoCore OS correctly implements the design.

5. Assumed requirements

The main use case of EB tresos AutoCore OS is the integration together with safety related SWC applications in one ECU.

Id:	OperatingSystem.Interference
Doctype:	`reqspec1`
Status:	APPROVED
Version:	1
Description:	EB tresos AutoCore OS shall not interfere with any SWC application.
Safety class:	ASIL-B
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:	OperatingSystem.Verification
Doctype:	`reqspec1`
Status:	APPROVED
Version:	1
Description:	EB tresos AutoCore OS shall be verified according to ISO26262 First Edition with ASIL-B
Safety class:	ASIL-B
Safety rationale:	EB tresos AutoCore OS shall be capable of being integrated in ECUs together with ASIL-B SWC applications.
Needs coverage of:	-

6. Required configuration settings

This chapter provides the verification criteria that you must use to verify that you have configured your OS correctly for use in a safety-related system.

The containers and parameters named in this chapter are those that you can see in the EB tresos Studio GUI. Some parameters have boolean values and are represented graphically by a checkbox. When a verification criterion specifies that such a parameter shall be `TRUE`, the checkbox must be checked. Conversely, `FALSE` means that the checkbox must not be checked.

When the term *enable* is used in this chapter, it refers to the ability to edit the parameter or container in the GUI. It does not apply to the value of any parameter.

6.1. The OsOS container

The container `OsOS` contains several standard AUTOSAR parameters that control how the OS handles errors. The OS is not free from interference if it cannot detect and report errors.

[Config.OsOS.OsStackMonitoring]
Verify that the parameter `OsOS/OsStackMonitoring` is set to `TRUE`.

[Config.OsOS.OsStatus]
Verify that the parameter `OsOS/OsStatus` is set to `EXTENDED`.

[Config.OsOS.OsUseGetServiceId]
Verify that the parameter `OsOS/OsUseGetServiceId` is set to `TRUE`.

[Config.OsOS.OsUseParameterAccess]
Verify that the parameter `OsOS/OsUseParameterAccess` is set to `TRUE`.

[Config.OsOS.OsHooks.OsErrorHook]
Verify that the parameter `OsOS/OsHooks/OsErrorHook` is set to `TRUE`.

[Config.OsOS.OsHooks.OsProtectionHook]
Verify that the parameter `OsOS/OsHooks/OsProtectionHook` is enabled and set to `TRUE`.

[Config.OsOS.OsHooks.OsShutdownHook]
Verify that the parameter `OsOS/OsHooks/OsShutdownHook` is set to `TRUE`.

The container `OsOS` also contains several EB vendor-specific parameters that enhance the AUTOSAR error checking and reporting. In addition:

- `OsSourceOptimization` eliminates a lot of dead code by removing configuration queries using macros with fixed values. This can result in a lot of compiler warnings of the form "condition is always true/false". If you set `OsSourceOptimization` to `TRUE`, you must build a new OS library for every configuration change.

- `OsProtection` disables hardware protection features to make debugging possible on some older microcontrollers. There are no microcontrollers in current use that need this feature.
- `OsStartupChecks` can have an adverse effect on the startup time of your system. If your system fails to meet its required start-up time, set the parameter to `FALSE` for the production build and compare that the generated configuration files `Os_config.h` and `Os_user.h` with the previous build. Verify that the files are identical except for the changes expected due to the parameter change.

[Config.OsOS.OsExtra_Runtime_Checks]
Verify that the parameter `OsOS/OsExtra_Runtime_Checks` is set to `TRUE`.

[Config.OsOS.OsStartupChecks]
Verify that the parameter `OsOS/OsStartupChecks` is set to `TRUE` unless the checks cause a startup-time requirement to be violated.

[Config.OsOS.OsSourceOptimization]
Verify that the parameter `OsOS/OsSourceOptimization` is set to `TRUE`, unless your quality requirements forbid its effects.

[Config.OsOS.OsProtection]
Verify that the parameter `OsOS/OsProtection` is set to `ON`.

[Config.OsOS.OsUseLastError]
Verify that the parameter `OsOS/OsUseLastError` is set to `TRUE`.

