Elektrobit

EB tresos® Safety OS user's guide

for TriCore family

Date: 2020-11-24, Document version V2.3.71, 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

Table of Contents

1. Document history

Version	Date	State	Description
V2.0.0	2015-05-13	DRAFT	Initial version for MK2.0.
V2.0.1	2015-04-28	DRAFT	Rebuilt for prototype release without changes.
V2.0.2	2015-06-18	DRAFT	Rebuilt for prototype release without changes.
V2.0.3	2015-08-06	DRAFT	▶ Replaced MK_eid_TrustedFunctionCallFromWrong-Core by MK_eid_TFCallFromWrongCore and MK_eid_-TrustedFunctionCallFromWrongContext by MK_eid_-TFCallFromWrongContext. ▶ Imported TRICORE supplement from MK1.1 without change, and added some dummy configuration sec-tions to satisfy references.
V2.0.4	2015-08-11	DRAFT	Rebuilt for release without changes.
V2.0.5	2015-11-17	DRAFT	▶ Added a section describing the MK_GetExceptionInfo() API. ▶ Rebuilt for development drop.
V2.0.6	2016-02-17	DRAFT	Rebuilt for release without changes.
V2.0.7	2016-02-24	DRAFT	Rebuilt for release without changes.
V2.1.0	2016-06-02	PROPOSED	▶ Marked `MK_ElapsedMicroseconds()` as deprecat-ed. ▶ Added missing, reworked and removed some existing API descriptions. ▶ Made several corrections and additions to chapters 7, 8, 9 and 10.
V2.1.1	2016-06-06	RELEASED	UG for TRICORE walkthrough was passed [UGTRICORERMK2R1]
V2.1.2	2016-06-21	PROPOSED	▶ Added descriptions of default memory regions. ▶ Added new configuration constants and removed no longer supported ones. ▶ Added `MK_InitSyncHere()`. ▶ Updated trusted function configuration.
V2.1.3	2016-06-22	RELEASED	UG for TRICORE walkthrough was passed [UGTRICORERMK2R1]

Version	Date	State	Description
V2.1.4	2016-06-28	PROPOSED	Corrected the description and error handling of `MK_EnableInterruptSource()` and `MK_DisableInterruptSource()`.
V2.1.4	2016-06-29	RELEASED	Minor changes in the glossary. UG for TRICORE walkthrough was passed [UGTRICORERMK2R1].
V2.1.5	2016-08-02	PROPOSED	▶ Incorporated rework from TE review. ▶ Corrected example in chapter 8.4.2.3. ▶ Corrected memory region-map array in chapter 9.4.8. ▶ Corrected memory region configuration array in chapter 9.4.9.
V2.1.5	2016-08-04	RELEASED	UG for TRICORE walkthrough was passed [UGTRICORERMK2R1].
V2.1.6	2016-10-05	PROPOSED	▶ Added `culpritApplicationId` field to `mk_protectioninfo_s` ▶ Added more information about the `culprit` field of `mk_protectioninfo_s`
V2.1.7	2016-11-16	PROPOSED	▶ Rework of description of `culpritApplicationId` field of `mk_protectioninfo_s`.
V2.1.8	2016-11-29	PROPOSED	▶ Removed configuration parameter `MK_CFG_Cx_TF_MEMPART`. ▶ Refined description of `MK_CFG_HWMASTERCOREINDEX`. ▶ Removed configuration parameter `LOCK_VIA_RESOURCES`.
V2.1.8	2016-12-05	RELEASED	UG for TRICORE walkthrough was passed [UGTRICORERMK2R1].
V2.1.9	2017-01-25	PROPOSED	▶ Added `coreIndex` parameter to `MK_IRQCFG`. ▶ Added descriptions of interrupt and exception handling on TriCore. ▶ [RH850] Updated CPU-family supplement for microkernel 2.0 and RH850F1H.
V2.1.10	2017-02-13	PROPOSED	▶ Added limits to the service and error code arguments of `MK_ReportError`. ▶ [RH850] Rework of CPU-family supplement.
V2.1.10	2017-02-14	PROPOSED	▶ [RH850] CPU-family supplement passed walkthrough [UGRH850RR1].

Version	Date	State	Description
V2.1.11	2017-02-16	PROPOSED	▶ Updated deviations against [AUTOSAROS42SPEC].
V2.1.12	2017-02-22	PROPOSED	▶ Rework after document walkthrough.
V2.1.13	2017-02-24	PROPOSED	▶ Additional rework after document walkthrough.
V2.1.13	2017-02-24	RELEASED	▶ UG for TRICORE walkthrough was passed [UGTRICORERMK2R1].
V2.1.14	2017-03-27	PROPOSED	▶ [RH850] Added notes on using the CPU-family specific interrupt locking instructions. ▶ Added description for `MK_InterruptLocking-Hook()` and related subfunctions
V2.1.15	2017-03-29	PROPOSED	▶ Rework after document walkthrough.
V2.1.15	2017-03-29	RELEASED	▶ [RH850] UG passed walkthrough [UGRH850RR1].
V2.1.16	2017-04-26	PROPOSED	▶ [ARM] Updated CPU-family supplement for microkernel 2.0 and S32V234AA32. ▶ Added background information about MMU support in the microkernel to the generic part.
V2.1.17	2017-04-28	PROPOSED	▶ Rework after document walkthrough.
V2.1.17	2017-05-02	RELEASED	▶ [ARM] UG passed walkthrough [UGARMRR1].
V2.1.18	2017-05-10	PROPOSED	▶ Initial version of UG for PA.
V2.1.19	2017-05-16	PROPOSED	▶ [RH850] Semantics of the fpu parameter of MK_HW-PS_INIT() changed.
V2.2	2017-05-30	PROPOSED	▶ Integrated simple schedule table user's guide. ▶ Interrupt locking/unlocking callouts changed. ▶ Add non-preemptive ISRs.
V2.2.1	2017-06-06	PROPOSED	▶ Rework after document walkthrough.
V2.2.2	2017-06-09	PROPOSED	▶ Rework after document walkthrough.
V2.2.2	2017-06-14	RELEASED	▶ [RH850] UG passed walkthrough [UGRH850RR1].
V2.2.3	2017-07-07	PROPOSED	▶ Update of the parameter type of API `MK_Condition-alGetResource()`. ▶ Update description of `StartCore`, `MK_AsyncActi-vateTask`, `MK_AsyncSetEvent` and `ShutdownAll-Cores` concerning single-core processors. ▶ Add AUTOSAR deviation concerning spinlocks on sin-gle-core processors.

Version	Date	State	Description
			▶ [ARM] Add CPU-family supplement for AR1642.
V2.2.4	2017-07-10	PROPOSED	▶ Rework after document walkthrough.
V2.2.4	2017-07-11	RELEASED	▶ [ARM] UG passed walkthrough [UGARMRR1].
V2.3	2017-08-01	PROPOSED	▶ Initial version for ARM64 ▶ Correct parameter type of API `MK_ElapsedMicroseconds()`. ▶ Correct parameter types of API `MK_MulDiv()`.
V2.3.1	2017-08-03	PROPOSED	Rework after document walkthrough.
V2.3.2	2017-08-03	PROPOSED	Rework after document walkthrough.
V2.3.3	2017-08-04	PROPOSED	▶ [ARM64] Update of processor state mapping. ▶ [ARM64] Add architecture specific configuration macro.
V2.3.3	2017-08-07	RELEASED	[ARM64] UG passed walkthrough [UGARM64RR1].
V2.3.4	2017-08-09	PROPOSED	[ARM64] Incorporate rework from TE review.
V2.3.4	2017-08-09	RELEASED	[ARM64] UG passed walkthrough [UGARM64RR1].
V2.3.5	2017-08-17	PROPOSED	▶ [ARM] Added architecture specific description for RCARV3MCR7. ▶ [ARM] Added supplementary for KEIL ARM Compiler.
V2.3.6	2017-08-17	PROPOSED	Rework after document walkthrough.
V2.3.7	2017-08-17	PROPOSED	Rework after document walkthrough.
V2.3.7	2017-08-17	RELEASED	[ARM] UG passed walkthrough [UGARMRR1].
V2.3.8	2017-09-07	PROPOSED	▶ [ARM] Added architecture specific description for RCARM3CR7.
V2.3.9	2017-09-08	PROPOSED	Rework after document walkthrough.
V2.3.9	2017-09-08	RELEASED	[ARM] UG passed walkthrough [UGARMRR1].
V2.3.10	2017-11-10	DRAFT	▶ [CORTEXM] Add CPU-family specific sections for CORTEXM. Currently empty.
V2.3.11	2018-02-05	PROPOSED	▶ [TRICORE] Add TC38X.
V2.3.12	2018-02-07	PROPOSED	Rework after document walkthrough.
V2.3.12	2018-02-09	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.13	2018-02-13	PROPOSED	Adjusted interrupt names and added startup conditions for R-Car derivatives.
V2.3.14	2018-02-14	PROPOSED	Rework after document walkthrough.

Version	Date	State	Description
V2.3.14	2018-02-14	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.15	2018-03-15	PROPOSED	[CORTEXM] Added initial version of the supplementary information for CORTEXM.
V2.3.16	2018-03-15	PROPOSED	Rework after document walkthrough.
V2.3.17	2018-03-16	PROPOSED	Rework after document walkthrough.
V2.3.17	2018-03-16	RELEASED	[CORTEXM] UG passed walkthrough [UGCORTEXMRR1].
V2.3.18	2018-04-18	PROPOSED	[ARM] Added description of memory protection with an ARMv8-R MPU. [ARM] Added S32S247-specific information.
V2.3.19	2018-04-27	PROPOSED	Rework after document walkthrough.
V2.3.19	2018-05-02	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.20	2018-05-07	PROPOSED	[RH850] Updated the processor mode settings.
V2.3.20	2018-05-07	RELEASED	[RH850] UG passed walkthrough [UGRH850RR1].
V2.3.21	2018-05-25	PROPOSED	[ARM] Updated description about memory region attributes for ARMv7 MPUs. [ARM] Updated description about interrupt handling for RCARM3CR7 and RCARV3MCR7.
V2.3.22	2018-06-04	PROPOSED	Rework after document walkthrough.
V2.3.22	2018-06-04	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.23	2018-06-05	PROPOSED	Added `MK_GetPanicExceptionInfo()`. [ARM] Updated `mk_hwexceptioninfo_t`. [CORTEXM] Updated `mk_hwexceptioninfo_t`. [PA] Updated `mk_hwexceptioninfo_t`. [RH850] Updated `mk_hwexceptioninfo_t`. [TRICORE] Updated `mk_hwexceptioninfo_t`.
V2.3.24	2018-06-07	PROPOSED	Rework after document walkthrough.
V2.3.24	2018-06-08	RELEASED	[RH850] UG passed walkthrough [UGRH850RR1].
V2.3.25	2018-06-18	PROPOSED	[RH850] Added supplementary information for RH850D1X.
V2.3.26	2018-06-22	DRAFT	[TRICORE] Added TC22XL and TC23XL.
V2.3.27	2018-06-29	PROPOSED	Replaced the data type `mk_tick_t` used for small time periods with the explicit 32-bit type `mk_uint32_t`.

Version	Date	State	Description
V2.3.28	2018-07-02	PROPOSED	[RH850] Rework for RH850D1X after walkthrough.
V2.3.28	2018-07-02	RELEASED	[RH850] UG passed walkthrough [UGRH850RR1].
V2.3.29	2018-07-03	PROPOSED	[ARM64] Updated and corrected struct description for `MK_-exceptionInfo()`. [ARM64] Updated `MK_REGION_ATTR_DEVICE`/`Attr0`. [ARM64] Updated `mk_hwexceptioninfo_t`.
V2.3.30	2018-07-04	PROPOSED	[PA] Added description of configurable cache inhibit bits.
V2.3.31	2018-07-05	PROPOSED	[PA] Rework after walkthrough.
V2.3.31	2018-07-05	RELEASED	[PA] UG passed walkthrough [UGPARR1].
V2.3.32	2018-07-11	PROPOSED	[TRICORE] Proposed document for TRICORE.
V2.3.33	2018-07-13	PROPOSED	[TRICORE] Rework after walkthrough.
V2.3.34	2018-07-16	PROPOSED	Rework after walkthrough.
V2.3.34	2018-07-16	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.35	2018-07-19	PROPOSED	Rework after TE review.
V2.3.35	2018-07-20	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.36	2018-08-09	PROPOSED	Updated naming schema of MK_SOFTVECTOR_*
V2.3.36	2018-08-10	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.37	2018-10-16	PROPOSED	[ARM64] Added architecture specific description for RCARM3N. [ARM64] Added startup prerequisite for enabled and coher-ent caches.
V2.3.37	2018-10-18	RELEASED	[ARM64] UG passed walkthrough [UGARM64RR1].
V2.3.38	2019-02-05	PROPOSED	[ARM] Added descriptions for RCARV3HCR7. [ARM64] Added descriptions for RCARV3H. [RH850] Added derivative RH850P1HC ▶ Revised section "Memory regions on RH850 proces-sors." ▶ Revised section "Timers on RH850 processors."

Version	Date	State	Description
			▶ Revised section "Simple schedule table on RH850 processors." ▶ Revised section "Multi-core support on RH850 processors." [ARM] Extended description of `mk_hwexceptioninfo_t` by the new member `fpexc`. [ARM] Switched from TMU to Global Timer and Private Timer for execution budgeting and SST, respectively, on RCARM3CR7, RCARV3MCR7 and RCARV3HCR7. Added atomic functions to Table 8.3, “EB-vendor-specific services in EB tresos Safety OS.”. [ARM] Added descriptions for ZUXEVCR5.
V2.3.39	2019-02-06	PROPOSED	[ARM] Rework after document walkthrough.
V2.3.39	2019-02-08	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.40	2019-02-08	PROPOSED	[ARM] Added startup conditions for ZUXEVCR5.
V2.3.40	2019-02-11	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.41	2019-03-01	PROPOSED	[ARM] Added descriptions for RCARM3NCR7.
V2.3.41	2019-03-04	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.42	2019-03-05	PROPOSED	[ARM64] Added descriptions for ZUXEV.
V2.3.42	2019-03-06	RELEASED	[ARM64] UG passed walkthrough [UGARM64RR1].
V2.3.43	2019-03-25	PROPOSED	Added warning note for releases without safety approval.
V2.3.44	2019-03-26	PROPOSED	[ARM] Fixed name and parameter for the function MK_EnableMmu.
V2.3.45	2019-03-29	PROPOSED	Added Cross_References.xml to configuration information.
V2.3.46	2019-04-30	PROPOSED	Update build environment. Update section for releases without safety approval. Update after TE review.
V2.3.47	2019-04-30	PROPOSED	Update after walkthrough.
V2.3.47	2019-04-30	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.48	2019-05-27	PROPOSED	Added description for MK_CFG_COREPAGETABLECONFIG.

Version	Date	State	Description
			[PA] Updated list of memory regions for SPC58XG. [TRICORE] Added derivative TC39xX.
V2.3.49	2019-05-27	PROPOSED	[TRICORE] Reworked the walkthrough findings.
V2.3.50	2019-05-29	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.51	2019-05-29	RELEASED	[PA] UG passed walkthrough [UGPARR1].
V2.3.52	2019-06-17	PROPOSED	[ARM] Added description of `MK_LibCheckInterrupt-Controller()`.
V2.3.52	2019-06-19	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.52	2019-06-26	RELEASED	[RH850] UG passed walkthrough [UGRH850RR1].
V2.3.53	2019-07-16	PROPOSED	[ARM64] Update configuration information.
V2.3.54	2019-07-24	PROPOSED	[ARM] Added descriptions for S6J3300. Rework after TE review.
V2.3.54	2019-07-24	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.55	2019-08-05	RELEASED	[ARM64] UG passed walkthrough [UGARM64RR1].
V2.3.56	2019-08-12	PROPOSED	Added description of Wind River® Diab Compiler stack smashing protection support.
V2.3.57	2019-09-12	PROPOSED	[TRICORE] Reworked the walkthrough findings.
V2.3.58	2019-09-13	PROPOSED	Rework after walkthrough.
V2.3.58	2019-09-13	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.59	2019-09-16	PROPOSED	[TRICORE] Update section on core ID mapping.
V2.3.60	2019-09-17	PROPOSED	[TRICORE] Rework after walkthrough.
V2.3.60	2019-09-17	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1].
V2.3.61	2019-11-20	PROPOSED	Revise usage of core IDs in configuration reference section. [PA] Update section on derivative-specific core ID mapping. [PA] Removed warnings related to the AutoCore OS time stamp API. Rework after TE review. [PA] Fix spelling mistake.

Version	Date	State	Description
V2.3.62	2019-11-20	PROPOSED	Rework after walkthrough.
V2.3.62	2019-11-21	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.63-MicroOs2.0_-TRICORE_PA_-inspected_2019cw35	2019-11-26	PROPOSED	[PA] Rework after walkthrough.
V2.3.63-MicroOs2.0_-TRICORE_PA_-inspected_2019cw35	2019-11-26	RELEASED	[PA] UG passed walkthrough [UGPARR1].
V2.3.63	2020-02-21	PROPOSED	▶ [ARM] Update the timer details for ZUXEVCR5. ▶ Update due to removal of macro MK_HWHASEXCEPTIONINFO.
V2.3.63	2020-02-27	RELEASED	[ARM] UG passed walkthrough [UGARMRR2].
V2.3.64-MicroOs2.0_-TRICORE_inspected_2020cw05	2020-03-20	PROPOSED	Rework after TE review [done in 2.3.61]
V2.3.64-MicroOs2.0_-TRICORE_inspected_2020cw05	2020-03-20	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1]
V2.3.65	2020-05-06	PROPOSED	[common] Removed error E_OS_SERVICEID from SST API functions. [ARM64] Updated the section on ARM64 timers. [ARM64] Added support for panic exception information. [ARM64] Removed support for predefined memory attributes for uncached and write-through cacheable memory. [ARM64] New API `MK_TriggerPeriodicChecks()`.

Version	Date	State	Description
			[TRICORE] Added derivative TC36XD. [TRICORE] Added derivative TC33XL. [TRICORE] Added derivative TC37XT. [PA] Added S32R294.
V2.3.66	2020-05-07	PROPOSED	[PA] Rework after walkthrough.
V2.3.66	2020-05-07	RELEASED	[PA] UG passed walkthrough [UGPARR1].
V2.3.67	2020-05-13	PROPOSED	[TRICORE] Proposed document for TRICORE.
V2.3.67	2020-05-25	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1]
V2.3.68	2020-06-17	PROPOSED	[ARM64] Corrected number of available interrupt priorities for non-secure mode.
V2.3.69	2020-06-18	PROPOSED	[ARM64] Rework after walkthrough.
V2.3.69	2020-06-18	RELEASED	[ARM64] UG passed walkthrough [UGARM64RR1].
V2.3.70	2020-11-13	PROPOSED	[ARM64] Removed description for MK_CFG_-COREPAGETABLECONFIG. [TRICORE] Derivative name alignment.
V2.3.70	2020-11-13	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1]
V2.3.71	2020-11-19	PROPOSED	[TRICORE] Removed limitations for TriCore processors chapter.
V2.3.71	2020-11-24	RELEASED	[TRICORE] UG passed walkthrough [UGTRICORERMK2R1]

Table 1.1. Document history

2. Begin here

The purpose of this document is to provide the reader with the information necessary to configure and use the microkernel, which is part of EB tresos Safety OS.

It is assumed that the reader is familiar with the AUTOSAR-OS module and its interfaces. If you are unfamiliar with AUTOSAR-OS, refer to the EB tresos AutoCore OS documentation [ASCOS_USERGUIDE] for a better description of the services provided by a standard AUTOSAR-OS.

Chapter 2, “Begin here“ (this chapter) describes the layout of the document and provides a glossary and a list of referenced documents.

Chapter 3, “About this documentation“ describes the style and typography of this document.

Chapter 4, “Safe and correct use of EB tresos Safety OS“ describes the features and limitations of the microkernel.

Chapter 5, “Conventions for UML diagrams“ contains information about conventions used in UML-diagrams.

Chapter 6, “Background information“ introduces EB tresos Safety OS.

Chapter 7, “Using the microkernel“ describes how to configure and use the microkernel.

Chapter 8, “Microkernel reference manual“ contains reference material including a detailed API reference and configuration guide.

Chapter 9, “Supplementary information for TriCore processors“ describes CPU family-dependent topics.

3. About this documentation

3.1. Typography and style conventions

Throughout the documentation you see that words and phrases are displayed in bold or italic font, or in Monospace font. To find out what these conventions mean, consult the following table. All default text is written in Arial Regular font without any markup.

Convention	Item is used	Example
Arial italics	to define new terms	The *basic building blocks* of a configuration are module configurations.
Arial italics	to emphasize	If your project's release version is mixed, all content types are available. It is thus called *mixed version*.
Arial italics	to indicate that a term is explained in the glossary	...exchanges *protocol data unit*s (*PDU*s) with its peer instance of other ECUs.
Arial boldface	for menus and submenus	Choose the **Options** menu.
Arial boldface	for buttons	Select **OK**.
Arial boldface	for keyboard keys	Press the **Enter** key
Arial boldface	for keyboard combination of keys	Press **Ctrl**+**Alt**+**Delete**
Arial boldface	for commands	Convert the XDM file to the newer version by using the **legacy convert** command.
Monospace font (Courier)	for file and folder names, also for chapter names	Put your script in the `function_name/abc-folder`
Monospace font (Courier)	for code	`for (i=0; i<5; i++) { /* now use i */ }`
Monospace font (Courier)	for function names, methods, or routines	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Monospace font (Courier)	for user input/indicates variable text	Enter a `three-digit prefix` in the menu line.
Square brackets [ ]	for optional parameters; for command syntax with optional parameters	`insertBefore [<opt>]`

Convention	Item is used	Example
Curly brackets {}	for mandatory parameters; for command syntax with mandatory parameters (in curly brackets)	`insertBefore {<file>}`
Three dots …	for further parameters	`insertBefore [<opt>…]`
A vertical bar |	to separate parameters in a list from which one parameters must be chosen or used; for command syntax, indicates a choice of parameters	`allowinvalidmarkup {on|off}`
Warning	to show information vital for the success of your configuration	**WARNING** ⚠ **This is a warning** This is what a warning looks like.
Notice	to give additional important information on the subject	**NOTE** ⓘ **This is a notice** This is what a notice looks like.
Tip	to provide helpful hints and tips	**TIP** 💡 **This is a tip** This is what a tip looks like.

4. Safe and correct use of EB tresos Safety OS

4.1. Intended usage of EB tresos Safety OS

EB tresos Safety OS is intended to be used in automotive projects based on AUTOSAR. For more information about the AUTOSAR consortium, see www.autosar.org.

4.2. Possible misuse of EB tresos Safety OS

- If you use the product in mass production projects without having undergone the mass production review process, it is likely that the product functions differently than defined. Elektrobit Automotive GmbH is not liable for such misuse.