6.2. The OsAutosarCustomization container

The `OsAutosarCustomization` container holds some parameters that extend the error checking and reporting capabilities of EB tresos AutoCore OS. The container is disabled by default because the parameters it contains affect the AUTOSAR conformance.

The extended error checking is required for safe operation of the system. You must therefore enable the container and verify that you configured the parameters correctly as described below.

[Config.OsOS.OsAutosarCustomization]
Verify that the container `OsOS/OsAutosarCustomization` is enabled.

[Config.OsOS.OsAutosarCustomization.OsExceptionHandling]
Verify that the parameter `OsOS/OsAutosarCustomization/OsExceptionHandling` is set to `TRUE`.

[Config.OsOS.OsAutosarCustomization.OsErrorHandling]
Verify that the parameter `OsOS/OsAutosarCustomization/OsErrorHandling` is set to `FULL`.

[Config.OsOS.OsAutosarCustomization.OsStrictServiceProtection]
Verify that the parameter `OsOS/OsAutosarCustomization/OsStrictServiceProtection` is set to `TRUE`.

[Config.OsOS.OsAutosarCustomization.OsCat1DirectCall]

Verify that the parameter `OsOS/OsAutosarCustomization/OsCat1DirectCall` is set to `FALSE`.

[Config.OsOS.OsAutosarCustomization.OsInterruptLockingChecks]

Verify that the parameter `OsOS/OsAutosarCustomization/OsInterruptLockingChecks` is set to `AUTOSAR`.

[Config.OsOS.OsAutosarCustomization.OsCallAppErrorHook]

Verify that the parameter `OsOS/OsAutosarCustomization/OsCallAppErrorHook` is set to `VIA_WRAPPER`.

[Config.OsOS.OsAutosarCustomization.OsCallAppStartupShutdownHook]

Verify that the parameter `OsOS/OsAutosarCustomization/OsCallAppStartupShutdownHook` is set to `VIA_WRAPPER`.

[Config.OsOS.OsAutosarCustomization.OsPermitSystemObjects]

Verify that the parameter `OsOS/OsAutosarCustomization/OsPermitSystemObjects` is set to `FALSE`.

[Config.OsOS.OsAutosarCustomization.OsUserTaskReturn]

Verify that the parameter `OsOS/OsAutosarCustomization/OsUserTaskReturn` is set to `KILL_TASK`.

Setting the `OsCallIsr=DIRECTLY` improves the performance of ISR handling, but prevents the OS from terminating ISRs. If it becomes necessary to terminate a running ISR for any reason other than normal completion, the OS converts the termination request into a shutdown. If this behavior is not compatible with your safety requirements verify that the parameter is set to `VIA_WRAPPER`.

[Config.OsOS.OsAutosarCustomization.OsCallIsr]

If you configure the parameter `OsOS/OsAutosarCustomization/OsCallIsr` to `DIRECTLY`, verify that a reaction of `SHUTDOWN` when attempting to terminate an ISR does not violate a safety requirement of your system.

6.3. Stack sizes

You must configure sufficient stack for each executable that you configure. As described in Section 4.4.3.1, “Stack sizes”, the stack size must take into account the worst case stack depth, including all functions that are called directly. It is usually not sufficient to estimate worst-case stack depth by using empirical methods.

[Config.OsTask.OsStacksize]

For each task that you configured, verify that you configured a sufficiently large value for `OsTask.OsStacksize`.

[Config.OsIsr.OsStacksize]

For each ISR that you configured, verify that you configured a sufficiently large value for `OsIsr.OsStacksize`.

[Config.OsApplication.OsAppErrorHookStack]
For each Os application that you configured with `OsAppErrorHook=TRUE`, verify that you configured a sufficiently large value for `OsAppErrorHookStack`.

[Config.OsApplication.OsAppStartupHookStack]
For each Os application that you configured with `OsAppStartupHook=TRUE`, verify that you configured a sufficiently large value for `OsAppStartupHookStack`.