  To use the product for manufacturing, you must undergo the review process as described in the [EBMASANNEX] document. You have received this document together with the product quote supplied by EB.
- If you use the product in applications that are not defined by the AUTOSAR consortium, the product and its technology may not conform to the requirements of your application. Elektrobit Automotive GmbH is not liable for such misuse.

4.3. Target group and required knowledge

- Basic software engineers
- Application developers
- Programming skills and experience in programming AUTOSAR-compliant ECUs

4.4. Quality standards compliance

EB tresos Safety OS has been developed following processes that have been assessed and awarded Automotive SPICE Level 2.

4.5. Suitability of EB tresos Safety OS for mass production

The suitability of EB tresos Safety OS for mass production is defined by the mass production review process. Prerequisite for this process is the *quality statement*. This quality statement is available for each release, platform, and derivative of EB tresos Safety OS.

The quality statement for your specific delivery, which is determined by release, platform, and derivative you ordered, is available on the *EB Command* server. The link to EB Command as well as your user login and password were sent to you via email.

The naming scheme of the quality statement is as follows:
`EB_tresos_AutoCore-quality_statement-SafetyOS_<Version>-<Derivative>.pdf`
For example, `EB_tresos_AutoCore-quality_statement-SafetyOS_1.0.0-XPC56XXL.pdf`

4.6. Releases with or without safety approval

Depending on your order, EB tresos Safety OS is delivered with or without safety approval.

4.6.1. Release with safety approval

EB tresos Safety OS is developed according to the procedures required for ASIL-D as laid out in [ISO26262_1ST]. You can use EB tresos Safety OS in safety-related systems up to that integrity level, provided that the conditions in the EB tresos Safety OS safety manual for TriCore family [SAFETYMANTRICORE] are observed.

4.6.2. Release without safety approval

You can use EB tresos Safety OS without safety approval in systems that are not safety related.

If you are using a release without safety approval, be aware of the following warning:

WARNING ⚠ **Memory protection should be enabled**

It is strongly recommended that memory protection is `enabled` at all times. EB tresos Safety OS relies on functional memory protection to detect stack corruption and stack overflow and underflow. Disabling the memory protection can lead to unpredictable behavior. See Section 6.8, “Memory protection” for details about memory protection.

4.6.3. Changing from EB tresos Safety OS without safety approval to EB tresos Safety OS with safety approval

To change from EB tresos Safety OS without safety approval, you need to purchase EB tresos Safety OS with safety approval.

In addition to the activities performed for products without safety approval, the following activities are required for products with safety approval:

- verification of the implementation according to the procedures required for ASIL-D
- analysis of the hardware vendor's safety manual
- implementation of safety measures derived from the hardware safety manual
- safety analysis of the OS implementation and documentation in the safety analysis report
- preparation of the EB tresos Safety OS safety manual

These activities will result in changes to the software implementation.

WARNING ⚠ **Software exchange required**

If you change EB tresos Safety OS without safety approval to EB tresos Safety OS with safety approval, you shall replace the existing software with the new software. Additionally, you shall follow the instructions given in the safety manual.

5. Conventions for UML diagrams

The conventions for UML diagrams are depicted in Figure 5.1, “Conventions used in UML message sequence charts”.

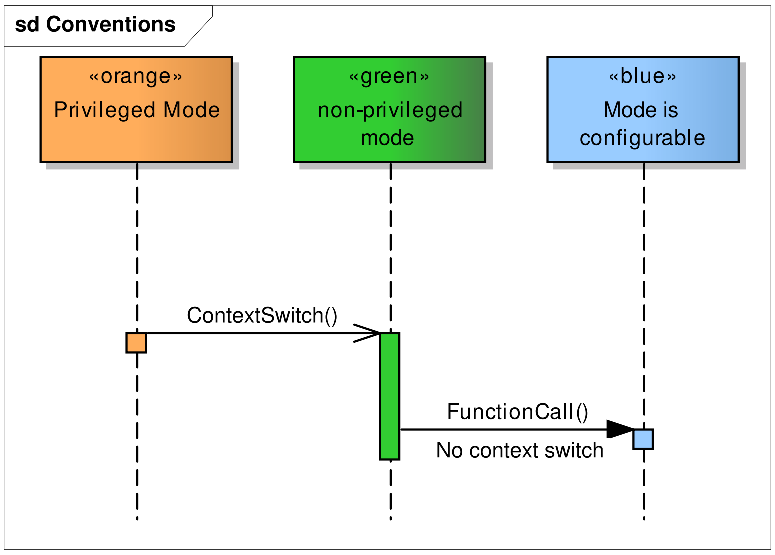


Figure 5.1. Conventions used in UML message sequence charts

In a message sequence chart, each lifeline represents one operating system context. A context switch is denoted by an asynchronous message pass. In contrast, a function call that does not involve a context switch is depicted using a synchronous message pass.

The following colors are used:

orange
: The context is executed in *privileged mode*.

green
: The context is executed in *non-privileged mode*.

blue
: The CPU mode depends on the configuration.

6. Background information

EB tresos Safety OS is an implementation of a subset of the `Os` module from the AUTOSAR standard that you can use in projects where a high safety integrity level is demanded.

The microkernel is the part of EB tresos Safety OS that manages the executable objects of an AUTOSAR system. The executable objects thus managed are the tasks, ISRs, and hook functions defined by the AUTOSAR standard. All of these objects are managed as executing instances of subprograms referred to in the microkernel as *threads*. When the microkernel is correctly configured, the threads are able to modify only those regions of memory that they are explicitly permitted to modify. By this means, the microkernel provides *spatial freedom from interference* between threads.

In addition, the microkernel provides an interface to the counter, alarm, and schedule table functionality of a standard AUTOSAR OS. The microkernel uses threads to manage this interface too, and so prevents spatial interference between the standard OS and your own executable objects.

The microkernel is designed to be as lightweight as possible for reasons of performance and simplicity. In several cases it deviates from the AUTOSAR specification. The deviations are listed in Section 8.6, “Deviations from the AUTOSAR standard”.

This chapter gives a brief introduction into the behavior and services of AUTOSAR-based operating systems. You can find further information in [ASCOS_USERGUIDE], [AUTOSAROS42SPEC] or [OSEKOS223SPEC].

6.1. Tasks

A *task* is an executable subprogram. EB tresos Safety OS manages tasks using *threads*. A **priority** is assigned to a task. If several tasks are ready to be executed in the system at the same point of time, the task with the highest priority is selected. This behavior is shown in Figure 6.1, “Scheduling tasks based on the priority”. Tasks with equal priority are executed in order of activation.

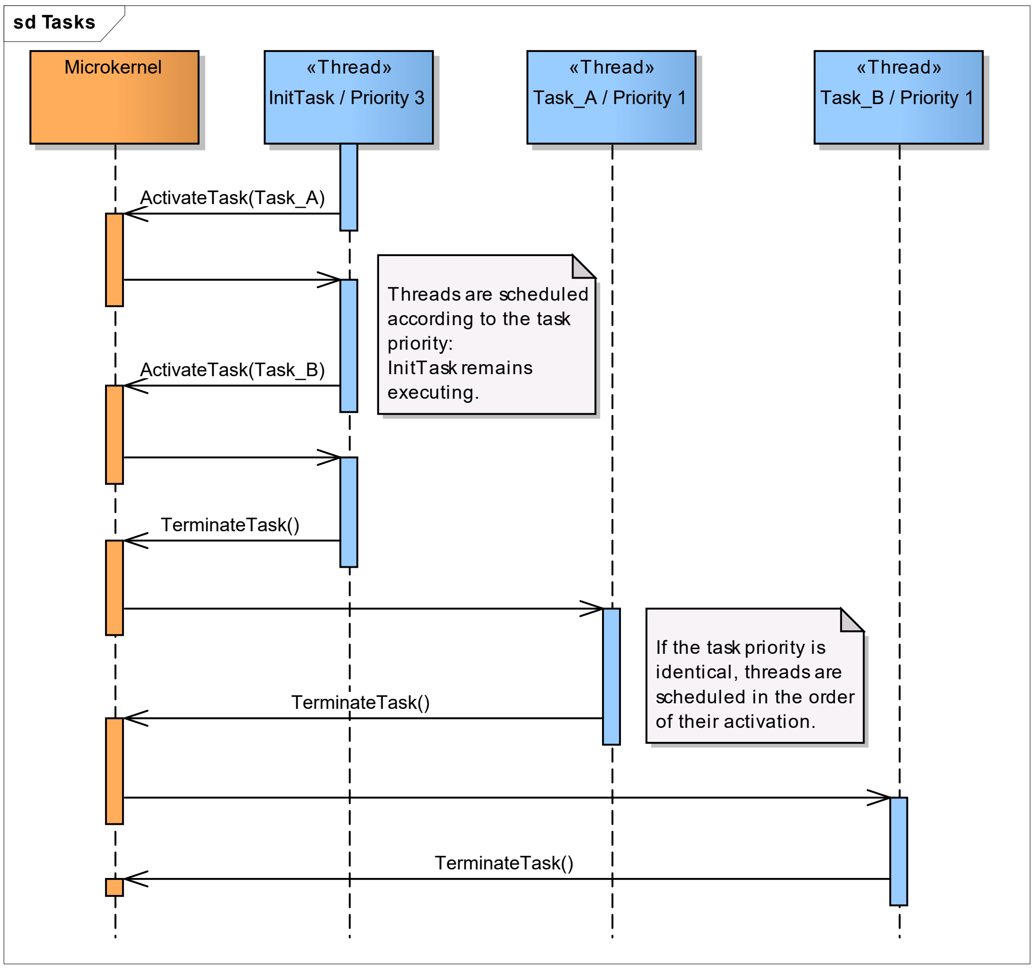

sd Tasks
Microkernel
«Thread» InitTask / Priority 3
«Thread» Task_A / Priority 1
«Thread» Task_B / Priority 1
ActivateTask(Task_A)
Threads are scheduled according to the task priority: InitTask remains executing.
ActivateTask(Task_B)
TerminateTask()
If the task priority is identical, threads are scheduled in the order of their activation.
TerminateTask()
TerminateTask()

Figure 6.1. Scheduling tasks based on the priority

The **scheduling policy** controls, *when* EB tresos Safety OS decides which task should get executed next. Two different scheduling policies exist:

full-preemptive
: If a task with a higher priority becomes ready, it is executed *immediately*. The currently executing task is preempted and resumed later. This behavior is depicted in Figure 6.2, “Scheduling of fully preemptable tasks”.

non-preemptive
: A non-preemptive task continues its execution until it terminates or it calls one of the following services:

- `Schedule()`
- `WaitEvent()` or `MK_WaitGetClearEvent()` if the event is not present.

For information on how EB tresos Safety OS schedules a non-preemptive task, see Figure 6.3, “Scheduling of non-preemptable tasks”.

You can set the scheduling policy for each task. Figure 6.4, “Non-preemptive scheduling policy for a task in EB tresos Studio” depicts how to configure the scheduling policy using EB tresos Studio.

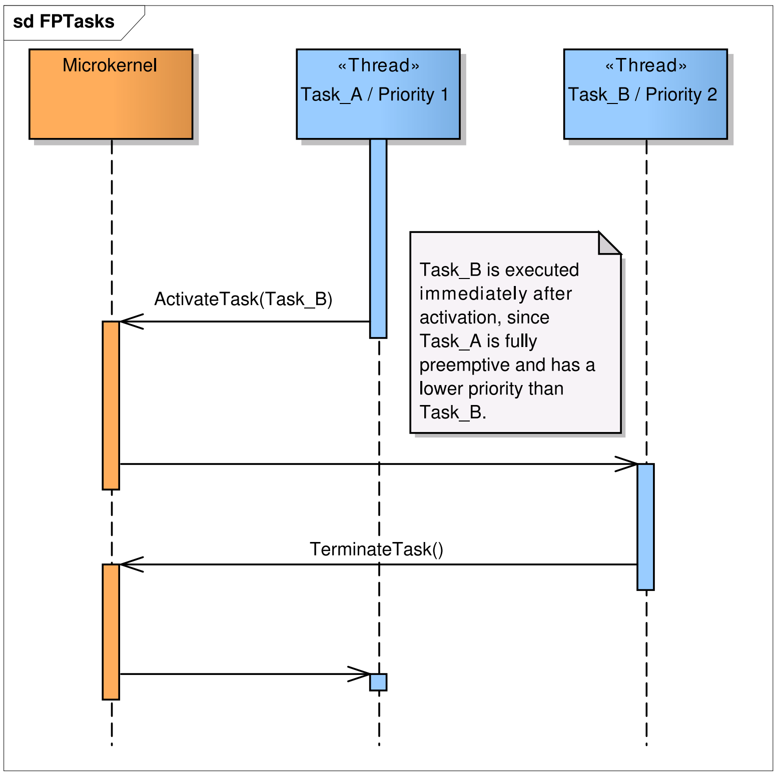

sd FPTasks
Microkernel
«Thread» Task_A / Priority 1
«Thread» Task_B / Priority 2
ActivateTask(Task_B)
Task_B is executed immediately after activation, since Task_A is fully preemptive and has a lower priority than Task_B.
TerminateTask()

Figure 6.2. Scheduling of fully preemptable tasks

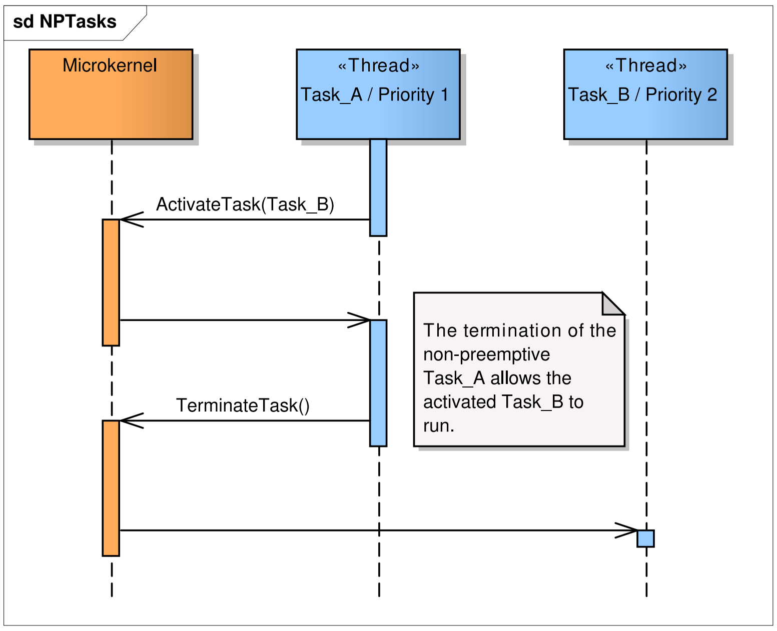
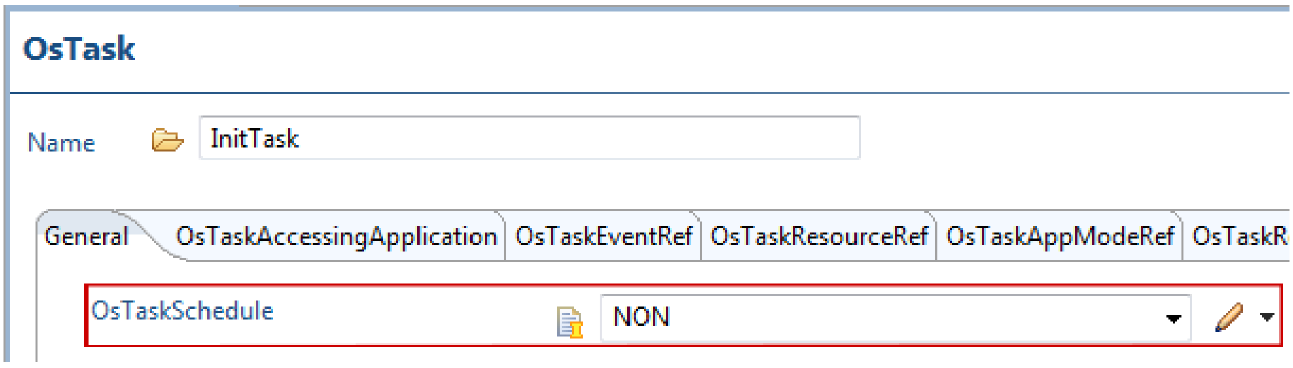

sd NPTasks
Microkernel
«Thread» Task_A / Priority 1
«Thread» Task_B / Priority 2
ActivateTask(Task_B)
The termination of the non-preemptive Task_A allows the activated Task_B to run.
TerminateTask()

Figure 6.3. Scheduling of non-preemptable tasks


OsTask

Name: InitTask

OsTaskSchedule: NON

Figure 6.4. Non-preemptive scheduling policy for a task in EB tresos Studio

AUTOSAR defines the following *states* for a task:

RUNNING
: A task in the state RUNNING is currently executed by the processor. On each core of the processor, only one task can be in the state RUNNING at a time.

SUSPENDED
: A task in the state SUSPENDED is passive and can get activated.

READY
: A task in the state READY is ready to be executed by the processor, but another task currently occupies the processor. Therefore a task in the state READY needs to wait until the scheduler assigns the processor to it.

WAITING
: A task in the state WAITING waits until an event is signaled.

In EB tresos Safety OS, these states are not maintained explicitly but are derived on demand from the underlying state of the threads and task objects.

6.2. Interrupts and ISRs

Hardware components can signal *interrupts* to denote that they need attention. To act on such a request, you can use an *interrupt service routine (ISR)*. EB tresos Safety OS activates the interrupt service routine when the interrupt request is accepted by the processor.

In EB tresos Safety OS, each ISR has two essential properties:

interrupt source
: The interrupt source is a hardware mechanism that connects a peripheral to the interrupt mechanism of the processor. It allows the peripheral to request service from the processor and the processor to exercise control over the requesting peripheral's requests. The process by which a running processor gets interrupted to service the interrupt is called *vectoring*. On typical hardware, the interrupt source is a register

and an entry in the *vector table*. The manual of the microcontroller contains a list of interrupt sources with their meaning. For the TriCore the interrupt source is configured with the parameter `OsTricoreVector`. For an example for the Power Architecture, see Figure 6.6, “ISR configuration in EB tresos Studio”. Other CPU families are similar.

interrupt level

The interrupt level is a number whose purpose is similar to the task priority: If an interrupt service routine is being executed, and an interrupt source with a higher level is triggered, then the processor accepts the interrupt request and preempts the currently executing interrupt service routine. The interrupt level is a hardware-dependent value, and the exact meaning depends on the hardware. Some hardware defines a higher numerical value to mean a higher level, while other hardware defines a lower numerical value to mean a higher level. In the case of the TriCore you can select the interrupt level of an ISR with the option `OsTricoreIrqLevel`. For an example for the Power Architecture, see Figure 6.6, “ISR configuration in EB tresos Studio”. Other CPU families are similar.

Note that the number programmed into the hardware might differ from the value you selected. This is because EB tresos Studio optimizes the range of levels used. However, the ordering is preserved.

Figure 6.5, “Execution of ISRs” depicts the execution of an ISR.

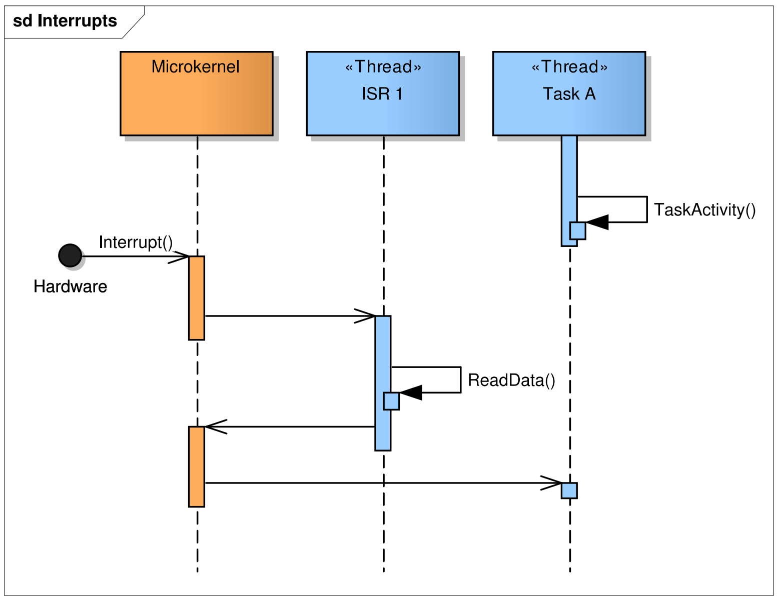
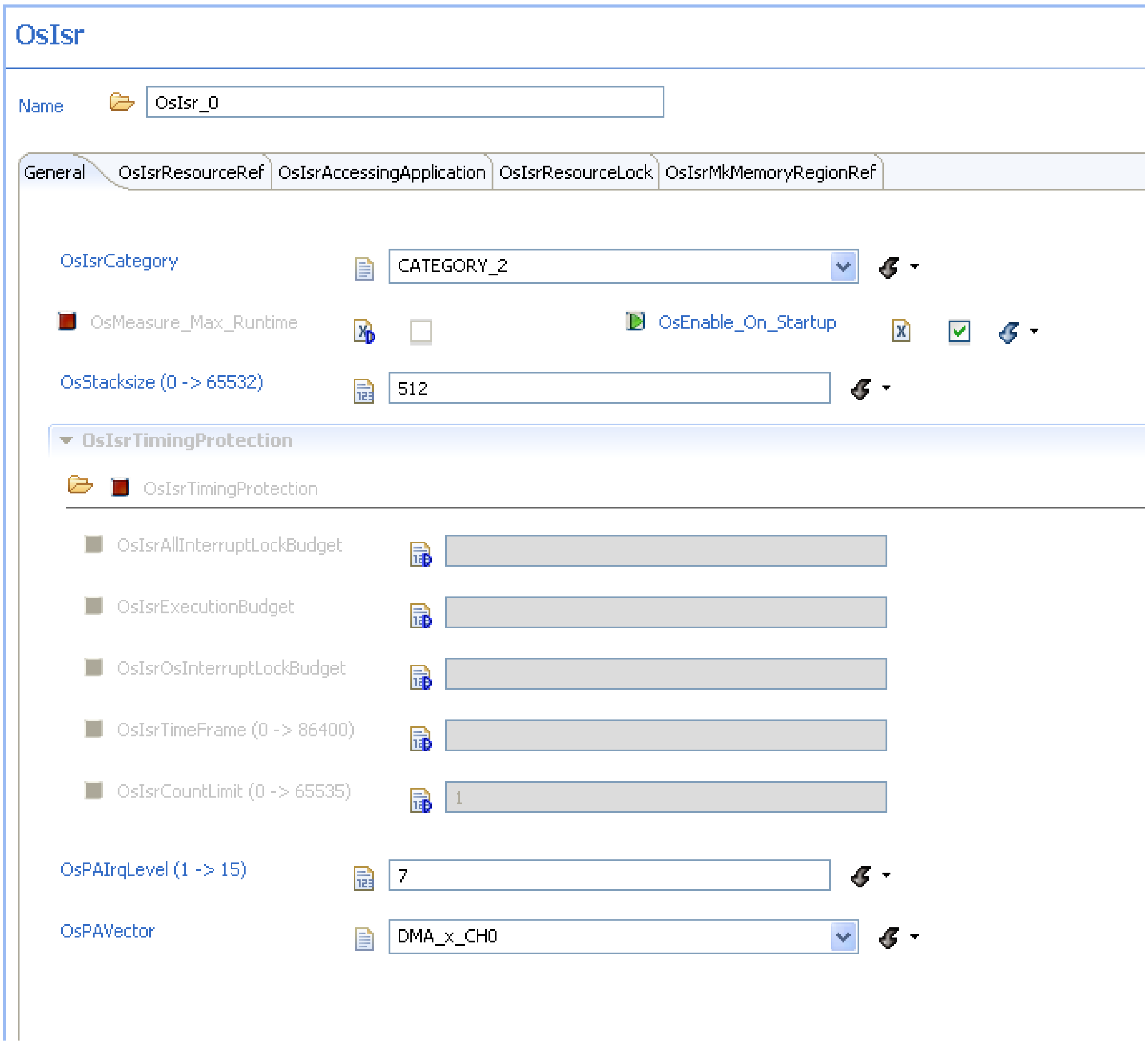


Figure 6.5. Execution of ISRs


OsIsr

Name: OsIsr_0

General | OsIsrResourceRef | OsIsrAccessingApplication | OsIsrResourceLock | OsIsrMkMemoryRegionRef

OsIsrCategory	CATEGORY_2
OsMeasure_Max_Runtime	☐
OsEnable_On_Startup	✓
OsStacksize (0 -> 65532)	512

OsIsrTimingProtection

OsIsrTimingProtection

OsIsrAllInterruptLockBudget	
OsIsrExecutionBudget	
OsIsrOsInterruptLockBudget	
OsIsrTimeFrame (0 -> 86400)	
OsIsrCountLimit (0 -> 65535)	1

OsPAIrqLevel (1 -> 15)	7
OsPAVector	DMA_x_CH0

Figure 6.6. ISR configuration in EB tresos Studio

The microkernel schedules ISRs according to their configured *priority* but programs their interrupt levels into the hardware while they are executing.

Interrupts occur asynchronously to the execution of tasks. This means that an ISR can preempt a task at any time. If you do not desire this behavior, you can disable interrupts temporarily.

You can use the service `DisableAllInterrupts()` to disable all interrupts. You can use `EnableAllInterrupts()` to re-enable interrupt processing. An example is shown in Figure 6.7, “Interrupt locks”.

NOTE

Critical section protection

Disabling the interrupts does not suffice to protect critical sections on multi-core processors if the critical section must be protected from accesses from another processor core.

For more information on how to implement critical sections, see Section 6.6, “Synchronizing access to common data”.

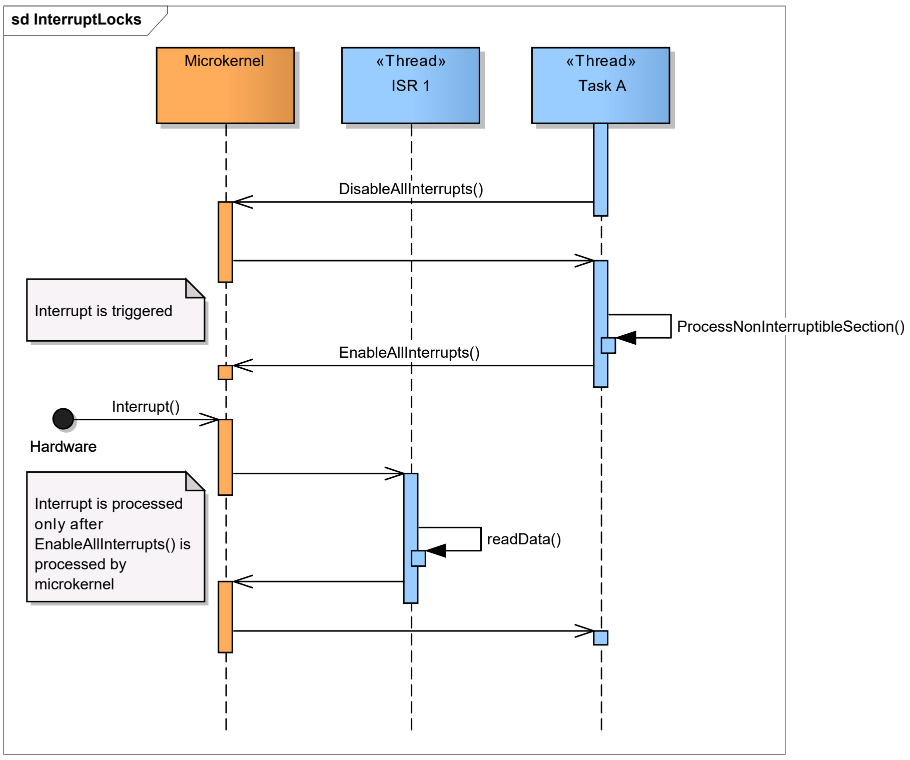


Figure 6.7. Interrupt locks

6.3. Trusted functions

Trusted functions are services which are typically provided by the application and which need to run with their own processor mode and memory protection settings in contrast to those of the calling task or ISR.

6.3.1. Introduction to trusted functions

Trusted functions are executable subprograms which cannot be executed in the environment of the caller as they need additional rights or in general a special environment. Examples are functions that need to run in supervisor mode e.g. to perform hardware accesses or that need write access to special memory regions, that the caller does not have.

Each trusted function has a configured processor mode in which it is started and can have their own memory protection configuration. It is furthermore assigned to a certain core and may only be called on that core in multi-core systems.

When a trusted function is called by an application via `CallTrustedFunction()`, the following behavior happens in the microkernel if no other trusted function already runs on the calling core:

1. The microkernel activates the trusted function thread, which is a special thread to execute trusted functions. The priority is set so that the thread precedes the thread of the caller. The processor mode and the memory protection settings are set to those configured for the trusted function.
2. The microkernel preempts the execution of the caller and executes the trusted function thread. There the requested trusted function is executed with the parameters which were provided to the `CallTrusted-Function()` call.
3. When the trusted function ends, the trusted function thread also ends and the microkernel transfers control back to the caller.

This behavior is shown in Figure 6.8, “Execution of a trusted function”.

If a trusted function is called and the trusted function thread is already occupied by another trusted function, the behavior changes as follows:

1. The microkernel enqueues the trusted function into the trusted function thread queue. It then raises the trusted function threads priority so that it precedes the thread of the caller.
2. The microkernel preempts the execution of the caller and continues with the trusted function thread. There, the current trusted function as well as all the queued ones are executed with the correct settings and parameters. That means e.g. that processor mode and memory settings are adapted for each trusted function executed in the thread.
3. The trusted function thread ends when the queue is empty and the microkernel transfers control back to the trusted function caller with the highest priority. There are several callers as the trusted function thread was already occupied.

In each case, when control is transferred back to the caller of a trusted function, that trusted function was executed and finished at that point.

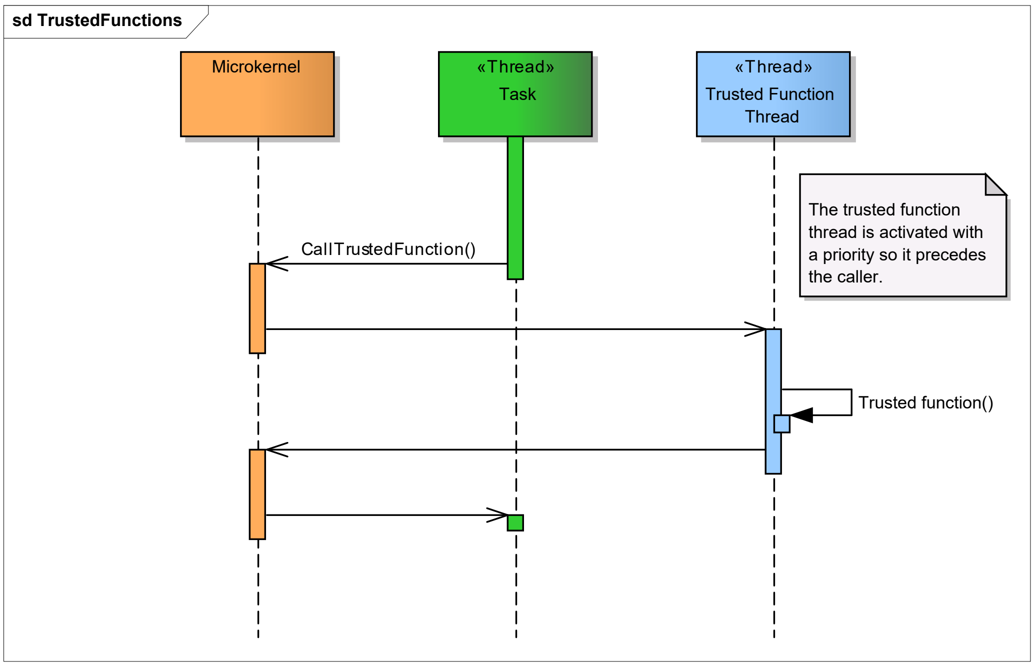

sd TrustedFunctions
Microkernel
«Thread» Task
«Thread» Trusted Function Thread
The trusted function thread is activated with a priority so it precedes the caller.
CallTrustedFunction()
Trusted function()

Figure 6.8. Execution of a trusted function

6.3.2. Configuration and use of trusted functions

If you want to use a trusted function within EB tresos Safety OS, you can use EB tresos Studio to configure the trusted function in the configuration containers `/Os/OsApplication/OsApplicationTrustedFunction`. There you can configure the trusted function itself including trusted function specific memory regions, the processor mode, the necessary stack size, and other properties.

Within your application code, call the service `CallTrustedFunction()` to call a trusted function. The first parameter is the function index. This is the index of the corresponding function in the trusted function configuration list `MK_CFG_TRUSTEDFUNCTIONLIST`. If you use EB tresos Studio to generate the basic configuration, you can find constants with the name of the trusted function in file `Mk_gen_user.h` that you can use as function index.

6.4. Interaction with the QM-OS

The microkernel does not implement all API services which are defined by AUTOSAR. API services with integrity category 3 are implemented by the QM-OS. API services with integrity category 2 have a wrapper implemented in the microkernel which executes the corresponding QM-OS service within a special thread, called the QM-OS thread. For more information on the integrity categories, see [SAFETYMANTRICORE].

6.4.1. Services with integrity category 2

Services with integrity category 2 are services which are implemented by the QM-OS but which have a wrapper in the microkernel to ensure freedom from interference in the data domain. In order to do this, the microkernel wrapper starts a special thread, the QM-OS thread and executes the QM-OS service in this thread. The handling of integrity category 2 services is shown in Figure 6.9, “Execution of a QM-OS service”.

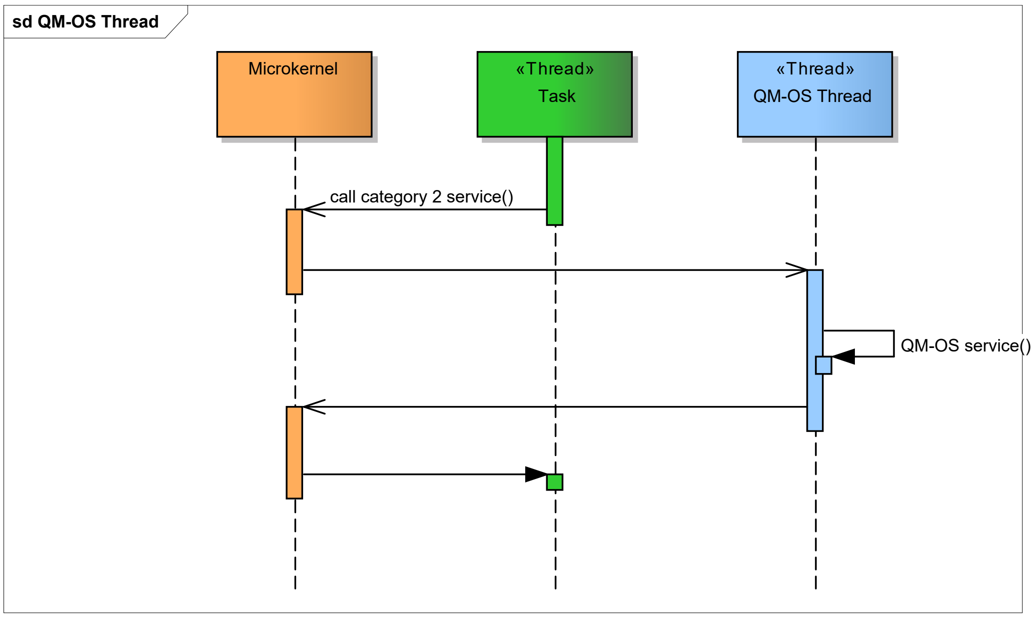


Figure 6.9. Execution of a QM-OS service

The handling of integrity category 2 services is similar to the handling of trusted functions, which is illustrated in Figure 6.8, “Execution of a trusted function”.

If you use EB tresos Studio to configure the microkernel, the QM-OS thread automatically gets access to the necessary memory regions and has an appropriate processor mode. In addition to that, you can add access to further memory regions to the thread when you set the parameter `/Os/OsMicrokernel/MkMemoryProtection/MkMemoryRegion/MkMemoryRegionOsThreadAccess` of the memory region you want to add. You could also change the threads processor mode with the parameter `/Os/OsMicrokernel/MkThreadCustomization/MkOsThreadMode` but you should not do this.

Services with integrity category 2 are marked as MK/OS in Section 8.2, “Microkernel API reference”.

6.4.2. Services with integrity category 3

Services with integrity category 3 are services which are implemented by the QM-OS without any additions from the microkernel. They are executed directly in the context of the caller and do not provide freedom from interference like integrity category 1 and 2 services. Therefore the caller must take appropriate measures if such a service is used in an ASIL partition.

If you use EB tresos Studio to configure the microkernel, the tasks and ISRs automatically get access to the necessary memory region. Therefore you do not need to configure anything explicitly in EB tresos Studio.

Services with integrity category 3 are marked as OS in Section 8.2, “Microkernel API reference”.

6.5. Multi-core operation

The microkernel is capable of executing concurrently on two or more cores of a multi-core processor. When the microkernel is configured for multi-core operation, the AUTOSAR concept of static core allocation for OS objects permits the microkernel to operate each core independently of the others. The scheduling mechanisms described for tasks and ISRs are applied independently on each core.

Inter-core requests are performed by message passing and are thus handled on the core on which the targeted object is configured to run. This mechanism avoids the need for inter-core synchronization locks inside the microkernel.

6.5.1. Core index and physical core mapping

AUTOSAR distinguishes between physical core IDs and the logical CoreID. The microkernel uses logical core indices which are similar but not equal to the AUTOSAR CoreID. These core indices start with the value `0` and are consecutive up to the value of `MK_MAXCORES - 1` which is hardware-dependent. For more information on the logical core indices and their mapping to the physical cores, see Section 9.10, “Multi-core support on TriCore processors”.

Logical core indices are used both in the software as parameters as well as in the configuration to identify the different cores.

If Advanced Logical Core Identifiers are enabled, different core mappings than the default mapping described above can be configured. You can find further information on this topic in [ASCOS_USERGUIDE].

Negative core indices

Several microkernel services require the core index as parameters. In addition to the logical core indices as supported by the hardware, these services allow negative values as core index as well. A negative core index is interpreted as *the current core* by these services.

This handling is used by several microkernel services to provide a simple version of the service for the current core. A call to `MK_GetPanicReasonForCore(-1)` for example behaves in the same way as `MK_GetPanicReasonForCore(i)` where `i` is the core index of the current core and also in the same way as `MK_GetPanicReason()`.

6.5.2. Inter-core communication

You can request task activations, event settings, and QM-OS services from any core, regardless of which core the targeted object is configured. The microkernel provides two mechanisms to handle inter-core requests:

- Synchronous inter-core requests
- Asynchronous inter-core requests

Synchronous inter-core requests

Synchronous inter-core requests are a mechanism used for services that trigger work on another core and need to return the result of that work. An example is the API `ActivateTask()` if it is called for a task that is assigned to a different core than the one, on which it is called.

In such a case, the microkernel performs the following actions:

Sequence	Source core	Target core
1	Adds a message for the requested action to the message queue of the target core. If the message queue is already full, the core waits until there is room and adds the message then.	Executes the target core application
2	Signals the new message to the target core.	
3	Waits for the reply.	Interrupts normal application execution to read the message and perform the requested action.
4		Sends a reply back to the source core which indicates the result of the requested action.
5	Reads the reply and return the result to the caller.	Returns to the target core application

Source core identifies the core on which the API service was called. *Target core* identifies the core on which the API functionality must be performed, e.g. the core on which the task runs which shall be activated by a `ActivateTask()` call.

The signaling is done via inter-core interrupts which have the highest priority of all used interrupts assigned to them. The priority is set in this way so that these interrupts are not locked out by any interrupt locking API, especially including `SuspendAllInterrupts()` and `DisableAllInterrupts()`. You do not need to configure these interrupts if you use EB tresos Studio for the configuration. If you manually configure the microkernel, you find more information on the inter-core interrupts in Section 9.10, “Multi-core support on TriCore processors”.

NOTE

Error reporting to the error hook

If an error is identified during the execution of an API that includes a request to another core, the core which identifies the error is the one which reports it to the error hook. The other core does not report the error to the error hook. Independent of where the error is identified, if there is a return value corresponding to the error, that return value is returned to the caller.

Example: If `ActivateTask()` is called with an invalid task ID, that error is already identified on the *source core* and therefore also reported there to the error hook. If on the other side the task ID is valid but the activation counter gets exceeded by the new task activation this is identified on the target core and reported to the error hook there.

Asynchronous inter-core requests

Synchronous inter-core requests are costly in CPU time. Therefore, the microkernel extends the AUTOSAR API for inter-core requests by adding a set of asynchronous *fire and forget* APIs. If the statically-verifiable conditions for the parameters are satisfied, e.g. with range checks, a result code of `E_OK` is returned to the caller immediately after the message is enqueued for the target core. If a dynamic error such as `E_OS_LIMIT` is detected, it is reported by means of the `ErrorHook()` function. The caller however is not notified.

The asynchronous API is available for all objects, even those on the same core as the caller. For microkernel objects on the same core asynchronous calls are converted to synchronous calls. However, asynchronous calls to the QM-OS services can be enqueued and executed later if the caller runs at a high priority.

You can identify asynchronous APIs by their name. They start with the prefix `MK_` as they are microkernel-specific functions, followed by `Async` to indicate *asynchronous* and end with the AUTOSAR API name. `MK_AsyncActivateTask()` is for example the asynchronous version of `ActivateTask()`.

6.6. Synchronizing access to common data

In a multitasking environment, tasks and ISRs typically share access to a number of physical as well as system resources. In EB tresos Safety OS, *locks* provide a means to coordinate concurrent access to shared data. The microkernel's lock mechanism implements all the coordination mechanisms specified by AUTOSAR resources, interrupt locks, and spinlocks, plus an EB-vendor-specific lock that is a combination of a resource and a spinlock.

6.6.1. Resources

AUTOSAR specifies the *Resource* as the mechanism for synchronizing tasks and ISRs that execute on the same core. Resources are free from deadlock and priority inversion.

6.6.1.1. Deadlock prevention

A deadlock occurs when a task or ISR attempts to acquire a lock that is occupied by a second task or ISR, while the second task or ISR waits to acquire a lock that is already occupied by the first task or ISR. OSEK/VDX resource objects prevent deadlock situations by imposing a set of restrictions:

- A task or ISR makes the transition from READY to RUNNING only if all of its resources are available.
- A task or ISR cannot terminate nor can a task enter the WAITING state while it holds resources.
- Multiple resources must be acquired and released in LIFO (last in first out) order. That means each task or ISR logically manages the resources it needs on a stack.
- A task or ISR must not attempt to acquire a resource that it already holds.

6.6.1.2. Priority inversion

Priority inversion arises when a high-priority task (`TH`) is forced to wait an indefinite amount of time for a low-priority task (`TL`) to complete. This situation can arise when `TL` holds a lock that is required by `TH`. `TH` does not enter the RUNNING state until it acquires the lock. The lock does not become available until `TL` releases it. Although the execution time of `TL` while it holds the lock is bounded, any task `TM` with a priority between those of `TL` and `TH` can preempt `TL` and prevent it from releasing the lock. This behavior can happen an indefinite number of times, perhaps by several different `TM` tasks and thus delay the execution of `TH` for an indeterminate time.

To prevent this situation, EB tresos Studio determines the highest priority of all *tasks* or *ISRs* which use a resource. This priority is called the resource's ceiling priority. Whenever a task or ISR acquires a resource, the task's or ISR's priority is set to the calculated ceiling priority. When the resource is released, the old priority is restored again. This approach is known as the OSEK/VDX *priority ceiling protocol* or (in some literature) as the *priority ceiling emulation protocol*.

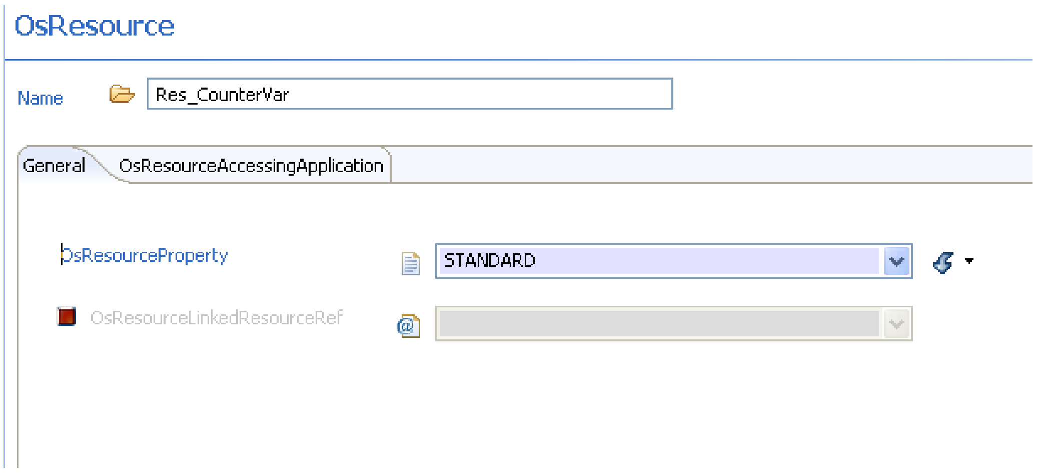
This protocol ensures that no task of lower priority than task `TH` executes while task `TL` holds the resource. Task `TL` is executed preferentially until the moment it releases its resources, thus enabling task `TH` to change into the RUNNING state.