[Config.OsApplication.OsAppShutdownHookStack]
For each Os application that you configured with `OsAppShutdownHook=TRUE`, verify that you configured a sufficiently large value for `OsAppShutdownHookStack`.

[Config.OsApplication.OsTrustedFunctionStacksize]
For each trusted function that you configured, verify that you configured a sufficiently large value for its `OsTrustedFunctionStacksize`.

6.4. The OsSpinlock object

`OsSpinlock` objects are intended to provide mutual exclusion analagous to `OsResource` objects, but which have an effect on all cores.

You must not use `OsSpinlock` objects if the OS is configured to run on one core.

AUTOSAR spinlock objects do not automatically lock interrupts unless configured to do so. Using spinlocks without interrupt locks can result in deadlock. To prevent deadlock, you must configure every spinlock with the highest possible lock level.

[Config.OsSpinlock]
If your system only runs on a single core, verify that you did not configure any `OsSpinlock` objects.

[Config.OsSpinlock.OsSpinlockLockMethod]
For each spinlock that you configured, verify that the parameter `OsSpinlockLockMethod` is set to `LOCK_ALL_INTERRUPTS`.

6.5. Schedule table synchronization parameters

EB tresos AutoCore OS implements the standard AUTOSAR APIs and mechanisms for synchronizing schedule tables with an external time source. These APIs and mechanisms have some characteristics to be aware of when designing and analyzing your system. The characteristics are described in Section 4.4.3.6, “Synchronization of schedule tables”.

You are not required to synchronize your schedule tables, but if you do, you need to be aware of the potential problems.

The synchronization mechanism works by shortening or lengthening the time between expiry points along the schedule table. You can configure the amount of time that is available for adjustments individually for each expiry point. The parameters governing the synchronization are:

- `OsScheduleTableMaxShorten` configures the maximum amount by which the time interval before this expiry point can be reduced.
- `OsScheduleTableMaxLengthen` configures the maximum amount by which the time interval before this expiry point can be increased.

Both of these parameters are in the `OsScheduleTblAdjustableExpPoint` container of the `OsScheduleTableExpiryPoint` object.

The value of the `OsScheduleTableMaxShorten` parameter cannot be greater than the offset of the expiry point relative to its immediate predecessor. However, the `OsScheduleTableMaxLengthen` is constrained only by the duration of the schedule table.

When the time between two expiry points is reduced, there is a possibility that an expiry point will occur before the activities from previous expiry points are complete. This can cause the maximum activation limit for a task to be exceeded (`E_OS_LIMIT`). You must ensure that your application can tolerate the reduced time between expiry points.

When the time between two expiry points is increased, there is a possibility that the timing constraints of your application are violated for the duration of the synchronization. You must ensure that your application can tolerate the increased time between expiry points.

If the schedule table is determined to be synchronized within the tolerance during a call to `SyncScheduleTable()`, the algorithm will reduce the remaining lateness by reducing delays, until there is no remaining adjustment. Conversely, if the schedule table is determined to be early, the remaining adjustment is reduced by increasing the delays.

If the schedule table is determined to be out of synchronization during a call to `SyncScheduleTable()`, the algorithm will attempt to regain synchronization in the smallest number of steps. If you configure the synchronization parameters `OsScheduleTableMaxLengthen` and `OsScheduleTableMaxShorten` asymmetrically with `OsScheduleTableMaxLengthen` much greater than `OsScheduleTableMaxShorten`, then it could be possible to achieve synchronization by taking a few large steps rather than by taking many small steps. If this occurs, then a single round of the schedule table could be stretched out to the time normally taken for two rounds. If you configure this type of asymmetry, you must ensure that your system can tolerate the lateness of task activations that could occur.

[Config.OsScheduleTblAdjustableExpPoint.OsScheduleTableMaxShorten]

If you use schedule table synchronization, verify that your system can tolerate the worst-case time reduction between expiry points without violating a safety requirement.

[Config.OsScheduleTblAdjustableExpPoint.OsScheduleTableMaxLength]

If you use schedule table synchronization, verify that your system can tolerate the worst-case time increase between expiry points without violating a safety requirement.