6.6.1.3. Configuration and use of resources

If you want to use a resource within EB tresos Safety OS, EB tresos Studio makes it straightforward to configure it:

1. Create the resource as depicted in Figure 6.10, “Resource configuration in EB tresos Studio”.
2. Configure a reference to the resource for every task and ISR that makes use of the resource, see Figure 6.11, “Referencing a resource in EB tresos Studio”.

Within your application code, call the service `GetResource()` to acquire a resource and `ReleaseResource()` to release it after the critical section. An example of this procedure is depicted in Figure 6.12, “Resources”.

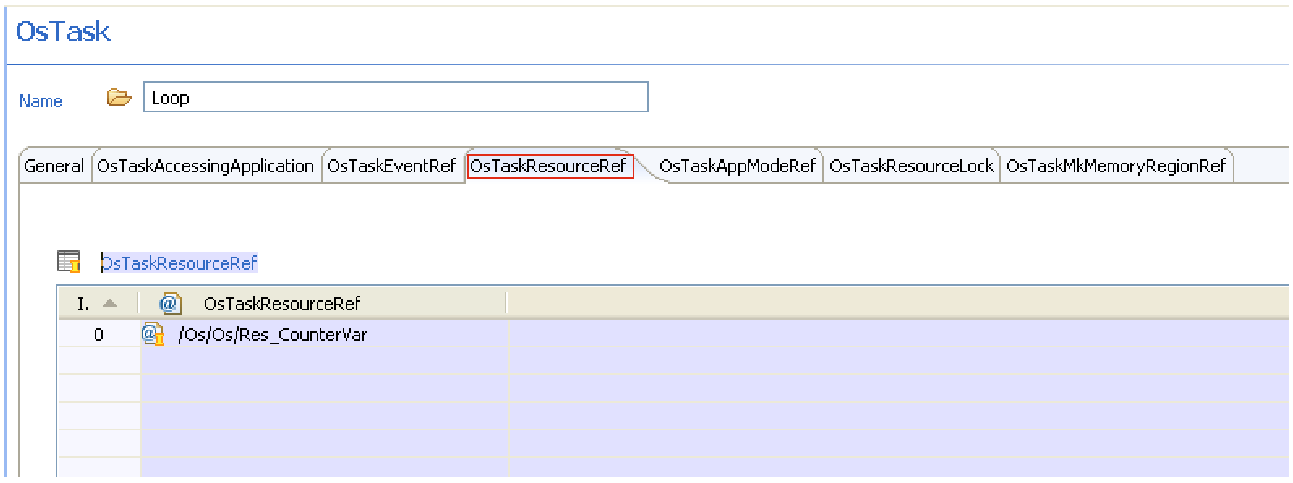

OsResource

Name: Res_CounterVar

OsResourceProperty: STANDARD

OsResourceLinkedResourceRef:

Figure 6.10. Resource configuration in EB tresos Studio


OsTask

Name: Loop

OsTaskResourceRef

I.	OsTaskResourceRef
0	/Os/Os/Res_CounterVar

Figure 6.11. Referencing a resource in EB tresos Studio

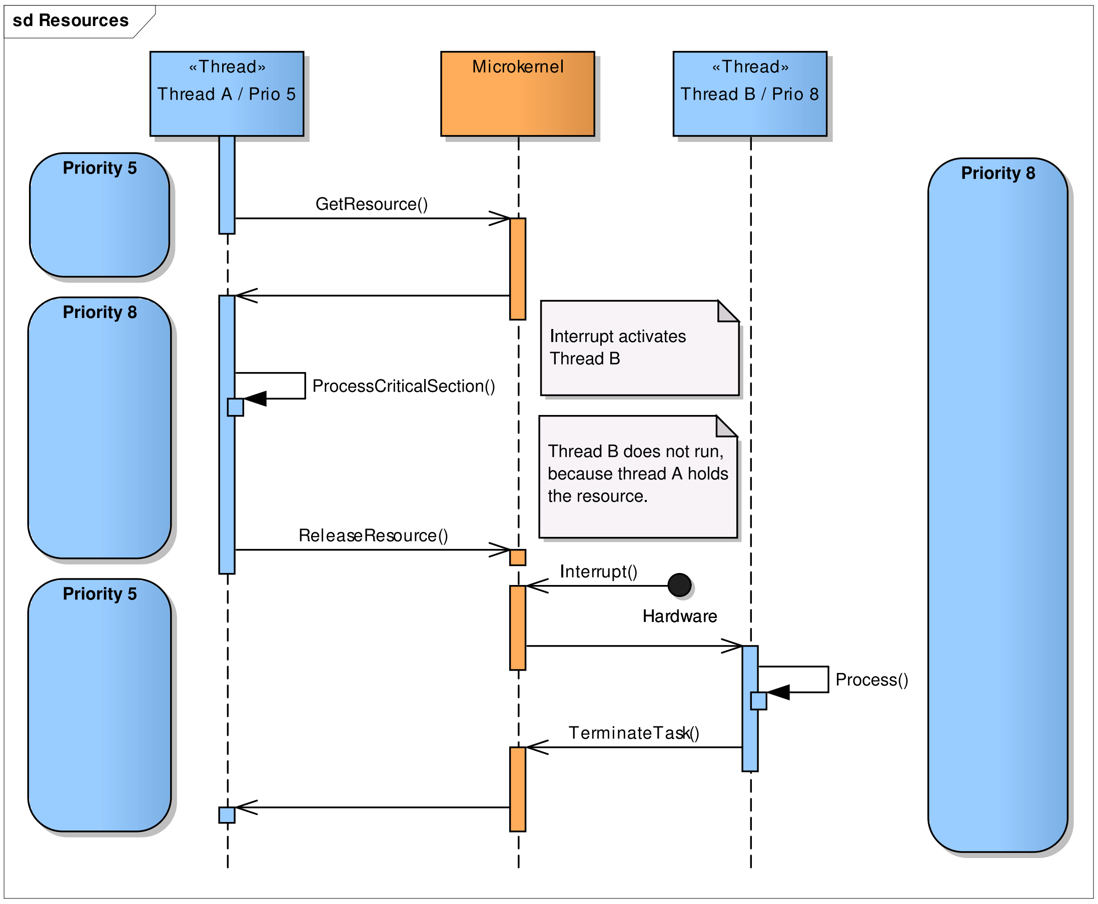


Figure 6.12. Resources

6.6.2. The scheduler resource

OSEK/VDX specifies the *scheduler resource* (`RES_SCHEDULER`) as a resource that is implicitly shared by all tasks, without having to configure it explicitly for each task. Recent versions of the AUTOSAR specification have omitted `RES_SCHEDULER`, but EB tresos Studio and the microkernel retain support for backwards compatibility.

In a multi-core environment, each configured core has its own copy of `RES_SCHEDULER`. Thus, a call to `GetResource(RES_SCHEDULER)` in a task on a specific core blocks all other tasks on that core from running until `RES_SCHEDULER` is released again.

6.6.3. Interrupt locks

Interrupt locking is a convenient way to prevent higher-priority tasks or ISRs from executing on the same core.

6.6.3.1. Introduction

AUTOSAR specifies several different interrupt lock types with the associated APIs, which are supported by the microkernel:

- Lock all category 1 and category 2 interrupts: `SuspendAllInterrupts()` and `ResumeAllInterrupts()` if nesting is needed and `DisableAllInterrupts()` and `EnableAllInterrupts()` if nesting is not necessary.
- Lock category 2 interrupts: `SuspendOsInterrupts()`, `ResumeOsInterrupts()`.

While `DisableAllInterrupts()` and `EnableAllInterrupts()` just enable or disable the interrupts, the other services store the previous lock state and use an acquisition counter. This means that the first `SuspendXxxInterrupts()` call stores the current interrupt lock state, disables the corresponding interrupts and increments the acquisition counter. Further calls to the same service do not change the interrupt lock but instead only increment the acquisition counter. The service `ResumeXxxInterrupts()` decrements the acquisition counter. It furthermore restores the stored interrupt lock state, but only if the acquisition counter reaches `0`. Otherwise it leaves the interrupt lock unchanged.

The two groups `Suspend/ResumeAllInterrupts()` and `Suspend/ResumeOsInterrupts()` have their own, independent acquisition counter.

6.6.3.2. Configuration and use of interrupt locks

If you want to use an interrupt lock, you do not need to configure this in EB tresos Safety OS. Just use the corresponding services within your application.

Do not mix up `Suspend/Disable` and `Resume/Enable` services as well as `Suspend` and `Resume` services for different lock types. If you use e.g. `SuspendAllInterrupts()` to acquire an interrupt lock, use `ResumeAllInterrupts()` to release it again.

Also do not use `DisableAllInterrupts()/EnableAllInterrupts()` nested within another interrupt lock.

NOTE

Use resources instead of interrupt locks
You should not use interrupt locks and instead prefer resources. Interrupt locks do not have a performance advantage over resources in the microkernel and resources can reach a much finer protection granularity than interrupt locks.

6.6.4. Spinlocks

A *spinlock* is a mechanism to synchronize tasks which run on different cores in a multi-core environment. As the name implies, waiting for a spinlock is a busy-waiting activity.

6.6.4.1. Configuration and use of spinlocks

AUTOSAR specifies a spinlock mechanism and the associated API that allows tasks and ISRs to use configured spinlocks. The spinlock implementation in the microkernel is based on the AUTOSAR specification.

If you want to use a spinlock within EB tresos Safety OS, you can do it by creating a spinlock, i.e. a `Os/OsSpinlock` container in EB tresos Studio.

Within your application code, call the services `GetSpinlock()` or `TryToGetSpinlock()` to acquire a spinlock and `ReleaseSpinlock()` to release it.

6.6.4.2. Deadlock prevention

Spinlocks are not deadlock-free. Deadlocks can happen e.g. if two cores want to take the same two spinlocks but use different sequences to get them. While the first core acquired spinlock `sp1` and now tries to get spinlock `sp2`, a second core already could acquire `sp2` and now tries to get `sp1`. Both cores cannot continue until care is taken to prevent such a deadlock.

Spinlocks also do not block tasks on the same core as the caller. If for example a task which acquired a spinlock is preempted by a higher priority task which also needs that spinlock, the higher priority task *spins* forever to acquire the lock and the deadlock is complete.

AUTOSAR defines some mechanisms and API return values to prevent such deadlocks. However, the microkernel does not implement any such mechanism. *The application must ensure that spinlocks are used in a way, that no deadlocks can occur.*

How to prevent deadlocks depends on the application itself. A good starting point are the following hints:

- If several spinlocks are necessary, always acquire the spinlocks in the same sequence.
- If a spinlock is used by several preemptable tasks with different priorities on the same core, encapsulate the section where the spinlock is used into an appropriate critical section, e.g. by using a resource.

6.6.5. Combined locks

The microkernel provides an EB-vendor-specific mechanism named *combined lock* that is essentially a combination of a spinlock with a *core local* lock that behaves like an interrupt lock or a resource.

In addition to its normal spinlock properties, such a combined lock has a ceiling priority and an interrupt lock level. When such a lock is acquired successfully, the caller's priority is raised to the ceiling priority, the interrupt lock level is raised to the combined locks lock level and the spinlock is acquired. This behavior effectively prevents other tasks with a priority lower than the ceiling priority and interrupts with a lower level than the lock level from being executed.

If the lock acquisition fails or the API returns a *try again* status, i.e. return value `MK_E_TRYAGAIN`, the priority and lock level remain unchanged. No part of the lock is taken.

6.6.5.1. Configuration and use of combined locks

If you want to use a combined lock within EB tresos Safety OS, you can do that in EB tresos Studio by creating a spinlock, i.e. a `Os/OsSpinlock` container, and changing the `Os/OsSpinlock/OsSpinlockLockMethod` parameter to one of the predefined combined lock values:

- **LOCK_NOTHING.** *Standard* spinlock
- **LOCK_ALL_INTERRUPTS.** Combined lock: spinlock plus interrupt lock for all interrupts.
- **LOCK_CAT2_INTERRUPTS.** Combined lock: spinlock plus interrupt lock for all category 2 interrupts.
- **LOCK_WITH_RES_SCHEDULER.** Combined lock: spinlock plus a lock similar to acquire `RES_SCHEDULER`. For more information, see also Section 6.6.2, “The scheduler resource”.

You can also manually configure combined locks with individual ceiling priorities and lock levels. For more information, see also Section 8.4.11, “Configuration of locks”.

Within your application code, call the services `GetSpinlock()` or `TryToGetSpinlock()` to acquire a combined lock and `ReleaseSpinlock()` to release it in the same way as you use them for a spinlock.

Some annotations:

- Combined locks can reduce the microkernel overhead if a local lock is needed in addition to a spinlock.
- Combined locks have an acquisition counter similar to Section 6.6.3, “Interrupt locks”. If you acquire a combined lock several times, the lock is acquired with the first call and the acquisition counter is set to `1`. Further acquisitions only increment the acquisition counter. When the combined lock is released, the acquisition counter is decremented for each release. Only when it reaches `0` again, the lock is in fact released.

6.6.5.2. Deadlock prevention

Combined locks are not deadlock-free. The deadlock scenario with the preempting task known from spinlocks above does not apply as long as an appropriate combined lock is chosen, e.g. `LOCK_ALL_INTERRUPTS` effectively prevents the task preemption. However, other scenarios like the one with multiple locks from the spinlock section still can. For more information, see also Section 6.6.4.2, “Deadlock prevention”.

Therefore, like for spinlocks, *the application must ensure, that spinlocks are used in a way, that no deadlocks can occur*.

6.7. Events

In an embedded system it is often necessary to notify a task of a state transition or other special occurrences. *Events* are a mechanism to implement such notifications.

An event is an exclusive signal which is assigned to an arbitrary extended task. You can assign more than one event to the same task. A task may wait for one or more events and thus enters the WAITING state. Any task can set events for arbitrary tasks. Setting an event causes the receiving task to enter the READY state if it waits for at least one of these events.

An exemplary behavior how to utilize events is shown in Figure 6.13, “Events”.

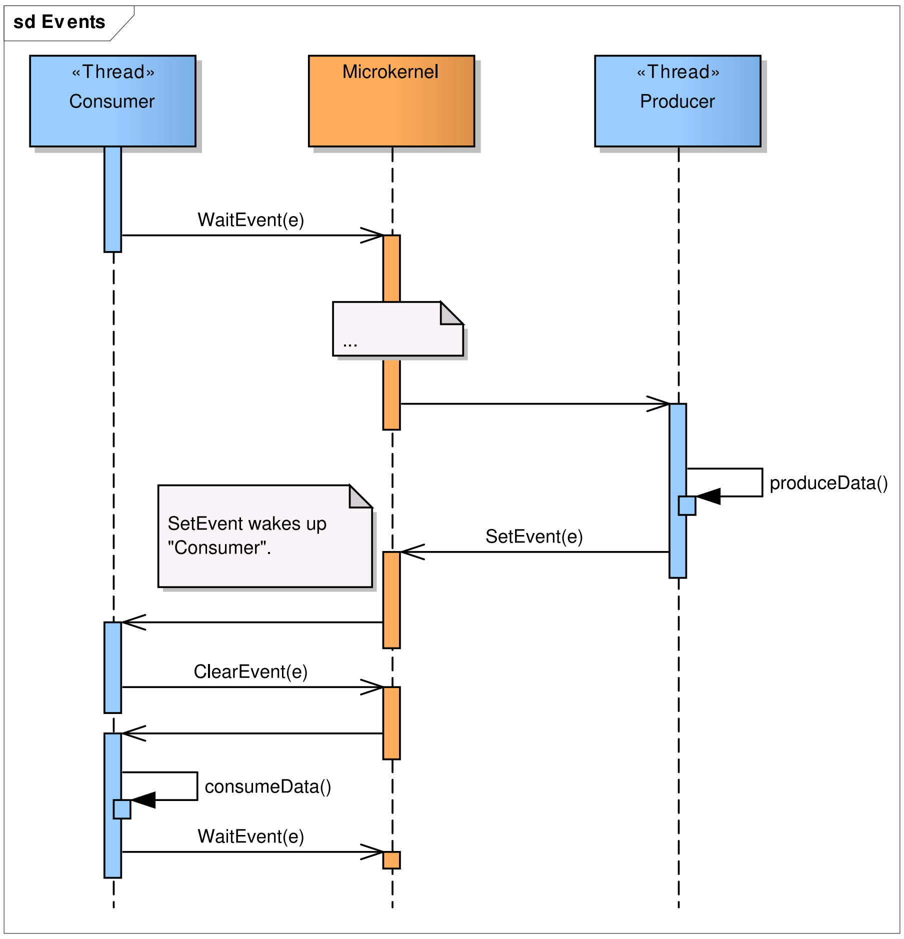


Figure 6.13. Events

6.8. Memory protection

Memory protection boundaries for AUTOSAR-compliant operating systems are based on *OS-objects* and *OS-Applications*: Each OS-object has its own memory region for a stack and its own memory region for private data. Additionally, one shared region for each application exists. You can use variables that reside in this shared region to exchange information between OS-objects of the application. This type of configuration is shown in Figure 6.14, “Memory protection boundaries based on OS-Applications”.

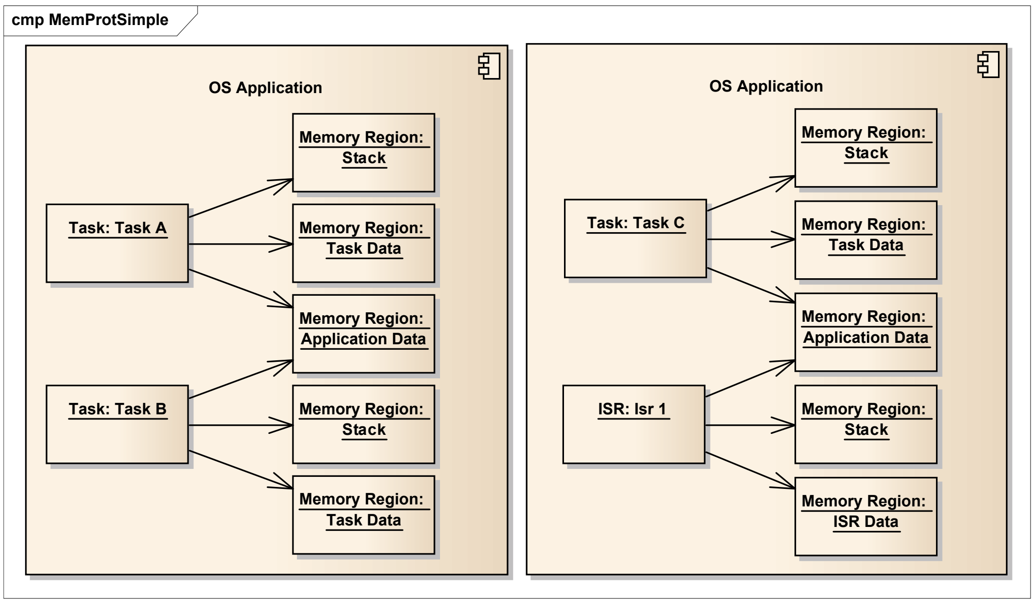


Figure 6.14. Memory protection boundaries based on OS-Applications

When you assign *tasks* and *ISRs* to *OS-Applications* in EB tresos Studio, a configuration for memory protection is created automatically.

WARNING

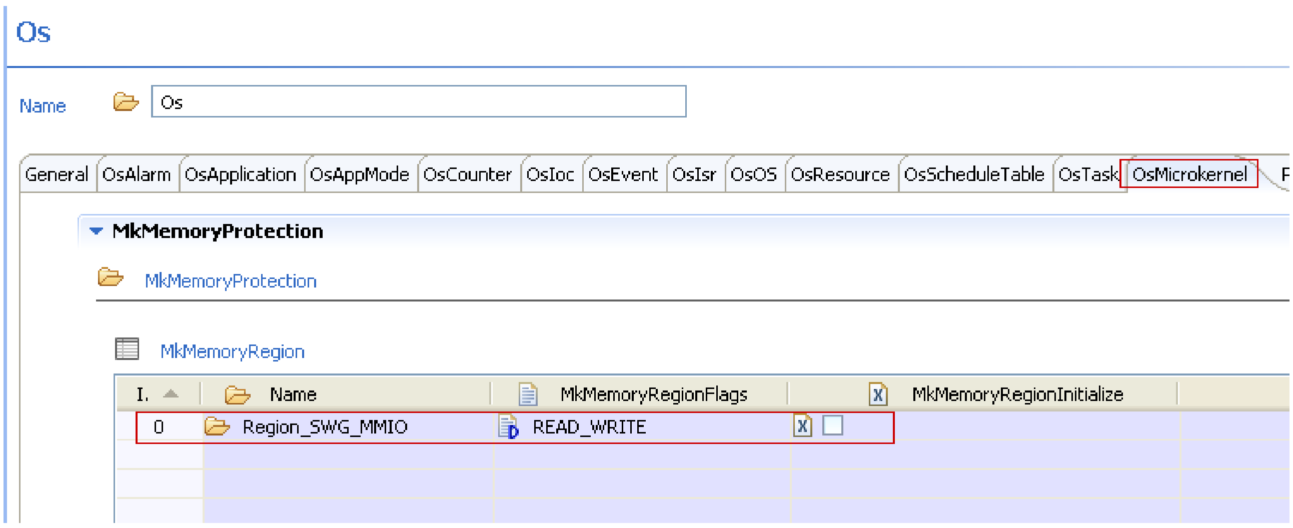
Memory protection configuration for OS-Applications

Configure the `OsTrusted` option for all OS-Applications to `FALSE`. Otherwise, the objects of the OS-Application obtain additional memory access rights.

6.8.1. Fine-granular memory region control

In addition to the aforementioned memory protection configuration, EB tresos Safety OS provides a very fine-grained control of different memory regions and their association with *OS-objects*: You can configure additional memory regions and associate these memory regions with *tasks* and *ISRs*.

Information on how you can configure an additional memory region using EB tresos Studio is shown in Figure 6.15, “Configuration of additional memory regions with EB tresos Studio”. Information on how a memory region can be added to a task is shown in Figure 6.16, “Referencing a memory region within a task in EB tresos Studio”.