Appendix A. Reserved identifiers

The identifiers listed in the following table are defined by the AUTOSAR Os specification. You must not use them 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.

ActivateTask	StartScheduleTableAbs	ACCESS
ActivateTaskAsyn	StartScheduleTableRel	E_OK[a]
AllowAccess	StartScheduleTableSynchron	E_OS_*[b]
CallTrustedFunction	StopScheduleTable	INVALID_ISR
CancelAlarm	SuspendAllInterrupts	INVALID_OSAPPLICATION
ChainTask	SuspendOSInterrupts	INVALID_TASK
CheckISRMemoryAccess	SyncScheduleTable	NO_ACCESS
CheckObjectAccess	TerminateApplication	NO_RESTART
CheckObjectOwnership	TerminateTask	OBJECT_ALARM
CheckTaskMemoryAccess	TryToGetSpinlock	OBJECT_COUNTER
ClearEvent	WaitEvent	OBJECT_ISR
ClearPendingInterrupt	WritePeripheral8	OBJECT_RESOURCE
ControlIdle	WritePeripheral16	OBJECT_SCHEDULETABLE
DisableAllInterrupts	WritePeripheral32	OBJECT_TASK
DisableInterruptSource	AccessType	OSDEFAULTAPPMODE
EnableAllInterrupts	AlarmBaseRefType	OSMAXALLOWEDVALUE
EnableInterruptSource	AlarmBaseType	OSMINCYCLE
GetActiveApplicationMode	AlarmType	OSTICKDURATION
GetAlarmBase	ApplicationStateType	OSTICKSPERBASE
GetAlarm	ApplicationStateRefType	PRO_IGNORE
GetApplicationID	ApplicationType	PRO_SHUTDOWN
GetApplicationState	AppModeType	PRO_TERMINATEAPPL
GetCoreID	AreaIdType	PRO_TERMINATEAP-PL_RESTART
GetCounterValue	CoreIdType	PRO_TERMINATETASKISR
GetCurrentApplicationID	CounterType	READY
GetElapsedValue	EventMaskRefType	

- GetElapsedCounterValue
- GetEvent
- GetISRID
- GetNumberOfActivatedCores
- GetResource
- GetScheduleTableStatus
- GetSpinlock
- GetTaskID
- GetTaskState
- IncrementCounter
- ModifyPeripheral8
- ModifyPeripheral16
- ModifyPeripheral32
- NextScheduleTable
- OSErrorGetServiceId
- ReadPeripheral8
- ReadPeripheral16
- ReadPeripheral32
- ReleaseResource
- ReleaseSpinlock
- ResumeAllInterrupts
- ResumeOSInterrupts
- Schedule
- SetAbsAlarm
- SetEvent
- SetEventAsyn
- SetRelAlarm
- SetScheduleTableAsync
- ShutdownAllCores
- ShutdownOS
- StartCore
- StartNonAutosarCore
- EventMaskType
- IdleModeType
- ISRType
- MemorySizeType
- MemoryStartAddressType
- ObjectAccessType
- ObjectTypeType
- OSServiceIdType
- PhysicalTimeType
- ProtectionReturnType
- ResourceType
- RestartType
- ScheduleTableStatusRefType
- ScheduleTableStatusType
- ScheduleTableType
- SpinlockIdType
- StatusType [a]
- TaskRefType
- TaskStateRefType
- TaskStateType
- TaskType
- TickRefType
- TickType
- TimeInMicrosecondsType
- TrustedFunctionIndexType
- TrustedFunctionParameter-RefType
- TryToGetSpinlockType
- DeclareAlarm
- DeclareEvent
- DeclareResource
- DeclareTask
- RES_SCHEDULER
- RESTART
- RUNNING
- SCHEDULETABLE_NEXT
- SCHEDULETABLE_RUN-NING_AND_SYNCHRONOUS
- SCHEDULETABLE_RUN-NING
- SCHED-ULETABLE_STOPPED
- SCHEDULETABLE_WAITING
- SUSPENDED
- TotalNumberOfCores
- WAITING
- ALARMCALLBACK
- ISR
- TASK
- OSMEMORY_IS_EXE-CUTABLE
- OSMEMORY_IS_READABLE
- OSMEMORY_IS_STACKS-PACE
- OSMEMORY_IS_WRITE-ABLE

▶ StartOS		

[a] `E_OK` and `StatusType` can be defined by an AUTOSAR module other than the OS if it adheres to the requirements of the OSEK/VDX binding specification [OSEKOS142BIND].

[b] All AUTOSAR and OSEK/VDX error code identifiers.

Table A.1. Identifiers reserved for use by AUTOSAR Os

▶ ISR1	▶ GetStackInfo	
▶ AdvanceCounter	▶ getUnusedIsrStack	
▶ IAdvanceCounter	▶ getUnusedTaskStack	
	▶ getUsedIsrStack	
	▶ getUsedTaskStack	
	▶ stackCheck	

Table A.2. Identifiers reserved for use by EB tresos AutoCore OS

Appendix B. Safety of OS services

Service name	Integrity level	Remarks
ActivateTask	ASIL-B	
ActivateTaskAsyn	ASIL-B	
CallTrustedFunction	ASIL-B	The integrity category refers to calling the trusted function and passing the parameter. No statement is made about the user-defined functionality of the trusted function that is called.
AllowAccess	ASIL-B	
CancelAlarm	ASIL-B	
ChainTask	ASIL-B	
CheckIsrMemoryAccess	ASIL-B	
CheckObjectAccess	ASIL-B	
CheckObjectOwnership	ASIL-B	
CheckTaskMemoryAccess	ASIL-B	
ClearEvent	ASIL-B	
ClearPendingInterrupt	ASIL-B	
ControlIdle	ASIL-B	
DisableAllInterrupts	ASIL-B	
DisableInterruptSource	ASIL-B	
EnableAllInterrupts	ASIL-B	
EnableInterruptSource	ASIL-B	
GetActiveApplicationMode	ASIL-B	
GetAlarm	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetAlarmBase	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetApplicationID	ASIL-B	
GetApplicationState	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetCoreID	ASIL-B	
GetCounterValue	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.

Service name	Integrity level	Remarks
GetCurrentApplicationID	ASIL-B	
GetElapsedCounterValue	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetEvent	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetISRID	ASIL-B	
GetNumberOfActivatedCores	ASIL-B	
GetResource	ASIL-B	
GetScheduleTableStatus	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetSpinlock	ASIL-B	
GetTaskID	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
GetTaskState	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
IncrementCounter	ASIL-B	
NextScheduleTable	ASIL-B	
OSErrorGetServiceId	ASIL-B	
OSError_<svc>_<param>	ASIL-B	
ReleaseResource	ASIL-B	
ReleaseSpinlock	ASIL-B	
ResumeAllInterrupts	ASIL-B	
ResumeOSInterrupts	ASIL-B	
Schedule	ASIL-B	
SetAbsAlarm	ASIL-B	
SetEvent	ASIL-B	
SetEventAsyn	ASIL-B	
SetRelAlarm	ASIL-B	
SetScheduleTableAsync	ASIL-B	
ShutdownOS	ASIL-B	
StartCore	ASIL-B	
StartNonAutosarCore		Not implemented.