Os

Name: Os

General | OsAlarm | OsApplication | OsAppMode | OsCounter | OsIoc | OsEvent | OsIsr | OsOS | OsResource | OsScheduleTable | OsTask | OsMicrokernel

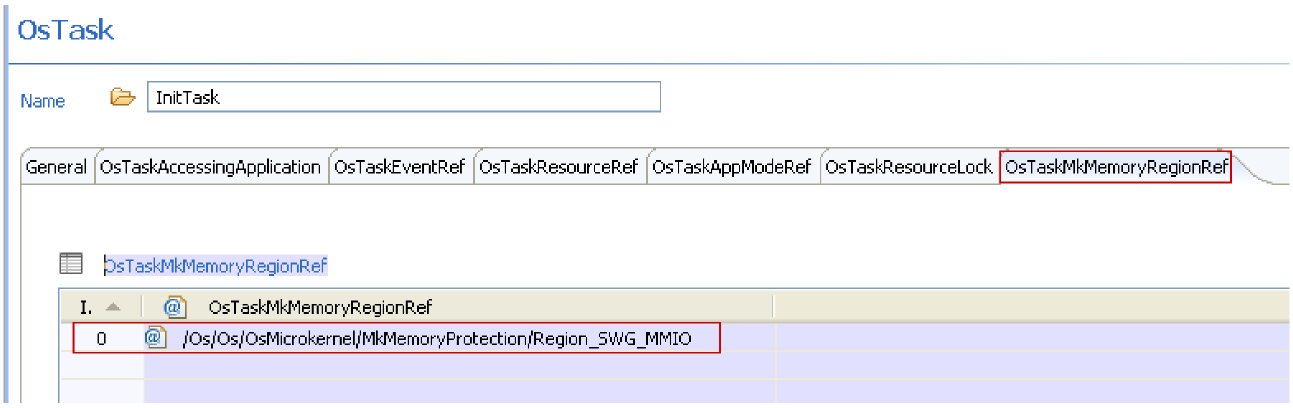
MkMemoryProtection

MkMemoryProtection

MkMemoryRegion

I.	Name	MkMemoryRegionFlags	MkMemoryRegionInitialize
0	Region_SWG_MMIO	READ_WRITE	☐

Figure 6.15. Configuration of additional memory regions with EB tresos Studio


OsTask

Name: InitTask

General | OsTaskAccessingApplication | OsTaskEventRef | OsTaskResourceRef | OsTaskAppModeRef | OsTaskResourceLock | OsTaskMkMemoryRegionRef

OsTaskMkMemoryRegionRef

I.	OsTaskMkMemoryRegionRef
0	/Os/Os/OsMicrokernel/MkMemoryProtection/Region_SWG_MMIO

Figure 6.16. Referencing a memory region within a task in EB tresos Studio

6.8.2. Examples

You can use the fine-grained control of the memory protection configuration for different purposes. Two typical use cases are shown in this section.

6.8.2.1. Restricting access to peripherals

It is often desirable to restrict access to peripherals to selected *OS-objects*. If the input/output-space of a peripheral is memory-mapped, i.e. the peripheral is accessed by using a dedicated portion of the memory address space, then memory protection can be applied to limit the access to selected peripherals.

To achieve this, create a dedicated memory region that refers to the memory-mapped IO-space of the peripheral in question. Reference this memory region in the selected *OS-objects*. Ensure, that no other memory region allows access to the peripheral.

NOTE

Additional provisions to OS-objects

Depending on the microcontroller and on the peripheral, additional provisions may be necessary, e.g. executing the OS-object in *privileged mode*. This example assumes that *non-privileged mode* is sufficient. Likewise, other hardware mechanisms like peripheral protection may provide additional protection and may need to be configured independently from this example.

6.8.2.2. Exchanging data between OS-objects

You can exchange data using the shared memory regions assigned to different *OS-objects*. When you do this you must ensure the consistency of the data. This section gives example approaches.

If data has a source and a sink, a well-suited approach is the *producer-consumer* design pattern. One OS-object acts as *producer* and generates data into a shared region. The *consumer* reads data from the shared region and processes it further.

You must take into account two particularities with the producer-consumer design pattern:

1. The memory shared between the producer and consumer must be mapped to memory regions.
2. The shared data must be kept consistent, for example by using some form of synchronization.

To map data to memory regions, you must first know how the data is accessed. The producer-consumer pattern is fit for the following access scheme:

- The producer **writes** data into a shared region.
- The consumer **reads** data from the shared region.

In particular, no write access to the shared region is needed for the consumer.

TIP

Acknowledgment of data

If the consumer needs to communicate back to the producer, you can use a memory region that is writable by the consumer and read-only for the producer. If the message is small enough to fit into a single bit, events can be applied instead. For more information, see Section 6.7, “Events”.

For information on how you can map this access scheme to memory regions, see Figure 6.17, “Using memory regions to share data”: One memory region is referenced from the producer. The access rights of this region are set to read/write. Another memory region is referenced by the consumer with read-only access rights. Both regions refer to the same address range and differ only in the respective access permissions.

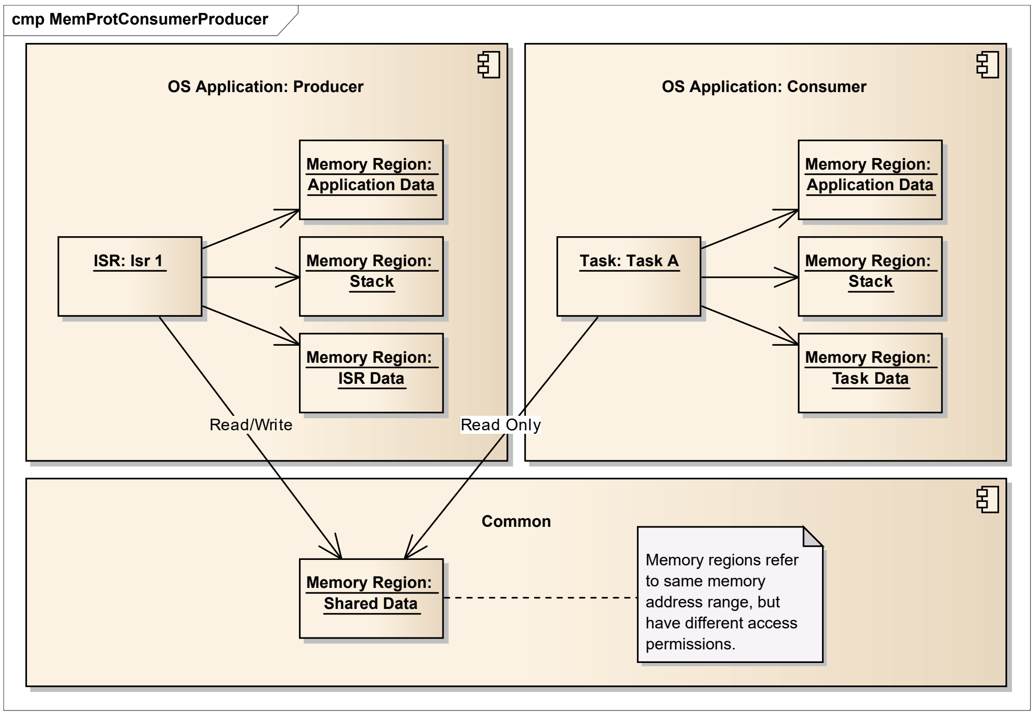

cmp MemProtConsumerProducer
OS Application: Producer
Memory Region: Application Data
ISR: Isr 1
Memory Region: Stack
Memory Region: ISR Data
OS Application: Consumer
Memory Region: Application Data
Task: Task A
Memory Region: Stack
Memory Region: Task Data
Read/Write
Read Only
Common
Memory Region: Shared Data
Memory regions refer to same memory address range, but have different access permissions.

Figure 6.17. Using memory regions to share data

NOTE

RAM read-access

You may want to use one region that is shared by all *OS-objects* and grants read-access to the entire RAM. In that case, no distinct memory region for the consumer is required. EB tresos Studio automatically produces such a region, see `MK_GlobalRam` in Section 7.4.2.9.3, “Memory regions when using EB tresos Studio”

There are several mechanisms to keep the data between the consumer and producer consistent:

- You can use locks for mutual exclusion of producer and consumer.
- You can use appropriate scheduling settings to guarantee mutual exclusion of producer and consumer.
- You can use data structures and access patterns that provide a consistent view even if accessed in parallel from producer and consumer. This approach is not described in this example section.

Using locks for mutual exclusion

Lock is a placeholder and covers several different mechanisms you can use to ensure mutual exclusion. These mechanisms include resources, interrupt locks, spinlocks and combined locks. For more information, see also Section 6.6, “Synchronizing access to common data”. Each of these mechanisms has their own field of application and you must choose them appropriately. If the producer and consumer are on the same core, i.e. both are in single-core as well as in multi-core systems, resources and interrupt locks shall be used while spinlocks shall be used, if the producer and the consumer are assigned to different cores in a multi-core system. Combined locks are intended for the situation of several producers and/or consumers which are distributed over several cores as well as on the same core.

In order to use a lock, configure an appropriate lock for the producer and the consumer. Prior to accessing the critical section, call the corresponding lock acquisition function, e.g. `GetResource()` in case of a resource. Call the lock release function after the critical section is left, e.g. `ReleaseResource` in case of a resource. This ensures that the producer and consumer cannot enter the critical section at the same time.

Using appropriate scheduling settings to guarantee mutual exclusion of producer and consumer

You can apply this approach, if both the producer and the consumer are *tasks* and those tasks are assigned to the same core. If both tasks are marked as non-preemptive, they cannot preempt each other asynchronously. Likewise, if the same scheduling priority is selected for both tasks, they cannot preempt each other at all. In both cases, pseudo-parallel access of the shared data from producer and consumer is avoided.

It is recommended to use locks for mutual exclusion since this provides a better maintainability, you can restrict the lock to the critical section and there are locks for each scenario, especially critical sections in multi-core systems. On the other hand, if you use scheduling settings this may result in improved performance, since no system calls are necessary.

6.9. Timing protection

To enforce timing protection in an AUTOSAR-compliant operating system, limit the execution time that is allocated to each task or ISR. To do this, enforce two limitations:

- The execution time of each invocation of the task or ISR
- The frequency of activation for each task or ISR

The intention is to guarantee that there is sufficient processor time available to meet all real-time deadlines. However, the scheduling analysis needed for such a guarantee is not trivial to perform.

Furthermore, these limitations alone would be insufficient to ensure that a system is running correctly, because there is no mechanism to apply a lower limit to the frequency of activation. For this reason, in many cases deadline monitoring is preferred over AUTOSAR timing protection.

Nevertheless, the microkernel provides some support for AUTOSAR timing protection. The microkernel also provides some low-level services that you can use to construct other forms of timing protection. The timing protection features provided by the microkernel are *time services* and *execution budget monitoring*. For more information, see Section 6.9.1, “Time services” and Section 6.9.2, “Execution time budgets”.

6.9.1. Time services

The microkernel's time services are based on a long-duration timer that is either provided directly by the hardware or derived from a hardware timer of shorter duration. The duration of time that you can measure via the long-duration timer is designed so that it should never be necessary to consider the effect of an overflow of the timer.

To achieve this feature, the microkernel uses a 64-bit data type called `mk_time_t` to manipulate raw time values. The data type is described in the reference section.

When you manipulate shorter intervals of time it is often more convenient to use a 32-bit data type. For this purpose the microkernel uses a basic unsigned integral type called `mk_uint32_t`.

The microkernel provides a set of services that you can use to obtain the time and work with absolute time and intervals of time. The services are described briefly in the following paragraphs and in full detail in Section 8.2, “Microkernel API reference”. You can call all of these services from threads running in any processor mode. No special privileges are necessary.

MK_ReadTime. To access the raw timer value, the microkernel provides the service MK_ReadTime. Time values obtained by calling this service can be viewed as having started at zero some time in the past. They continue increasingly until the next time the system is restarted. The exact range of the timer depends on the hardware and its configuration, see Section 9.6, “Timers on TriCore family processors”.

MK_DiffTime. To calculate the 64-bit difference between two times, the microkernel provides the service MK_DiffTime.

MK_ElapsedTime. To obtain the time that has elapsed since a given time, the microkernel provides the service MK_ElapsedTime. This service combines `MK_ReadTime` and `MK_DiffTime` and updates the *given time* so that repeated calls with the same variable return the time elapsed between the calls.

MK_DiffTime32. To calculate the 32-bit difference between two times, the microkernel provides the service MK_DiffTime32. The value provided by this service saturates at the largest possible value. Thus, when you monitor the time between events, you cannot confuse a long duration that is outside the permitted limits with a short duration that is within the limits.

MK_ElapsedTime32. To obtain the time that has elapsed since a given time, the microkernel provides the service MK_ElapsedTime32. This service combines `MK_ReadTime` and `MK_DiffTime32` and updates the *given time* so that repeated calls with the same variable return the time elapsed between the calls.

In addition to the above services the microkernel provides the following old convenience services for compatibility:

- MK_ElapsedMicroseconds
- MK_ElapsedTime1u
- MK_ElapsedTime10u
- MK_ElapsedTime100u

WARNING

Overflow protection

The services `MK_ElapsedMicroseconds`, `MK_ElapsedTime1u`, `MK_ElapsedTime10u` and `MK_ElapsedTime100u` are deprecated and do not provide overflow protection. Use `MK_ElapsedTime` or `MK_ElapsedTime32` instead.

The microkernel provides conversion macros that you can use to convert between timer ticks and nanoseconds.

6.9.2. Execution time budgets

You can give an execution time budget to all threads that you configure in the microkernel. The budget specifies the maximum time for which the thread can occupy the processor for a single invocation of the thread, where *invocation* is defined to be an activation or a successful call to WaitEvent or MK_WaitGetClearEvent.

The microkernel counts only the time that the thread actually occupies the CPU against the thread's budget. Time spent while waiting for the CPU, for example, while a higher priority thread is running, is not counted. This means that time spent in trusted functions or calls to QM-OS services does not count against the execution budget of the caller.

If a thread's execution budget expires before the thread terminates or calls `WaitEvent()` or `MK_WaitGetClearEvent()`, the microkernel reports a protection fault, which results in the protection hook being called.

There are no microkernel services associated with execution budgets.

6.9.3. Time conversion

The microkernel provides macros and functions to convert between time values specified in nanoseconds and equivalent tick values for clocks running at a range of commonly-used frequencies.

You can find the frequencies for which conversion is available in the header file `Mk_timeconversion.h`. This section describes the general pattern.

For each supported frequency *f*, the microkernel defines four macros:

MK_NsToTicks_*f*(ns). converts the parameter `ns` nanoseconds to the equivalent number of ticks at the given frequency, without overflow in intermediate calculations. This macro can be evaluated at compile time and so you can use it to initialize constants.

MK_TicksToNs_*f*(tk). converts the parameter `tk` ticks of the given frequency to the equivalent number of nanoseconds, without overflow in intermediate calculations. If the resulting value is too large to be represented by a 32-bit unsigned quantity, the macro evaluates to `0xffffffff`. This macro can be evaluated at compile time and so you can use it to initialize constants.

MK_NsToTicksF_*f*(ns). converts the parameter `ns` nanoseconds to the equivalent number of ticks at the given frequency, without overflow in intermediate calculations. The macro might use the MK_MulDiv function for better range and accuacy, and so is not guaranteed to evaluate at compile time.

MK_TicksToNsF_*f*(tk). converts the parameter `tk` ticks of the given frequency to the equivalent number of nanoseconds, without overflow in intermediate calculations. If the resulting value is too large to be represented by a 32-bit unsigned quantity, the macro evaluates to `0xffffffff`. The macro might use the MK_MulDiv function for better range and accuracy. Thus it does not guarantee to evaluate at compile time.

The requirement to evaluate at compile time using 32-bit unsigned arithmetic without overflow means that the macros `MK_NsToTicks_`*`f`* and `MK_TicksToNs_`*`f`* can introduce rounding errors in the intermediate calculations. The errors usually occur for input values that are not a round number, e.g. a multiple of 10 or 100. The microkernel's implementation attempts to minimize these errors. Nevertheless, when it deals with values that cannot be predicted by the design, e.g. to convert tick values obtained at run-time to nanoseconds, you may prefer to use `MK_NsToTicksF_`*`f`* or `MK_TicksToNsF_`*`f`*, which gives better accuracy at the expense of a little execution time.

6.10. API access protection

The microkernel provides access protection mechanisms for the following APIs:

- `ShutdownOs()`
- `ShutdownAllCores()`
- `TerminateApplication()`

For the `ShutdownOs()` and `ShutdownAllCores()` API there are application configuration parameters (`/Os/OsApplication/OsAppMkPermitShutdownOS` and `/Os/OsApplication/OsAppMkPermitShutdownAllCores`) which allow the access to the corresponding API function for all tasks and ISRs which are part of that application.

The access to `TerminateApplication()` is handled differently: for each application, the applications which are allowed to access it are configured in the list `/Os/OsApplication/OsAppAccessingApplication`. If an application `app1` is configured to have access to an application `app2`, the tasks and ISRs of application `app1` are allowed to call `TerminateApplication()` for application `app2`.

In both cases if the access to an API is restricted, the API aborts and reports the error `MK_eid_WithoutPermission`. Otherwise, the API continues and performs the intended functionality.

6.11. Simple schedule table

You can use the simple schedule table (SST) service to perform the basic operations of an AUTOSAR schedule table. It is based on the concept of a counter that you can increment by your tasks or ISRs, or automatically in response to a regular timer interrupt.

Each counter has exactly one schedule table, which is fixed by the configuration. Each schedule table has a set of expiry points.

At each expiry point on the schedule table you can activate one or more tasks or set events for previously-activated tasks.

You can start, stop, and advance counters and therefore their associated schedule tables from tasks and other threads by means of the API that is described here.

You can configure as many counters as you need, but the number of counters that you can increment automatically is limited by the number of hardware timers that are available.

The SST service implements a subset of the features specified by AUTOSAR. The differences between the SST and a standard AUTOSAR schedule table are:

- Each SST counter only supports one schedule table.
- Synchronization with an external time source is not available.
- Single-shot operation is not implemented.
- The SST has no equivalent to the `NextScheduleTable()` API in AUTOSAR.
- No *autostart* facility. The counter must be started using the API.

6.11.1. Configuring a SST in EB tresos Studio

The SST is best configured in EB tresos Studio. To do this, add an `OsCounter` object to your configuration and set its parameters as follows:

- Set **OsCounterType** to **SOFTWARE**.
- Set **OsMaxAllowedValue** to one less than the desired duration of your schedule table.
- Set **OsCounterMinCycle** to `1`.

If you want your counter to be incremented automatically at regular intervals:

- Select a hardware timer from the list under **OsHwModule**.
- Enter the desired interrupt level.

- Set **OsSecondsPerTick** to the tick interval that you need.

When you have configured your counter, you need to attach a schedule table to it:

- Create a schedule table as you would do for a standard OS schedule table.
- Add expiry points to the schedule table.
- Attach the schedule table to your counter.
- Set **OsScheduleTableDuration** to one greater than the **OsMaxAllowedValue** of the counter.
- Set **OsScheduleTableRepeating** to true.
- Set **OsScheduleTableIsSimple** to true.

The offsets of the expiry points must lie in the range from `0` to the duration. If there is an expiry point with an offset equal to the duration, it expires on the same tick as the expiry point at `0` in the next round. However, the expiry point with an offset equal to the duration occurs logically before the expiry point at `0`, and does not occur at time `0` in the first round after starting the SST.

The SST configuration is checked by the microkernel during startup. If an error in the SST configuration is detected, a startup panic is used to report this. For more information on panics, see Section 7.3.4, “Panics”. This means that one or more parameters of a counter configuration are out of range.

6.11.2. Calling the SST module

To call the SST module, include the header file `public/Mk_sst_api.h` at the top of your source file. The API declared in this header file is for use in any thread such as a task or ISR, but also for use in the `main` function.

To start a counter and its associated schedule table, call `MK_SstStartCounter(counterId,delay)`. The first parameter, `counterId` identifies the SST for which you want to start the counter. You can use the name you gave your SST here, because an appropriately defined macro is generated by EB tresos Studio. The second parameter, `delay`, is the number of ticks of the counter that elapses before the first round of the schedule table starts. The delay must be in the range `0` to the duration of the SST, which corresponds to the counter's modulus.

To stop a counter and its associated simple schedule table, call `MK_SstStopCounter(counterId)`. The parameter is the counter ID as described for `MK_SstStartCounter`.

To advance a counter and cause expiry points on the associated simple schedule table to be processed, call `MK_SstAdvanceCounter(counterId,nTicks)`. The first parameter is the counter ID as described for `MK_SstStartCounter`. The second parameter, `nTicks`, is the number of ticks by which to advance the counter, and must be in the range `1` to (duration-1). Furthermore, the counter must not have a hardware-dependent ticker, i.e. `tkr` of the respective `MK_SSTCOUNTERCONFIG()` invocation must be `-1`.

Comprehensive descriptions of the API functions, including all the possible error codes that they can return, are given in Chapter 8, “Microkernel reference manual“.

6.11.3. Error handling

The SST API functions report errors by returning a `StatusType` result code. For `MK_SstAdvanceCounter()` the result code may also include errors encountered on the *first failed* activate task or set event operation of the set of expiry point actions, which become due when a SST counter is advanced. Whether the `ErrorHook` feature of the EB tresos Safety OS is invoked depends on the core assignment of the task, which is activated or for which events are set.

The `ErrorHook` is not invoked, when the SST runs on the same core as the task. This means that dynamic errors detected while processing the ticker interrupt (for example, attempting to activate a task that is already active) are silently ignored.

On the other hand, when the task activated or for which events are set, is located on a different core than the SST, then the `ErrorHook` is invoked on that core. The error code passed to the `ErrorHook` reports any errors concerning task activation or event setting. The result code returned by the SST API functions is related to the message passing between the cores.

If as part of its normal operation the ticker interrupt function detects that the scheduled time of the next interrupt is already in the past, it invokes the `ProtectionHook()` with the protection error `MK_E_INTHEPAST`. Whether this is detected depends on the characteristics of the hardware.

If the ticker hardware has an automatic reload feature, the SST is unlikely to detect the error. In this case, if an interrupt is delayed by more than one interval, counter ticks are lost and expiry actions are processed later than expected.

The SST does not actively measure the time between successive interrupts to detect deadline violations. It is therefore important to use a timing protection mechanism to detect deadline violations and other timing errors that cause your system to function incorrectly.

7. Using the microkernel

7.1. Preparing your application

Develop your application by following the guidelines in the EB tresos AutoCore OS documentation [ASCOS_USERGUIDE]. Configure the `OS` module to provide the set of *tasks*, *ISRs*, and other *OS-objects* that you need.

The configuration in EB tresos Studio of a system based on the microkernel is not different from the configuration for EB tresos AutoCore OS. The microkernel simply ignores the configuration of features such as task- or application-specific hooks that it does not implement. Nevertheless, EB recommends that you do not configure your system with such features enabled.

If you want to use the protection features provided by the microkernel you should develop your application to scalability class 3 or 4 with memory protection. For optimum freedom from interference you should not configure any *trusted OS-Applications*. If there is a function that requires a high privilege level to access hardware, you should configure this function as a trusted function.