Service name	Integrity level	Remarks
StartOS	ASIL-B	
StartScheduleTableAbs	ASIL-B	
StartScheduleTableRel	ASIL-B	
StartScheduleTableSynchron	ASIL-B	
StopScheduleTable	ASIL-B	
SuspendAllInterrupts	ASIL-B	
SuspendOSInterrupts	ASIL-B	
SyncScheduleTable	ASIL-B	
TerminateApplication	ASIL-B	
TerminateTask	ASIL-B	
TryToGetSpinlock	ASIL-B	
WaitEvent	ASIL-B	
OS_DisableInterruptSource	ASIL-B	Vendor-specific.
OS_EnableInterruptSource	ASIL-B	Vendor-specific.
AdvanceCounter	Do not use	Vendor-specific; for backwards compatibility. Use `IncrementCounter()` instead.
IAdvanceCounter	Do not use	Vendor-specific; for backwards compatibility. Use `IncrementCounter()` instead.
getUnusedIsrStack	Do not use	Vendor-specific; for backwards compatibility. Use `Os_GetUnusedIsrStack()` instead.
getUnusedTaskStack	Do not use	Vendor-specific; for backwards compatibility. Use `Os_GetUnusedTaskStack()` instead.
getUsedIsrStack	Do not use	Vendor-specific; for backwards compatibility. Use `Os_GetUsedIsrStack()` instead.
getUsedTaskStack	Do not use	Vendor-specific; for backwards compatibility. Use `Os_GetUsedTaskStack()` instead.
stackCheck	Do not use	Vendor-specific; for backwards compatibility. Use `Os_StackCheck()` instead.
OS_DiffTime32	ASIL-B	Vendor-specific.
OS_GetCurrentStackArea	ASIL-B	Vendor-specific. See Section 4.4.2.1, “Services with RefType parameters”.
OS_GetErrorInfo	ASIL-B	Vendor-specific.

Service name	Integrity level	Remarks
OS_GetIsrMaxRuntime	ASIL-B	Vendor-specific. See Section 4.4.2.1, “Services with RefType parameters”.
OS_GetScheduleTableStatus	ASIL-B	Vendor-specific. See Section 4.4.2.1, “Services with RefType parameters”.
OS_GetStackInfo	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
OS_GetTaskState	ASIL-B	Vendor-specific. See Section 4.4.2.1, “Services with RefType parameters”.
OS_GetTimeStamp	ASIL-B	Vendor-specific. See Section 4.4.2.1, “Services with RefType parameters”.
OS_GetUnusedIsrStack	ASIL-B	Vendor-specific.
OS_GetUnusedTaskStack	ASIL-B	Vendor-specific.
OS_GetUsedIsrStack	ASIL-B	Vendor-specific.
OS_GetUsedTaskStack	ASIL-B	Vendor-specific.
OS_IsScheduleNecessary	ASIL-B	Vendor-specific.
OS_IsScheduleWorthwhile	ASIL-B	Vendor-specific.
OS_ScheduleIfNecessary	ASIL-B	Vendor-specific.
OS_ScheduleIfWorthwhile	ASIL-B	Vendor-specific.
OS_SimTimerAdvance	Do not use	Vendor-specific; intended for test purposes only.
OS_SimTimerSetup	Do not use	Vendor-specific; intended for test purposes only.
OS_StackCheck	ASIL-B	Vendor-specific.
OS_TimeGetHi	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
OS_TimeGetLo	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.
OS_TimeSub64	ASIL-B	See Section 4.4.2.1, “Services with RefType parameters”.

Table B.1. Safety status of OS services in EB tresos AutoCore OS

Appendix C. ASIL-B methods

The following table lists the recommended methods for ASIL-B and how they are achieved in EB tresos AutoCore OS.

Number	Method [a]	Reasoning
1a	Enforcement of low complexity (++)	Cyclomatic complexity of the code is restricted by adherence to the HIS source code metrics.
1b	Use of language subsets (++)	Safe programming style is enforced by adherence to MISRA-C:2012.
1c	Enforcement of strong typing (++)	Use of ANSI-C and MISRA-C standards.
1d	Use of defensive implementation techniques (+)	The EB tresos AutoCore OS implementation used defensive programming techniques as specified by the AUTOSAR requirements.
1e	Use of established design principles (+)	Design is derived from AUTOSAR requirements. Well established implementation techniques / algorithms used.
1f	Use of unambiguous graphical representation (++)	Graphical representations are not used in design document.
1g	Use of style guides (++)	The EB tresos AutoCore OS development follows a coding style guideline.
1h	Use of naming conventions (++)	The EB tresos AutoCore OS uses naming conventions in addition to the AUTOSAR convention.
7a	Natural language (++)	Design is written in natural language with some code snippets and mathematical calculations.
7b	Informal notations (++)	Design is written in natural language with some code snippets and mathematical calculations.
7c	Semi-formal notations (++)	Design is written in natural language with some code snippets and mathematical calculations.
7d	Formal notations (+)	Design is written in natural language with some code snippets and mathematical calculations.
8a	One entry and one exit point in subprograms and functions (++)	Checked by MISRA-C:2012 Rules 15.5 and 21.4.
8b	No dynamic objects or variables, or else online test during their creation (++)	Static OS configuration is used. Use of dynamic allocation is prohibited by MISRA-C:2012 Rule 20.4.
8c	Initialization of variables (++)	Checked by MISRA-C:2012 Rules 9.x.