A list of all implemented services is given in Section 8.2, “Microkernel API reference”. Ensure that your application only uses AUTOSAR services that are implemented in EB tresos Safety OS.

NOTE

Verify the EB tresos Safety OS configuration generated by EB tresos Studio
If you use EB tresos Studio to configure EB tresos Safety OS, verify the generated configuration according to the criteria given in the EB tresos Safety OS safety manual for TriCore family [SAFETYMANTRICORE].

7.2. Using the microkernel API

The microkernel supports a subset of the AUTOSAR-OS standard. The microkernel also provides an isolation layer between your tasks and ISRs and the services of the QM-OS. The subset of the AUTOSAR-OS services that is supported by the microkernel is described in Section 8.2, “Microkernel API reference”.

The microkernel manages all your executable objects, i.e. tasks, ISRs, and hook functions as instances of objects called *threads*. This uniformity means that you can call most of these services from any thread, including your hook functions, and expect that they work. For example, `TerminateTask()` always terminates the caller, regardless of whether it is a task, an ISR, or a hook function.

The only restrictions imposed by the microkernel are:

- `WaitEvent()`, `MK_WaitGetClearEvent()`, and `ClearEvent()` can only be called from an EXTENDED task thread, because only EXTENDED tasks have the event status variables that are needed for these services.
- Calls to QM-OS services that are invoked via the microkernel thread interface cannot be called from hook functions or from category 1 ISRs, because QM-OS threads are not permitted to gain a higher priority than these threads.

The lack of restrictions means that the microkernel does not fully conform to the AUTOSAR standard. However, a system that is developed to run under a conforming AUTOSAR system runs without problems under the microkernel provided that the unimplemented services are not called.

NOTE

Using API functions outside their intended use

Keep in mind that being able to do something does not mean that it is meaningful to do that. For example if you call `TerminateTask` in the `ProtectionHook()` the `ProtectionHook()` thread is terminated. However, in that case the return value of the `ProtectionHook()` is undefined as is the further execution of the system based on this return value.

Make sure that you understand the consequences if you use API functions outside of their intended use case.

7.3. Error handling

Error handling in EB tresos Safety OS is divided into three categories:

- Errors, described in Section 7.3.1, “Errors”.
- Protection faults, described in Section 7.3.2, “Protection faults”.
- Panics, described in Section 7.3.4, “Panics”.

The standard AUTOSAR callout functions `ErrorHook()`, `ProtectionHook()`, and `ShutdownHook()` are supported by the microkernel. Application-specific `ErrorHook()` and `ShutdownHook()` functions are not supported. You must supply the hook functions. Their use is described in Section 8.3, “Microkernel callout reference”.

Unlike standard AUTOSAR systems, the `ErrorHook()`, `ProtectionHook()`, and `ShutdownHook()` functions are not called directly by the microkernel. Instead, they are started in high-priority threads. This means that the functions run with the processor mode and memory protection boundaries that are configured for them.

In a multi-core environment, the hook function runs on the core on which the error is detected, which in the case of the error hook may not be the same as caller of the API.

7.3.1. Errors

An error is detected when a thread calls an API function of EB tresos Safety OS with an incorrect parameter, or when the current state of the system means that EB tresos Safety OS cannot perform the request. Examples of such errors are:

- A thread passes an out-of-range ID parameter to a service.
- A thread attempts to activate a task whose configured maximum number of simultaneous activations is already reached.
- A thread attempts to acquire a resource that is already occupied up to the nesting limit which is `1` for AUTOSAR.

When EB tresos Safety OS detects an error, the microkernel fills the error information structure with the information about the error. Then EB tresos Safety OS invokes the `ErrorHook()` if it is enabled in the configuration, and passes the AUTOSAR-style error code as the parameter. On completion of the `ErrorHook()` control returns to the thread that causes the error and, in most cases, the API function returns the AUTOSAR-style error code. The error codes and return values for API functions are described in their reference pages in Section 8.2, “Microkernel API reference”.

A typical invocation of the `ErrorHook()` is depicted in Figure 7.1, “Invocation of the `ErrorHook()`”.

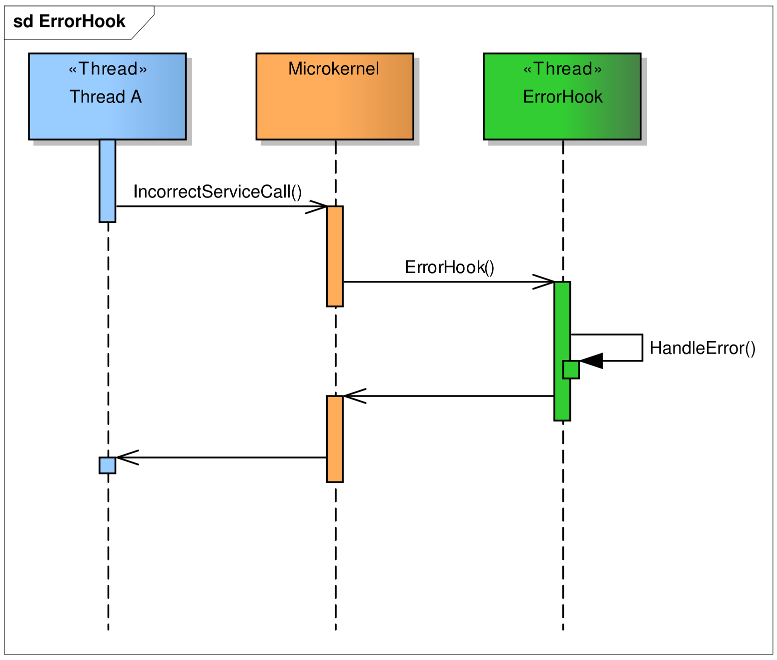


Figure 7.1. Invocation of the `ErrorHook()`

The error information structure contains information that is used by the OSEK-standard error information services, e.g. `OSErrorGetServiceId()` etc. plus additional information that can help to identify the exact cause of the error. The error information structure is described in Error information.

If the `ErrorHook()` causes another error, the `ErrorHook()` is neither started for the second error, nor is the error information structure filled. You can still use the return value to detect erroneous service calls in this situation.

If the `ErrorHook()` causes a protection fault, it is handled just like it would be handled for any other type of thread.

7.3.2. Protection faults

A protection fault is detected when a thread attempts to perform an operation that is not permitted by its configuration. Most types of protection fault are detected by the hardware and reported to the microkernel by means of an exception trap. Examples of operations that cause protection faults are the following:

- A thread attempts to use an instruction for which it does not have the required privilege.
- A thread attempts to access a memory location for which it does not have permission.
- A thread continues to execute longer than it is permitted by its configured execution budget.

When EB tresos Safety OS detects a protection fault, the microkernel fills the protection fault information structure with the information about the fault. If the protection fault is reported by a hardware exception trap, additional hardware-specific information if any is placed in a hardware-specific exception information structure. The microkernel then invokes the `ProtectionHook()` if it is enabled in the configuration, and passes the AUTOSAR-style error code as the parameter. On completion of the `ProtectionHook()`, its return value is used to determine the course of action to take. The return values from the protection hook are described in Section 7.3.3, “Protection hook return values”.

A typical invocation of the `ProtectionHook()` is depicted in Figure 7.2, “Invocation of the `ProtectionHook()`”.

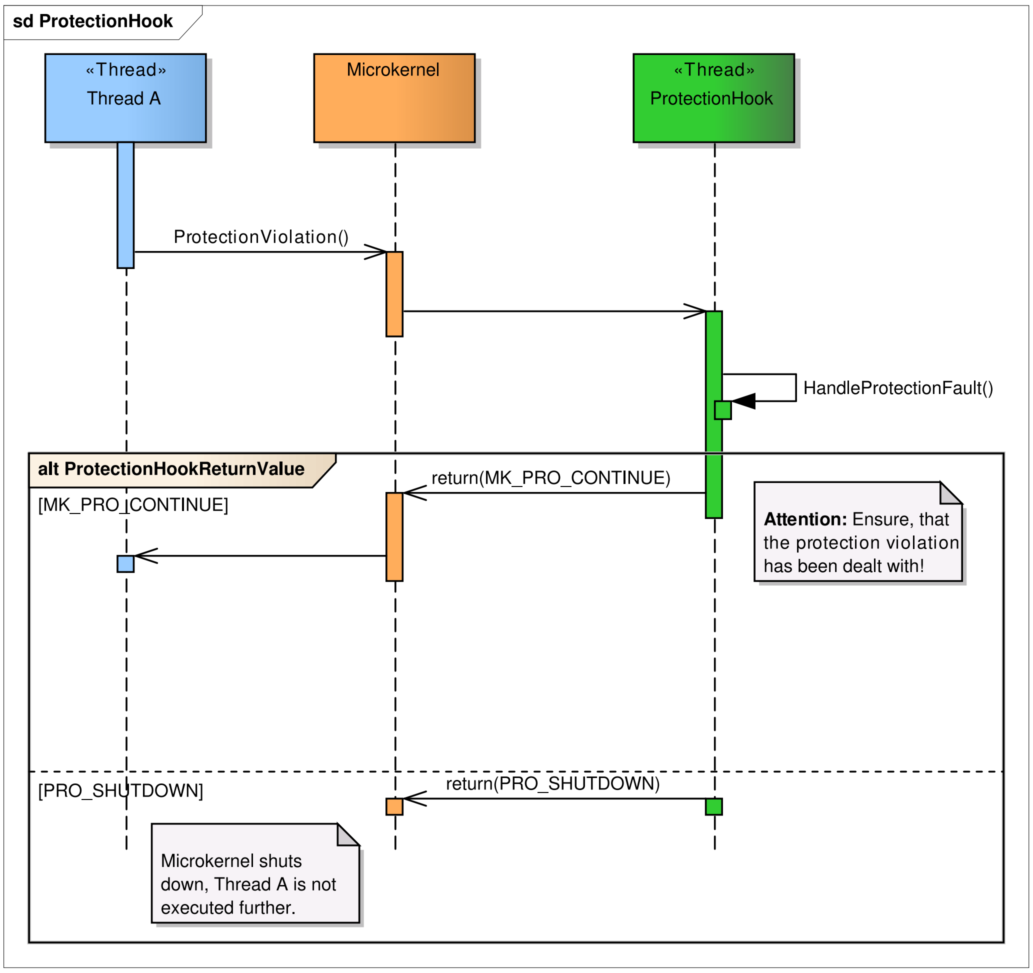


Figure 7.2. Invocation of the `ProtectionHook()`

If the `ProtectionHook()` causes an error, the `ErrorHook()` is started but does not run until the `ProtectionHook()` is completed. However, the API function that detects the error returns the error code and the error information structure is filled.

If the `ProtectionHook()` causes a protection fault, the microkernel panics. See Section 7.3.4, “Panics”.

The protection fault information structure is described in Protection fault information. If there is a hardware-specific exception information structure, it is described in MK_exceptionInfo.

7.3.3. Protection hook return values

The value returned by the `ProtectionHook()` is called the *protection action* and determines the course of action that the microkernel takes after the fault is reported.

“EB tresos Safety OS protection actions” shows the supported protection actions. The protection actions with the prefix `PRO_` are defined by the AUTOSAR standard and have the same meaning. The protection actions with the prefix `MK_PRO_` are specific to EB tresos Safety OS. If the `ProtectionHook()` returns a value that is not listed, the microkernel behaves as if the return value were `MK_PRO_INVALIDACTION`.

EB tresos Safety OS protection actions

`MK_PRO_CONTINUE`
: The microkernel takes no further action. This means that control returns to the faulty thread. In most cases the instruction that causes the protection fault is executed again, so the same fault is reported again. If the `ProtectionHook()` handles this in the same way, the result is an endless loop. If the protection fault is caused by the thread exceeding the execution budget, the fault is reported again and control does not return to the faulty thread.

`PRO_IGNORE`
: This protection action is defined by the AUTOSAR standard and is only valid for arrival rate violations. The microkernel does not provide arrival-rate monitoring, so the microkernel *always* treats `PRO_IGNORE` as invalid and behaves as if the return value is `MK_PRO_INVALIDACTION`.

`PRO_TERMINATETASKISR`
: If the faulty thread runs a task or ISR, the microkernel terminates the thread. If the thread runs some other kind of executable entity, the microkernel behaves as if the return value is `PRO_TERMINATEAPPL`.

`MK_PRO_TERMINATE`
: The microkernel terminates the faulty thread regardless of the type of executable entity that runs in it. If the executable entity is a QM-OS service or a trusted function, the status code `MK_E_KILLED` is returned to the thread that requests the service.

`PRO_TERMINATEAPPL`
: If the faulty thread belongs to an OS-Application, the microkernel terminates the OS-Application as though it calls `TerminateApplication()` with the `NO_RESTART` option. For more information, see TerminateApplication. If the faulty thread does not belong to an OS-Application, the microkernel behaves as though the return value is `PRO_SHUTDOWN`.

`PRO_TERMINATEAPPL_RESTART`
: If the faulty thread belongs to an OS-Application, the microkernel terminates the OS-Application as though it calls `TerminateApplication()` with the `RESTART` option. For more information, see TerminateApplication. If the faulty thread does not belong to an OS-Application the microkernel behaves as though the return value is `PRO_SHUTDOWN`.

`PRO_SHUTDOWN`
: The microkernel shuts down the system. It is equivalent to call `ShutdownOS()` with the same error code that is passed to the protection hook.

`MK_PRO_PANIC`
: The microkernel behaves as if it detects a serious problem by itself. The panic reason `MK_panic_PanicFromProtectionHook` is used. The subsequent behavior is described in Section 7.3.4, “Panics”.

`MK_PRO_PANICSTOP`
: The microkernel calls its internal function `MK_PanicStop()` with the panic reason `MK_panic_PanicFromProtectionHook`. The subsequent behavior, an endless loop with interrupts disabled, is de-

scribed in Section 7.3.4, "Panics". You should use this protection action only in extreme circumstances. For example, if the `ProtectionHook()` determines that the integrity of the hardware is so badly compromised that there is no other safe course of action.

`MK_PRO_INVALIDACTION`
: `MK_PRO_INVALIDACTION` is defined by the microkernel in order to implement defined behavior if the `ProtectionHook()` returns an out-of-range protection action. You should never design the `ProtectionHook()` function to return `MK_PRO_INVALIDACTION`. If the `ProtectionHook()` returns `MK_PRO_INVALIDACTION` or any other value which is not listed, the microkernel behaves as though the return value is `PRO_SHUTDOWN`. No error is reported for the invalid return value.

WARNING

Wrong return values of the protection hook can violate availability requirements

Incorrect return values can violate availability requirements. You must consider the return values of the protection hook carefully during system design.

7.3.4. Panics

Panic is a term that is used for the microkernel's behavior when a serious problem is encountered. Examples of serious problems are:

- The microkernel detects that its internal state is inconsistent.
- The microkernel detects that it caused an exception.
- The `ProtectionHook()` causes a protection fault.
- The `ProtectionHook()` returns `MK_PRO_PANIC`.

The first time a serious problem is encountered, the microkernel stores the panic reason and starts a shutdown sequence as described in Section 7.6, "Shutdown sequence". The error code that is passed to the `ShutdownHook()` in this case is `MK_E_PANIC`. You can retrieve the panic reason in the `ShutdownHook()` by calling MK_GetPanicReason or MK_GetPanicReasonForCore.

The second time and for all subsequent times that a serious problem is encountered, the microkernel calls its internal function `MK_PanicStop()` with the reason for the new panic as parameter. `MK_PanicStop()` is an endless loop. When you examine the parameter in the CPU registers, you can examine the second panic reason in the debugger.

NOTE

Panics in multi-core environments

The microkernel stores the panic reason for each core separately. Therefore `MK_PanicStop()` is called in multi-core environments only, if the panic is reported on a core, on which another panic was previously reported.

You can see from the above description that the microkernel only calls `MK_PanicStop()` as a last resort when a fault is detected and all attempts to notify the application, by means of the `ProtectionHook()` and `ShutdownHook()`, result in more detected faults. The only other way to call `MK_PanicStop()` is if you explicitly request a protection action by returning `MK_PRO_PANICSTOP` from the `ProtectionHook()`. For more information, see Section 7.3.3, “Protection hook return values”. The safety analysis of EB tresos Safety OS assumes that a dead stop is safe, as implemented in `MK_PanicStop()`. Alternatively the safety analysis assumes that an external mechanism resets the system or makes it safe if it stops responding.

7.3.4.1. Panics during startup

You only encounter serious problems during startup if the microkernel's configuration is faulty or the processor's safety integrity is compromised.

During startup, the microkernel cannot shut down in a normal way because the thread scheduler is not available. Serious problems that are detected during startup are handled in a similar way to multiple serious problems during normal operation. The only difference is that you can configure the handler function `MK_StartupPanicStop()` to call a user-defined function before its endless loop. However, there are strict restrictions on what this callout function can do. In particular, it must not call any EB tresos Safety OS service, either directly or indirectly. This precludes calls to AUTOSAR modules.

7.3.4.2. Exception during kernel execution

If an exception occurs during the execution of the kernel this leads to a *panic* and consequently to a shutdown of the system. In most cases this is caused by a configuration error, e.g. an incorrect kernel memory region. Non-maskable interrupts or ECC errors can also cause in-kernel exceptions.

Further information about the exception is available to the `ShutdownHook()`. If the panic reason is `MK_panic_ExceptionFromKernel` you can obtain information about the exception via MK_GetPanicExceptionInfo. The values obtained by MK_GetExceptionInfo are undefined.

7.4. Configuring the microkernel manually

This section describes how to configure the microkernel manually. If you use the output of the EB tresos Safety OS generator without modification you can ignore this section. You need to read this section if you create your own configuration files without using the EB tresos Safety OS generator. If you modify the configuration files that are generated, you need only refer to the appropriate reference tables in Section 8.4, “Microkernel configuration reference”.

A microkernel configuration consists of the three header files `Mk_gen_user.h`, `Mk_gen_config.h`, and `Mk_gen_addons.h`, and the source file `Mk_gen_global.c`.

The header files `Mk_gen_user.h` and `Mk_gen_config.h` are included by assembler source files as well as by C-source files, so it is important that you prevent the assembler from parsing C-constructs. To do this, place `#ifndef MK_ASM ... #endif` pairs around each group of C-constructs.

Example:

```
#ifndef MK_ASM
extern mk_stackelement_t stack42[200];
#endif
```

You can use the standard practice to prevent multiple inclusions, but in normal cases the files are protected by the header files that include them.

7.4.1. Mk_gen_user.h

The `Mk_gen_user.h` file contains the definitions that are needed by the application code. Place the definitions of macros that represent the task, ISR, resource, and add-on identifiers[1] and event masks in this file.

Allocate the object identifiers separately for each type of object. The identifiers are consecutive integers which start at zero. They are used as indices into the object tables defined in `Mk_gen_config.h`. The order is not important.

Ensure that your event masks are unique in every task that uses the event. The easiest way to do this is to include the name of the task in the name of each event so that each task's events are independently defined. It is always a point of unexpected behavior if events are sent to a task that does not handle it, even if the event mask is a mask that is handled by the task. If this is not possible you might need to devise a method of allocating the events. An event is a 32-bit unsigned value with exactly one bit set to `1`, so each task can react to up to 32 events.

You should also place the function prototypes for your task and ISR functions in this file. If you use the same naming scheme as the EB tresos Safety OS generator, you should name your functions `OS_TASK_<task-name>` for tasks and `OS_ISR_<isr-name>`. These names are compatible with the AUTOSAR `TASK()` and `ISR` construction macros respectively.

Do not use the same name for two objects, even if those objects are of a different type.

[1]For more information on the expected add-on identifiers, see the documentation of the used add-ons.

Example:

```
/* Tasks */
#define My2msTask           0
#define MyEventHandler      1
#define My10msTask          2

#ifndef MK_ASM
void OS_TASK_My2msTask(void);
void OS_TASK_EventHandler(void);
void OS_TASK_My10msTask(void);
#endif

/* ISRs */
#define MyCanIsr            0
#define MyGptIsr            1

#ifndef MK_ASM
void OS_ISR_MyCanIsr(void);
void OS_ISR_MyGptIsr(void);
#endif

/* Events */
#define Ev_CanRx           0x00000001u
#define Ev_GptTick         0x00000002u
#define Ev_RunMode         0x00000004u
#define Ev_StandbyMode     0x00000008u

/* Add-ons */
#define MK_ADDON_ID_IOC    0u
```

Locks are a little different. On microcontrollers with more than one core, configured `OsResource` objects are bound to a specific core and have their core index encoded as part of the ID. This encoding is performed by the macro `MK_MakeLockId(coreIndex, resourceIndex)`. Locks such as interrupt locks, spinlocks, the scheduler resource `RES_SCHEDULER` if present and EB-vendor-specific combined locks, must occupy the lowest IDs and are defined without the core index.

Example:

```
/* Resources */
#define MK_RESCAT1       0
#define MK_RESCAT2       1
#define RES_SCHEDULER    2
#define MySpinlock       3
```

```
#define MyCombi          4

#define MyCanBufferCore0Mutex  MK_MakeResourceId(0, 5)
#define MyGptCore1Mutex        MK_MakeResourceId(1, 5)
```

`Mk_gen_user.h` is included by `MicroOs.h`.

7.4.2. Mk_gen_config.h

`Mk_gen_config.h` is included by `Mk_Cfg.h`.

The `Mk_gen_config.h` file contains the definitions that are needed by the microkernel. Place the definitions of macros to construct the task, ISR, and resource tables in this file. For a precise description of each type of object and how to configure the object tables for the microkernel, see the reference Section 8.4, “Microkernel configuration reference”.

7.4.2.1. Allocating priorities

This section describes how to calculate the queueing priorities for your tasks and ISRs from the nominal relative priorities that your design demands. The term *priority* in this subsection refers to the configured *queueing priority* of a thread. The configured *running priority* of a thread is determined by the thread's relationship with other threads and is described in Section 7.4.2.7, “Determining the running priority”.

You must allocate queueing priorities strictly in the following order:

```
Idle < Tasks <= Scheduler < ISR (cat2) <= Cat2 Lock <
ISR (cat1) <= Cat1 Lock < ErrorHook() < ProtectionHook()
```

The idle thread has the queueing priority `0`. Priorities are integers in the range 0..32767, where a higher number represents a higher priority.

You must allocate ISR priorities in the same order as their hardware *level*. Bear in mind that some processors use a descending number to represent a higher level of interrupt request.

You might find it useful to allocate a fixed range of priorities for each type of thread. A typical scheme might be the following:

Object	Priority
Idle thread	0
Task threads	1 .. 1021

Object	**Priority**
Scheduler	1021
ISR (category 2)	1024 .. 2045
Category 2 lock	2045
ISR (category 1)	4001 .. 4093
Category 1 lock	4093
`ErrorHook()`	4094
`ProtectionHook()`	4095

Table 7.1. Typical object priorities

7.4.2.2. Core allocation

If you configure a single-core system you can ignore this section because all your objects are allocated to the same core.

How to allocate objects such as tasks, ISRs, and counters to the various cores on a multi-core processor is not a subject that can be covered in depth by this document. Inter-core communication and synchronization are costly activities and a poor allocation strategy reduces the performance of your system. In extreme cases the performance could be worse than if a single-core processor was used. The aim of the core allocation should therefore be to minimize the interaction between cores.

The most obvious interactions between cores occur when a thread which runs on one core calls an API function to control an object that is allocated to a different core. A task that calls `ActivateTask()` for another task which is allocated to a different core is an example for this. There are other less obvious examples, such as a call to `SyncScheduleTable()` for a schedule table that is connected to a counter object on another core.

The core allocation for alarms and schedule tables controlled by the QM-OS is determined by the core allocation of the counter object to which they are attached. The allocation of the counter object may in turn be determined by hardware constraints.

You must allocate ISRs to the same core as the corresponding interrupt source. The core or cores to which an interrupt can be directed might be constrained by the hardware.

A schedule table can activate tasks on different cores, but instead of configuring a single schedule table to control all task activations, it might be more efficient to configure a counter and a schedule table on each core. The extra overhead of the counter interrupt may well be less than the overhead of inter-core activations.

Allocation of OS objects is achieved at the level of the OS-Application. You must allocate an OS-Application to a specific core, and all the tasks, counters etc. that belong to this application are automatically allocated to that core.

Variables shared between tasks running on different cores usually require inter-core locks to prevent concurrent access. It is not possible to use efficient mechanisms such as priority, internal resources and non-preemptable tasks to avoid concurrency because these mechanisms have no effect on the other cores.

The placement of variables, even when they are not shared, can cause inter-core interaction at a hardware level such as bus contention, cache contention etc., especially if the memory architecture of your microcontroller is non-uniform.

The microkernel's own variables are arranged such that the variables that are *owned* by a core can be gathered together in areas of memory that are separate from the variables that are *owned* by the other cores. *Owned* here means *maintained by* and implies write access. The other cores only require read access to a core's variables and this separation can be enforced by memory protection.

7.4.2.3. Allocating tasks to threads

You must allocate at least one thread for each task queueing priority.

On multi-core configurations you must allocate tasks to threads individually for each core.

If you do not use tasks with multiple activations, you can allocate one thread for each task. This might give better performance but also uses more memory.

To save memory you can allocate all the tasks with the same queueing priority that run on the same core to a single thread. To do this, allocate a job queue to the thread. The job queue should be long enough to accommodate all the queued activations of all the tasks that are allocated to the thread. This is one less than the total number of activations at this priority, because one activation of a task occupies the thread before the job queue starts to fill.

You can choose between separate threads and a single thread with a job queue independently for each queueing priority.

You need a job queue for each task with multiple activations. In this case, you should allocate all other tasks with the same queueing priority to the same thread.

WARNING

Task order might not conform to AUTOSAR

If you allocated a task with priority X to a thread and allocated another task with priority X to a different thread which has a job queue, the microkernel still operates in a defined manner but the scheduling no longer conforms to AUTOSAR. It is possible in such a case that the order in which the tasks of equal priority run is not the same as their activation order, as required by AUTOSAR.

Example: Let `t1`, `t2`, and `t3` be tasks with the same priority, `t1` and `t2` be assigned to the same thread which has a job queue, and `t3` be assigned to another thread. If `t2` and `t3` are activated in that sequence while `t1` is executed, `t3` is scheduled first when `t1` terminates although `t2` was activated before. That is because job queues are checked after the threads are terminated and then a new instance of the thread is enqueued behind other threads with the same priority if there is a new job waiting in the job queue.

7.4.2.4. Allocating ISRs to threads

You can allocate all *ISRs* with a given interrupt level and queueing priority to a single thread. Job queues are not supported for ISRs by the microkernel because they are not needed.

On multi-core configurations you must allocate ISRs to threads individually for each core.

7.4.2.5. Allocating stacks to threads

You can allocate a private stack for each task and ISR, but this uses a lot of memory. It is more efficient to share stacks between tasks and ISRs if you can.

Provided that the memory protection mechanism provides fine enough granularity that you can place each stack in its own memory region, you can share stacks between BASIC tasks of the same queueing priority and between ISRs of the same priority, provided that the tasks or ISRs sharing the stack are allocated to the same core. There is no risk that two tasks or ISRs attempt to use the same stack at the same time. This is possible because the microkernel ensures the mutual exclusion of two threads on the same core with the same priority.

If two or more tasks, or two or more ISRs, share a stack, the stack must be large enough to cover the requirements of the task or ISR with the largest stack requirement.

You must allocate a private stack for each EXTENDED task. The stacks of these tasks remain in use while the task is in the WAITING state. During this state you can run any other task.

You can determine a better sharing scheme with knowledge about the scheduling and internal resources used by a task. The EB tresos Safety OS generator uses such an algorithm, but its description is beyond the scope of this manual. You can also use your knowledge about how the system operates to devise a better scheme.

The basic principle is that a stack can be shared by BASIC tasks A and B, provided that A cannot preempt B and B cannot preempt A. Clearly, if A and B have the same queueing priority this condition is satisfied. It is also satisfied if A has a higher priority than B, but B is non-preemptable. For a preemption configuration, see Section 7.4.2.7, “Determining the running priority”.

7.4.2.6. Allocating register stores for task and ISR threads

You must allocate a separate register store for each EXTENDED task, because the values of the registers like the stack are preserved while the task is WAITING.

You need one register store per thread for BASIC tasks and ISRs.

7.4.2.7. Determining the running priority

In addition to the queueing priority, each thread has a running priority. A thread's priority is elevated to the running priority when it starts to execute. It stays at that level at least until the thread terminates. The `Schedule()` service temporarily reduces the thread's priority to its queueing priority (subject to locks that are occupied) to permit lower-priority threads to run. But when `Schedule()` returns to the caller, its priority is once more elevated to at least the running priority.

The EB tresos Safety OS generator uses this dual-priority mechanism to implement non-preemptable tasks (`OsSchedule=NON`), non-preemptable ISRs (`MkNonPreemptiveISRs=true`) and internal resources. If a task is non-preemptable, its running priority is set to the Scheduler priority. Otherwise, if a task uses an internal resource, its running priority is set to the resource's ceiling priority. A preemptable task that does not use an internal resource has its running priority set to the same value as its queueing priority. A non-preemptable ISR has a running priority that is the same as the highest priority ISR of the same category.

According to the AUTOSAR standard tasks are permitted to use at most one internal resource and ISRs are not permitted to use them. This means that the ceiling priority of an internal resource cannot be higher than the scheduler priority. However, the microkernel works as expected if a thread's queueing priority is in the range of priorities allocated to ISRs, provided that you set the thread's locking level correspondingly. When you use the microkernel, it should also be clear that the restriction of one internal resource per task is an artificial restriction.

7.4.2.8. Determining the ceiling priority of a resource

The ceiling priority of a resource is simply the highest queueing priority of the set of tasks that use the resource. This applies to standard resources and internal resources. For linked resources you must find the highest queueing priority of the set of tasks that use one or more of the linked resources.

The lock level of a resource must correspond to its ceiling priority. If the priority is in the range allocated for tasks, the lock level should permit all interrupts. If the priority is in the ISR range, you must set the lock level to a value that locks out the highest-level interrupt request from the ISRs using the resource.

Linked resources are often used to work around the problem that a task might need to occupy a resource twice or more at the same time, for example in nested functions. You can also solve this problem with a single resource if you configure it with a nesting count higher than one. This is an EB-vendor-specific feature.

7.4.2.9. Configuring memory protection

To configure memory protection, assign a memory partition to each executable object. When the executable object gains the CPU, the microkernel enables its memory partition. When an executable object relinquishes the CPU, the microkernel disables its memory partition.

7.4.2.9.1. Memory regions, memory partitions, and the memory region mapping table

A *memory region* is a contiguous block of physical memory that is defined by its start and end address and the permissions that are granted.

A *memory partition* is a set of memory regions.

A *region mapping table* maps memory regions to a memory partition.

The configuration for a memory partition defines the start and length of a portion of the region mapping table. Each entry in the region mapping table is a reference to a memory region in the memory region table. This relationship between memory partitions and memory regions is shown in figure Figure 7.3, “Memory partitions and regions”.

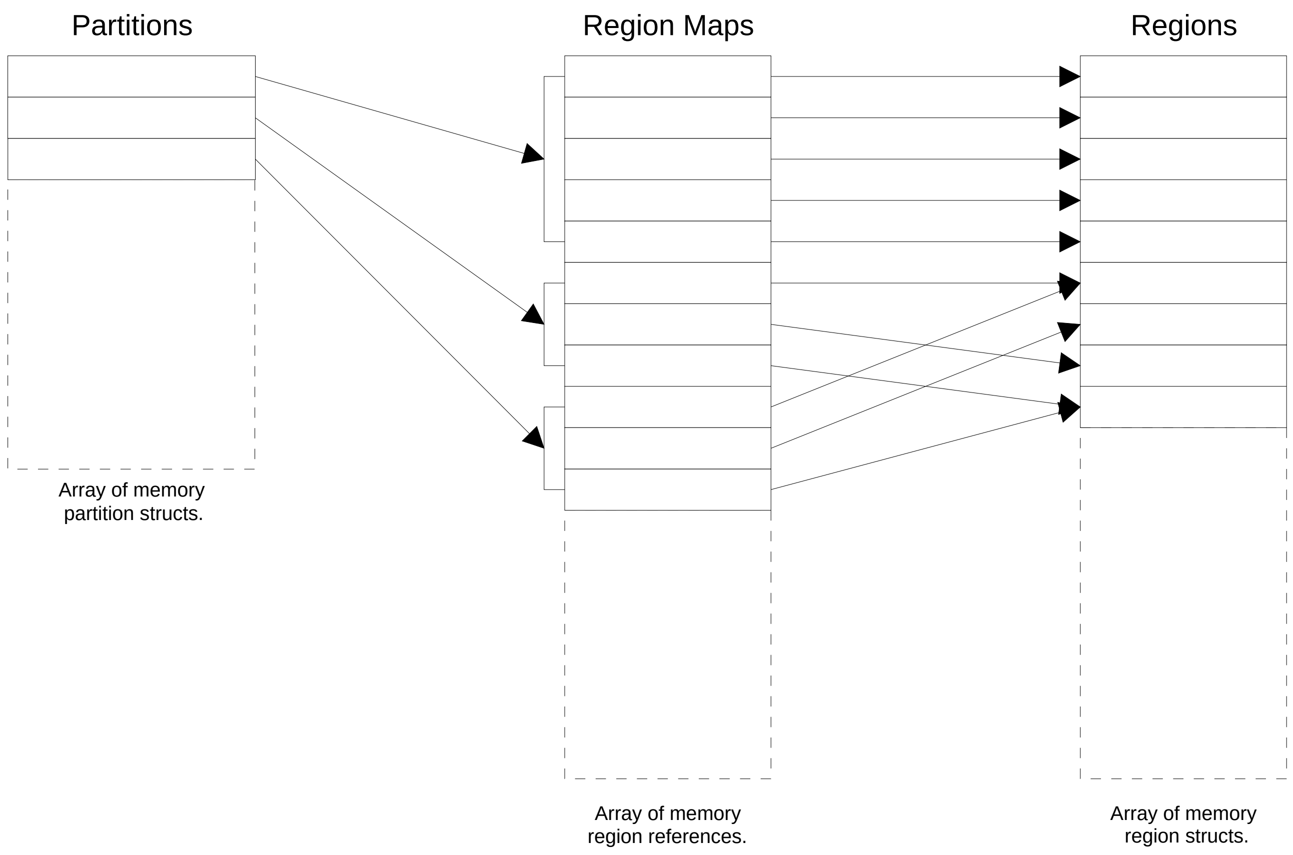

Partitions
Region Maps
Regions
Array of memory partition structs.
Array of memory region references.
Array of memory region structs.

Figure 7.3. Memory partitions and regions

When you use this mechanism, you can assign a memory region to more than one memory partition.

WARNING **Memory partitions must ensure the necessary freedom from interference**
The memory partitioning must correspond with the freedom from interference argumentation of your entire software architecture.

Global memory partition

The first entry of the memory partition table is the *global memory partition*. This partition is enabled at startup and remains enabled indefinitely. Do not assign the global partition to any executable object explicitly.

The global partition should specify the region to allow access to the code and constants as well as the region for read-only access that are used by all executable objects. You can add further memory regions for access by all executable objects, but you must ensure that the safety of the system is not compromised.

The number of regions in the global partition plus the number of regions in the largest non-global partition must not be greater than the number of regions that the memory protection hardware can manage simultaneously.

The microkernel does not adjust the regions for any thread dynamically and if you configure a partition that is too large, the microkernel shuts down when it tries to start the thread.

NOTE **Different handling of the global partition**

Some derivatives of certain architectures (e.g. ARMv8-R-based CPUs) may handle the global partition differently. See the architecture-specific supplement Chapter 9, “Supplementary information for TriCore processors“ for more details.

Memory regions (continued)

You can associate initialization rules with memory regions. The initialization is defined by two addresses, the address of a ROM copy of the initial values and an address within the region which marks the start of the implicitly-initialized variables, i.e. the `.bss` start address. When the microkernel starts, data is copied from the ROM to the region up to the `.bss` start address. The memory from the `.bss` start address to the end of the region is initialized to zero. The following initialization configurations are possible:

Only variables initialized from ROM
: Set the address of the ROM copy to the location where the initialization values are located and the `.bss` start address to zero.

Only variables initialized with zero
: Set the address of the ROM copy to zero and the `.bss` start address to the start address of the memory region.

Some variables initialized from ROM, the rest with zero
: Set the address of the ROM copy to the location where the initialization values are located and the `.bss` start address to the location within the memory region where the implicitly-initialized variables start.

Some variables are uninitialized, the rest is initialized with zero
: Set the address of the ROM copy to zero and the `.bss` start address to the location within the memory region where the implicitly-initialized variables start.

All variables are uninitialized
: Set both the address of the ROM copy and the `.bss` start address to zero.

You should configure your memory regions with symbols and define the symbols in the linker script. If you use the construction macros described in Section 8.4.10, “Configuration of memory regions”, the microkernel uses a naming scheme for the symbols based on the name you assign to the region :

- `MK_RSA_<name>`: start address of the memory region with name `<name>`.
- `MK_RLA_<name>`: end address of the memory region. The end address of a region is the address of the first memory cell above the region that is not part of the region. This is compatible with the behavior of almost all linkers.
- `MK_BSA_<name>`: `.bss` start address of the memory region.

- `MK_RDA_<name>`: start address of the ROM copy for the memory region.

You do not have to add `MK_BSA_<name>` and `MK_RDA_<name>` in the linker script for a specific memory region if you use the macros `MK_MR_STACK()` or `MK_MR_NOINIT()` to configure that memory region.

If your variables get their initial values outside the control of the microkernel, you can define all the memory regions without initialization.

Noteworthy

Memory regions do not have to be disjoint. Overlapping regions, regions within regions and duplicated regions are all possible. You can use this feature to reduce the number of regions in your partitions and so reduce the run-time overhead of switching partitions. You may also need to use it when two executable objects need to have different permissions. You must take care with the initialization in overlapping regions. The best approach is that you ensure that the initial value of each memory cell is defined by at most one region.

You can assign the same memory partition to several executable objects. Normally you only do this when the two executable objects need access to exactly the same memory, including the stack. To reduce the overhead of memory protection, you can create larger partitions that contain all the stacks for related executable objects. But you must be aware that the microkernel may not be able to detect all out-of-bounds errors for these executable objects.

On many microcontrollers you need to define memory regions for the hardware peripherals. You should not initialize these memory regions.

7.4.2.9.2. Multi-core systems

In multi-core systems each core has its own memory partitions, i.e. memory partitions cannot be shared between different cores. In the array of memory partition structs, i.e. `MK_CFG_MEMORYPARTITIONCONFIG`, these memory partitions must be sorted according to the associated core. The core-specific memory partition configuration constants `MK_CFG_Cx_FIRST_MEMORYPARTITION` and `MK_CFG_Cx_NMEMORYPARTITIONS` identify the core-specific memory partitions in `MK_CFG_MEMORYPARTITIONCONFIG`. It is recommended to start with all memory partitions of the first core followed by all memory partitions of the second core and so on for all further cores. Other core sequences are also allowed but are not recommended.

Note that the first entry of the core-specific memory partitions of each core must be the global memory partition of that core.

In contrast to memory partitions, the memory region maps and memory regions can be shared between different cores. The memory region configuration contains the information of which core initializes the memory region if it must be initialized. For more information, see also Section 8.4.10, “Configuration of memory regions”.

7.4.2.9.3. Memory regions when using EB tresos Studio

Table 7.2, “Generated microkernel memory regions” lists the microkernel-specific memory regions which are generated if you use EB tresos Studio to configure the microkernel. Application-specific memory regions are not listed in the table.

NOTE

Architecture-specific microkernel memory regions and memory region content

For architecture-specific microkernel memory regions and memory region content, see also Section 9.4.3, “Memory regions on TriCore processors”.

You can change the memory region configuration for the microkernel manually. For more information on how to do this, see Section 8.4, “Microkernel configuration reference”.

Memory region	Description	Access		
`MK_Rom`	All code and constants which the microkernel must be able to access	r		x
`MK_GlobalRam`	Region to access all RAM on the processor	r		
`MK_Ram`	RAM region for general, i.e. not core-specific microkernel data	r	w	
`MK_Ram_C<x>`	RAM region for microkernel data for core `<x>`.	r	w	
`MK_OsRam`	RAM region for general, i.e. not core-specific QM-OS data	r	w	
`MK_OsRam_C<x>`	RAM region for QM-OS data for core `<x>`.	r	w	
`MK_Io`	Memory mapped peripherals that are accessed by the microkernel	r	w	
`MK_OsIo`	Memory mapped peripherals that are accessed by the QM-OS	r	w	
`MK_c<x>_kernelStack`	Kernel stack for core `<x>`	r	w	
`MK_c<x>_aux1Stack`	Stack for the QM-OS thread on core `<x>`	r	w	
`MK_c<x>_aux2Stack`	Stack for the trusted function thread core `<x>`	r	w	
`MK_c<x>_idleshutdownStack`	Stack for the idle and shutdown-idle thread on core `<x>`	r	w	
`MK_c<x>_errorhookStack`	Stack for the `ErrorHook()` on core `<x>`	r	w	
`MK_c<x>_protectionHookStack`	Stack for the `ProtectionHook()` on core `<x>`	r	w	
`MK_c<x>_threadStack<x>`, `MK_c<x>_threadStack<x>_slot<i>`	Stack for a task or ISR thread on core `<x>`. If there are several such threads assigned to a core, the threads are enumerated in the sequence of their handling in the script and the stack region gets a `_slot<i>` suffix where `<i>` denotes the thread.	r	w	

Table 7.2. Generated microkernel memory regions

The column *Access* indicates the access rights for these regions. *r* stands for read-access, *w* for write access and *x* for execute access.

`MK_Rom` and `MK_GlobalRam` are global regions and part of the global memory partition.

7.4.2.10. Configuring simple schedule tables

The key concepts for simple schedule table configuration are *expiry points* and *actions*. Each SST contains multiple expiry points that are associated with a specific expiry time, represented by a value of the underlying counter. At each expiry point, the microkernel can execute one or more actions, usually the activation of a task thread.

The macros that are used to configure expiry points and actions are described in Section 8.4.13, “Configuration of simple schedule tables”.

7.4.3. Mk_gen_addons.h

`Mk_gen_addons.h` contains the configuration of the configured microkernel add-ons, see also Section 8.4.14, “Configuration of add-ons”.

7.4.4. Mk_gen_global.c

You need to create one source file `Mk_gen_global.c`.

Place the definitions of the stacks that are used in the configuration into this file. Each stack is an array of `mk_stackelement_t` elements. You must place the stacks into separate sections by using `#pragma` directives or `attribute` qualifiers, depending on the compiler that you use.

You should arrange the stacks so that the top-of-stack is correctly aligned according to the application binary interface (ABI) of the processor or compiler that you use. The easiest way to do this is to ensure that the array is aligned by using a command in the linker script. Then the size of the stack should be rounded up to a multiple of the alignment. Then calculate the initial values of the stack pointers so that the alignment is maintained.

7.5. Starting the microkernel

7.5.1. Starting the microkernel in a single-core environment

To start the microkernel in a single-core environment, transfer control to the label `MK_Entry2` at the end of your startup code. The microkernel then performs its own initialization and executes your configured *init function* (typically `main()`) when the initialization is complete. The microkernel's startup code is designed so that it is nearly impossible to start the microkernel by error without a prior jump to `MK_Entry2()`.

Your *init function* runs in a thread. What happens afterwards depends on the behavior of your *init function*. If you do nothing, there is only an idle thread in your system and nothing happens. In a typical AUTOSAR system, `StartOS()` is called either directly or indirectly to initialize the counters, alarms etc. and to start all the objects that are configured to start automatically in the *application mode* that you pass as parameter.

When control is transferred to `MK_Entry2`, the processor should be in the state defined in Section 9.1, "Startup conditions for TriCore processors". For more information, see also the EB tresos Safety OS safety manual for TriCore family [SAFETYMANTRICORE].

When a microkernel-based system starts immediately after a reset, the reset vector can transfer control to the label `MK_Entry`. The routine at this location performs a low-level initialization of the processor and places it in the state expected by `MK_Entry2`. Then it transfers control to `MK_Entry2`.

However, in most systems boot management software must run after the low-level initialization. It is expected that the boot management software performs a low-level initialization in order to run correctly. It is therefore recommended that the boot manager should transfer control to `MK_Entry2`.

Figure 7.4, "Startup procedure of EB tresos Safety OS" depicts a typical startup scenario in a single-core environment: The boot management software *boot loader* starts the microkernel. The microkernel itself calls the function `MkInitHardwareBeforeData()`, then initializes the data sections and calls the function `MK_InitHardwareAfterData()`. For a description of these functions, see Section 8.5.2, "Mk_board.c".