Number	Method [a]	Reasoning
8d	No multiple use of variable names (++)	Checked by MISRA-C:2012 Rules 5.6-5.9, 21.2.
8e	Avoid global variables or else justify their usage (+)	Access to global variables for non-trusted applications can be restricted using memory protection.
8f	Limited use of pointers (+)	MISRA-C:2012 Rules 18.x restrict the use of pointers.
8g	No implicit type conversions (++)	MISRA-C:2012 Rules 10.x,11.x
8h	No hidden data flow or control flow (++)	Verified during source code review as one of the review checklist items.
8i	No unconditional jumps (++)	Checked by MISRA-C:2012 Rule 15.1.
8j	No recursions (+)	Checked by MISRA-C:2012 Rule 17.2.
9a	Walkthrough (+)	Reviews with a predefined checklist of objectives are used instead.
9b	Inspection (++)	► Reviews adhere to a review process. Revision controlled code is reviewed by one person considering defined review objectives. ► The EB tresos AutoCore OS code base is proven in use and observed by field monitoring.
9c	Semi-formal verification (+)	Not performed.
9d	Formal verification (o)	Not a recommendation for ASIL-B.
9e	Control flow analysis (+)	The goals of a control flow analysis (according to [ISO/IEC 61508-7:2010] chapter C.5.9) are: ► Detect inaccessible code: achieved by decision coverage using Check_C ► Detect knotted code: achieved by applying MISRA rules for ► not using goto and continue statements. ► restrict the use of the return statement to at most one such statement per function, which must be at the end of the function. ► restrict the use of the break statement: At most one break statement per loop is permitted.
9f	Data flow analysis (+)	Not performed.

Number	Method [a]	Reasoning
9g	Static code analysis (++)	The software unit implementation is statically analyzed against the coding guidelines using MISRA checker tool.
9h	Semantic code analysis (+)	Not performed.
10a	Requirements-based test (++)	All requirements are derived from the SWS and are tested or linked to a specification object which provides coverage for it. Traceability is ensured by requirement analysis tool and is checked during RFI and release process.
10b	Interface test (++)	Module interfaces tested in line with SWS requirements.
10c	Fault injection test (+)	Config patches are used to inject faults in the test configuration.
10d	Resource usage test (+)	Not applicable.
10e	Back-to-back comparison test between model and code, if applicable (+)	Not applicable.
11a	Analysis of requirements (++)	The test specification is derived from requirements based on SWS.
11b	Generation and analysis of equivalence classes (++)	Not applicable as AutoCore OS does not perform computation on input data, therefore no equivalence classes can be defined. For testing the AutoCore OS discrete input values are used.
11c	Analysis of boundary values (++)	Not applicable as the EB tresos AutoCore OS does not perform computation on input data, therefore no data ranges with boundaries can be defined. For testing the EB tresos AutoCore OS discrete input values are used.
11d	Error guessing(+)	We have error guessing tests but they are not formally documented as error guessing ones.
12a	Statement coverage (++)	Stronger method used (12b).
12b	Branch coverage (++)	Decision (C1 branch) coverage is performed to measure code coverage for the EB tresos AutoCore OS.
12c	MC/DC (Modified Condition/Decision Coverage)(+)	Not performed.
16a	Hardware-in-the-loop (+)	Not applicable.

Number	Method [a]	Reasoning
16b	Electronic control unit network environments(++)	Not applicable.
16c	Vehicles (++)	Not applicable.

[a]
- ++ indicates a highly recommended method
- + indicates a recommended method
- o indicates a not recommended method

Table C.1. Applied ASIL-B recommended methods

Appendix D. Document configuration information

This document has been 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/autosar/asc_Os/trunk/doc/public/SAG.

Filename	Revision
../../project/fragments/cross_references/Cross_References.xml	33830
../../project/fragments/entities/OS-entities.ent	29552
../../project/fragments/glossary/Glossary.xml	33823
ApiSafety.xml	33823
AppliedMethods.xml	33823
Assumed_Requirements.xml	29552
AutoCore_OS_safety_application_guide.xml	33823
ConfigCriteria.xml	33823
Description.xml	33823
DocumentInformation.xml	29552
History.xml	33830
ReservedWords.xml	33823
SafeUse.xml	33823

Glossary

core	A core is a single processing unit of a microcontroller, containing registers, arithmetic/logic unit and other processing hardware. A multi-core microcontroller has two or more cores that can execute software simultaneously.
error-correction code	Error correction codes (ECC) are redundant information stored next to the data itself so that the hardware can detect and in some cases correct errors.
exception	Processor-internal event, e.g. division by zero. The current program flow is interrupted and continued at the address which is associated with the specific exception. Usually accompanied by a switch to *privileged mode*. Exceptions are commonly used to indicate a fault detected by hardware, e.g. a detected memory protection violation. In general, exceptions have a higher priority than *interrupts* and can therefore interrupt the interrupt handling.
interrupt	The interrupt request signals the CPU to stop the program execution at the current point and to continue the execution at the address which is associated with the specific interrupt request.
interrupt service routine	An interrupt service routine (ISR) is an *Os object*. An ISR is the code that is executed to handle an *interrupt* request. ISRs can be global or belong to an *Os application*.
lockstep	Lockstep means that hardware components are replicated and perform the identical function for each clock cycle. The output of the hardware components is compared. If the output diverges, the error is detected and appropriate hardware-specific actions are performed.
non-privileged mode	CPU mode with restricted access rights. Opposite of *privileged mode*.
Os application	An Os application is a group of *Os objects*. All objects of an Os application belong to one entity and can share data among each other and have memory areas with common write access.
Os object	An Os object is a data structure managed by the OS. This includes the runnable code of *tasks*, *ISR*s, and *exception* handlers.
privileged mode	CPU mode with elevated rights. In this mode, it is allowed to alter the memory and register protection. Opposite of *non-privileged mode*.
task	A task is an *Os object* that is started, scheduled and terminated by the operating system.

Bibliography

[ASCOS_ARCH-NOTES] *EB tresos AutoCore OS architecture notes*, AutoCore_OS_architecture_notes_<architecture>.pdf

[ASCOS_KNOWN-PROBLEMS] *EB tresos AutoCore OS known problems*, EB_tresos_Autocore_known_issues-AutoCoreOs_<version>_COMMON_SRC.pdf

[ASCOS_QSTATEMENT] *EB tresos AutoCore OS quality statement*, EB_tresos_Autocore-quality_statement-AutoCoreOS-<version>-<derivative>.pdf

[ASCOS_REL-NOTES] *EB tresos AutoCore OS release notes*, AutoCore_OS_release_notes_<architecture>.pdf

[ASCOS_SAG_-VR_1] *Verification report for the EB tresos® AutoCore OS safety application guide*, SAG_Review_1.xls

[ASCOS_SAG_-VR_2] *Verification report 2 for the EB tresos® AutoCore OS safety application guide*, SAG_-Review_2.txt

[ASCOS_USER-GUIDE] *EB tresos AutoCore OS documentation*, AutoCore_OS_documentation.pdf

[ASR_19_11SPEC] AUTOSAR:*Specification of Operating System*, Version R19-11

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

[ISO26262-3_1ST] *INTERNATIONAL STANDARD ISO 26262-3*: Road vehicles - Functional safety - Part 3: Concept phase, 2011

[OSEKOS142BIND] OSEK/VDX:*OSEK Binding Specification*, Version 1.4.2

http://portal.osek-vdx.org/files/pdf/specs/binding142.pdf