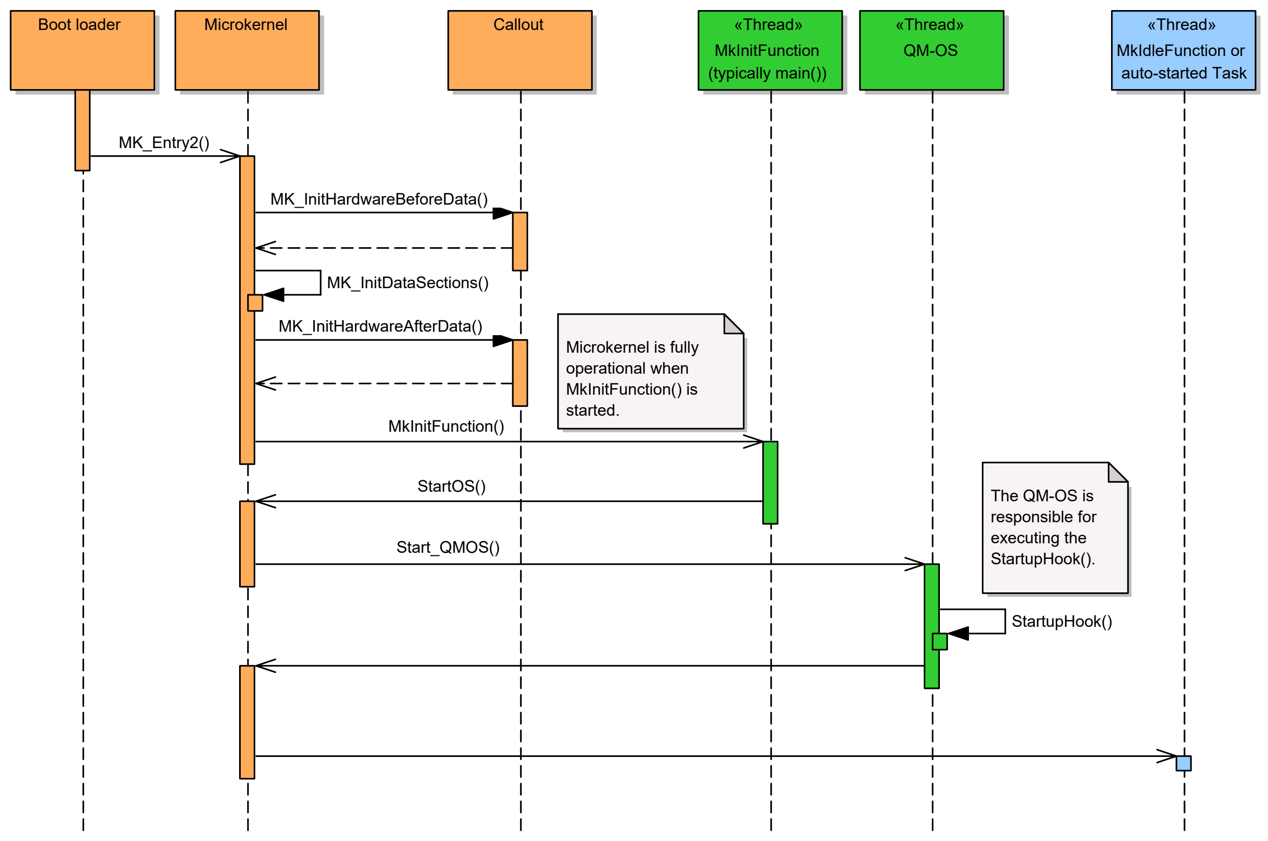

Boot loader
Microkernel
Callout
«Thread» MkInitFunction (typically main())
«Thread» QM-OS
«Thread» MkIdleFunction or auto-started Task
MK_Entry2()
MK_InitHardwareBeforeData()
MK_InitDataSections()
MK_InitHardwareAfterData()
Microkernel is fully operational when MkInitFunction() is started.
MkInitFunction()
StartOS()
The QM-OS is responsible for executing the StartupHook().
Start_QMOS()
StartupHook()

Figure 7.4. Startup procedure of EB tresos Safety OS

7.5.2. Starting the microkernel in a multi-core environment

The microkernel startup in a multi-core environment is basically the same as in a single-core environment so make sure to read Section 7.5.1, “Starting the microkernel in a single-core environment” first.

The main difference is that you have to start the microkernel on all cores that are configured for the microkernel. Therefore you must transfer the control to the label `MK_Entry2` at the end of the startup code of each of those cores.

Before you do that you must initialize the microkernel synchronization data. Call `MK_InitSyncHere()` to achieve this.

In contrast to a single-core environment the microkernel only calls the *init function* automatically for the master core, i.e. the core defined by the constant `MK_CFG_HWMASTERCOREINDEX`. On the other cores the microkernel waits for the core to be started via `StartCore()` before it calls their *init function*. Therefore you must call `StartCore()` for all waiting cores.

In addition to that, `StartOS()` synchronizes with all other cores which are configured for the microkernel. Therefore make sure to start all cores and then call `StartOS()` at the end of the *init function* of each core.

A typical multi-core microkernel startup works as follows:

1. One core calls `MK_InitSyncHere()` during its startup code.
2. All cores configured for the microkernel synchronize themselves with each other before the next step.
3. All cores configured for the microkernel start `MK_Entry2` at the end of their startup code.
4. The core `MK_CFG_HWMASTERCOREINDEX` starts all other cores in its *init function* by calling `StartCore()` for all those cores.
5. All cores call `StartOS()` at the end of their *init function*.

7.6. Shutdown sequence

The shutdown sequence of the microkernel is depicted in Figure 7.5, “Shutdown procedure of the microkernel”: If the shutdown sequence is triggered, for example by a call to `ShutdownOS()`, the microkernel starts the `ShutdownHook()` and executes the shutdown-idle thread after its termination. The microkernel is still fully operational after the complete shutdown sequence. However, the shutdown-idle thread does not allow any other activity, since it is executed with interrupts disabled and never transfers control back to the microkernel.

NOTE

Activation of other threads

You can activate other threads through the actions of `ShutdownHook()` and these threads run in the correct priority order. The shutdown-idle thread runs on completion of the last of these threads.

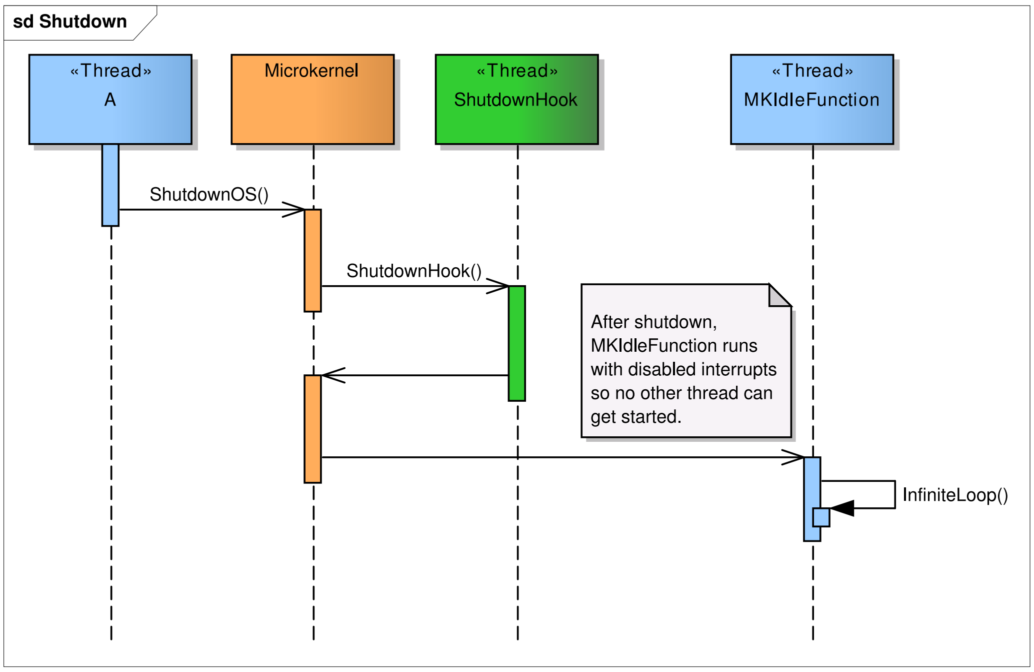


Figure 7.5. Shutdown procedure of the microkernel

In a multi-core configuration, `ShutdownOS()` shuts down the core on which it is called. To shut down the entire system, use `ShutdownAllCores` instead.

8. Microkernel reference manual

8.1. Microkernel limits

The following limits exist for the number of microkernel objects:

Object	Max. number
Tasks	255
ISRs	255
Resources	252
Events	32 per task

Table 8.1. Limits on number of microkernel objects.

NOTE

Environmental limits

The actual number of microkernel objects you can use in a particular microkernel system depends on environmental conditions like available memory, toolchain limitations etc. It can thus be significantly lower than the maximum numbers given in Table 8.1, “Limits on number of microkernel objects.”.

8.2. Microkernel API reference

There are four classes of AUTOSAR services provided by EB tresos Safety OS:

MK
: Services provided by the microkernel.

  These services are native microkernel services. You can use them in each partition without taking special care on the assigned ASIL level.

MK/OS
: Services provided by a microkernel interface to the QM-OS, see also Section 6.4.1, “Services with integrity category 2”.

  The microkernel ensures freedom from interference for these services from other threads but the services themselves are not reliable.

OS
: Services provided directly by the QM-OS, see also Section 6.4.2, “Services with integrity category 3”.

  The microkernel does not ensure anything for these services. Take appropriate measures if they are used in an ASIL partition.

UNIMP
: Unimplemented services.

The following tables provide a list of all the AUTOSAR and EB-vendor-specific services and indicate into which of these classes each service falls. The remainder of this section provides descriptions of the services provided by the microkernel and of the interfaces to the QM-OS that are provided by the microkernel.

Service name	**Class**
ActivateTask	MK
AllowAccess	MK
CallTrustedFunction	MK
CancelAlarm	MK/OS
ChainTask	MK
CheckIsrMemoryAccess	UNIMP
CheckObjectAccess	UNIMP
CheckObjectOwnership	UNIMP
CheckTaskMemoryAccess	UNIMP
ClearEvent	MK
DisableAllInterrupts	MK
EnableAllInterrupts	MK

Service name	Class
GetActiveApplicationMode	OS
GetAlarm	MK/OS
GetAlarmBase	OS
GetApplicationID	MK
GetApplicationState	MK
GetCoreID	MK
GetCounterValue	MK/OS
GetCurrentApplicationID	MK
GetElapsedValue	OS
GetEvent	MK
GetISRID	MK
GetNumberOfActivatedCores	OS
GetResource	MK
GetScheduleTableStatus	OS
GetSpinlock	MK
GetTaskID	MK
GetTaskState	MK
IncrementCounter	MK/OS
NextScheduleTable	MK/OS
OSErrorGetServiceId	MK
OSError_<svc>_<param>	MK
ReleaseResource	MK
ReleaseSpinlock	MK
ResumeAllInterrupts	MK
ResumeOSInterrupts	MK
Schedule	MK
SetAbsAlarm	MK/OS
SetEvent	MK
SetRelAlarm	MK/OS
SetScheduleTableAsync	MK/OS
ShutdownAllCores	MK

Service name	Class
ShutdownOS	MK/OS
StartCore	MK
StartNonAutosarCore	UNIMP
StartOS	MK/OS
StartScheduleTableAbs	MK/OS
StartScheduleTableRel	MK/OS
StartScheduleTableSynchron	MK/OS
StopScheduleTable	MK/OS
SuspendAllInterrupts	MK
SuspendOSInterrupts	MK
SyncScheduleTable	MK/OS
TerminateApplication	MK/OS
TerminateTask	MK
TryToGetSpinlock	MK
WaitEvent	MK

Table 8.2. Standard AUTOSAR services in EB tresos Safety OS

Service Name	Class
MK_AcquireLock	MK
MK_AsyncActivateTask	MK
MK_AsyncCancelAlarm	MK/OS
MK_AsyncIncrementCounter	MK/OS
MK_AsyncNextScheduleTable	MK/OS
MK_AsyncSetAbsAlarm	MK/OS
MK_AsyncSetEvent	MK
MK_AsyncSetRelAlarm	MK/OS
MK_AsyncSetScheduleTableAsync	MK/OS
MK_AsyncStartScheduleTableAbs	MK/OS
MK_AsyncStartScheduleTableRel	MK/OS
MK_AsyncStartScheduleTableSynchron	MK/OS
MK_AsyncStopScheduleTable	MK/OS
MK_AsyncSyncScheduleTable	MK/OS

Service Name	Class
MK_ConditionalGetResource	MK
MK_DiffTime	MK
MK_DiffTime32	MK
MK_DisableInterruptSource	MK
MK_ElapsedTime	MK
MK_ElapsedTime32	MK
MK_EnableInterruptSource	MK
MK_ErrorGetParameter	MK
MK_ErrorGetParameterForCore	MK
MK_ErrorGetServiceId	MK
MK_ErrorGetServiceIdForCore	MK
MK_GetClearEvent	MK
MK_GetErrorInfo	MK
MK_GetErrorInfoForCore	MK
MK_GetExceptionInfo	MK
MK_GetExceptionInfoForCore	MK
MK_GetPanicExceptionInfo	MK
MK_GetPanicExceptionInfoForCore	MK
MK_GetPanicReason	MK
MK_GetPanicReasonForCore	MK
MK_GetProtectionInfo	MK
MK_GetProtectionInfoForCore	MK
MK_InitSyncHere	MK
MK_IsScheduleNecessary	MK
MK_ReadTime	MK
MK_ReleaseLock	MK
MK_ReportError	MK
MK_ScheduleIfNecessary	MK
MK_WaitGetClearEvent	MK
MK_SuspendInterrupts	MK
MK_ResumeInterrupts	MK

Service Name	Class
MK_SstStartCounter	MK
MK_SstAdvanceCounter	MK
MK_SstStopCounter	MK
OS_AtomicClearFlag	OS
OS_AtomicCompareExchange	OS
OS_AtomicExchange	OS
OS_AtomicFetchAdd	OS
OS_AtomicFetchAnd	OS
OS_AtomicFetchOr	OS
OS_AtomicFetchSub	OS
OS_AtomicFetchXor	OS
OS_AtomicInit	OS
OS_AtomicLoad	OS
OS_ATOMIC_OBJECT_INITIALIZER	OS
OS_AtomicStore	OS
OS_AtomicTestAndSetFlag	OS
OS_AtomicThreadFence	OS

Table 8.3. EB-vendor-specific services in EB tresos Safety OS.

Name

ActivateTask — activates a task

Synopsis

```
#include <MicroOs.h>

StatusType ActivateTask(TaskType taskId);

StatusType MK_AsyncActivateTask(TaskType taskId);
```

Description

`ActivateTask()` activates the task specified by the `taskId` parameter. If the task's thread is idle, `ActivateTask()` enqueues the thread for execution on the core to which it is allocated, behind all threads of higher or equal priority but ahead of all threads of lower priority.

If the task is allocated to the same core as the caller and has a higher priority than the caller's current priority, it executes immediately. Control does not return to the calling thread until after the activated task terminates or successfully waits for an event.

If the task is allocated to the same core and has a priority lower than or the same as the current priority of the calling thread, it remains queued for execution and runs when it arrives at the head of the queue. `ActivateTask()` returns to the caller immediately after the task is successfully enqueued.

If the task is allocated to a different core than the caller, `ActivateTask()` uses a synchronous inter-core activation request to activate the task on that core.

`MK_AsyncActivateTask()` behaves like `ActivateTask()` with the exception that it uses an asynchronous inter-core activation request if the task is allocated to another core. This means that `MK_AsyncActivateTask()` does not wait for the result of the operation on the other core and therefore cannot return the error codes `E_OS_LIMIT` or `E_OS_STATE`. Instead, the return value is `E_OK`, which indicates that the activation request was sent.

On single-core processors `MK_AsyncActivateTask()` is an alias for `ActivateTask()`.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_ActivateTask` or `OSServiceId_ActivateTask`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_InvalidTaskId	ASIL	E_OS_ID	The `taskId` parameter does not identify a configured task.
MK_eid_MaxActivationsExceeded	ASIL	E_OS_LIMIT	The task already reached its activation limit.
MK_eid_Quarantined	ASIL	E_OS_STATE	The OS-Application to which this task belongs is terminated.

Name

AllowAccess — sets the OS-Application state to `APPLICATION_ACCESSIBLE`

Synopsis

#include <MicroOs.h>

```
StatusType AllowAccess(void);
```

Description

`AllowAccess()` sets the state of the OS-Application of the caller to `APPLICATION_ACCESSIBLE` if that state is `APPLICATION_RESTARTING`.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_AllowAccess` or `OSServiceId_AllowAccess`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_AppNotRestarting	ASIL	E_OS_STATE	The OS-Application of the caller is in the wrong state.
MK_eid_CallFromNonApplication	ASIL	E_OS_CALLEV-EL	The caller does not belong to an OS-Application.

Name

CallTrustedFunction — starts trusted function in a separate thread

Synopsis

#include <MicroOs.h>

```
StatusType CallTrustedFunction(TrustedFunctionIndexType Func-
tionIndex, TrustedFunctionParameterRefType FunctionParams);
```

Description

`CallTrustedFunction()` activates the trusted function specified by the `FunctionIndex` parameter in a separate thread. Both of the parameters are passed to the trusted function.

You must allocate the trusted function to the core from which it is called.

For more information, see Section 6.3, “Trusted functions”.

Return value

A return value of `E_OK` indicates a successful completion of the call but gives no information about the success or failure of the trusted function itself. Any other return value indicates an error code as described below.

Service identification

`MK_sid_CallTrustedFunction` or the function's ID.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_TFCallFromWrongContext	ASIL	E_OS_CALLEV-EL	The service is called from an incorrect context.
MK_eid_InvalidTrustedFunctionId	ASIL	E_OS_-SERVICEID	Invalid function identifier. The designated function does not exist.

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_TFCallFromWrongCore	ASIL	E_OS_CALLEV-EL	The designated function is configured on a different core.

Name

ChainTask — activates a task and terminates the current thread

Synopsis

#include <MicroOs.h>

```
StatusType ChainTask(TaskType taskId);
```

Description

`ChainTask()` first terminates the calling thread unconditionally. Then it activates the task specified by the `taskId` parameter and enqueues its thread for execution. The thread remains queued for execution and runs when it arrives at the head of the queue, which may happen immediately.

If the task specified by `taskId` is allocated to a different core than the caller, `Chaintask()` uses an asynchronous inter-core chain task request to chain the task on the other core.

In each case control is never returned to the caller.

Return value

None. `ChainTask()` never returns to the caller. Possible errors are reported via `ErrorHook()` if enabled. The return type is a dummy to satisfy AUTOSAR interface specifications.

Service identification

`MK_sid_ChainTask` or `OSServiceId_ChainTask`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_InvalidTaskId	ASIL	E_OS_ID	The `taskId` parameter does not identify a configured task.
MK_eid_MaxActivationsExceeded	ASIL	E_OS_LIMIT	The task has already reached its activation limit.

ErrorId	Mode	Error code	Description
MK_eid_Quarantined	ASIL	E_OS_STATE	The OS-Application to which this task belongs is terminated.

Name

ClearEvent — clears pending events

Synopsis

```
#include <MicroOs.h>

StatusType ClearEvent(EventMaskType Mask);
```

Description

`ClearEvent()` clears the events that are specified by the `Mask` parameter from the caller's set of pending events. The caller must be an EXTENDED task.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_ClearEvent` or `OSServiceId_ClearEvent`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_ThreadIsNotATask	ASIL	E_OS_CALLEV-EL	The calling thread is not a task.
MK_eid_TaskIsNotExtended	ASIL	E_OS_ACCESS	The calling thread is a BASIC task.

Name

GetApplicationID — gets the ID of the current OS-Application

Synopsis

```
#include <MicroOs.h>

ApplicationType GetApplicationID(void);
```

Description

`GetApplicationID()` returns the ID of the OS-Application that is currently executed on the calling core. If no application is running, the return value is `INVALID_APPLICATION`.

Return value

`GetApplicationID()` returns an application ID, or `INVALID_APPLICATION` if no OS-Application is active. The values and range are equal to the OS-Application symbolic names configured in `Mk_gen_user.h`.

Service identification

`MK_sid_GetApplicationId` or `OSServiceId_GetApplicationID`.

Error handling

No errors are reported.

Name

GetApplicationState — gets the state of the given OS-Application

Synopsis

#include <MicroOs.h>

```
StatusType GetApplicationState(ApplicationType
    app, ApplicationStateRefType pState);
```

Description

`GetApplicationState()` gets the current state of the given OS-Application `app` and writes it to the memory location given by `pState`.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_GetApplicationState` or `OSServiceId_GetApplicationState`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_InvalidOsApplication	ASIL	E_OS_ID	The given OS-Application does not exist.

Name

GetCoreID — gets the core ID

Synopsis

#include <MicroOs.h>

```
CoreIdType GetCoreID(void);
```

Description

`GetCoreID()` returns the ID of the processor core (= logical core index) on which the call was made.

Return value

`GetCoreID()` returns the processor core ID.

Service identification

`MK_sid_GetCoreID` or `OSServiceId_GetCoreID`.

Error handling

No errors are reported.

Name

GetCurrentApplicationID — gets the ID of the current OS-Application

Synopsis

```
#include <MicroOs.h>

ApplicationType GetCurrentApplicationID(void);
```

Description

`GetCurrentApplicationID()` returns the ID of the OS-Application which is executed on the core of the caller. If no application is running, the return value is `INVALID_OSAPPLICATION`.

Note that if the caller is not within a `CallTrustedFunction()` call, the value is equal to the result of `GetApplicationID()`.

Return value

`GetCurrentApplicationID()` returns the ID of the OS-Application, or `INVALID_OSAPPLICATION` if no OS-Application is active. The values and range are equal to the OS-Application symbolic names configured in `Mk_gen_user.h`.

Service identification

`MK_sid_GetCurrentApplicationID` or `OSServiceId_GetCurrentApplicationID`.

Error handling

No errors are reported.

Name

GetEvent — gets a task's pending events

Synopsis

#include <MicroOs.h>

```
StatusType GetEvent(TaskType taskId, EventMaskRefType eventRef);
```

Description

`GetEvent()` places the events that are pending for the task specified by the `taskId` parameter into the event mask variable. The `eventRef` parameter references the event mask variable.

`GetEvent()` is a library function that runs with the permissions of the calling thread. No system call is made. This means that any access violations, e.g. caused by an incorrect `eventRef`, are associated with the calling thread.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_GetEvent` or `OSServiceId_GetEvent`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_InvalidTaskId	ASIL	E_OS_ID	The `taskId` parameter does not identify a configured task.
MK_eid_TaskIsNotExtended	ASIL	E_OS_ACCESS	The task identified by the `taskId` parameter is a BASIC task.

Name

GetISRID — gets the current ISR's ID

Synopsis

```
#include <MicroOs.h>

ISRType GetISRID(void);
```

Description

`GetISRID()` returns the ID of the current ISR on the caller's core. The current ISR is defined as the ISR object associated with the highest priority ISR thread in the thread queue. If there is no ISR thread in the queue, `INVALID_ISR` is returned.

Return value

`GetISRID()` returns an ISR ID, or `INVALID_ISR` if no ISR is active. The values and range are equal to the ISR symbolic names configured in `Mk_gen_user.h`.

Service identification

`MK_sid_GetIsrId` or `OSServiceId_GetISRID`.

Error handling

No errors are reported.

Name

GetResource, SuspendOSInterrupts, SuspendAllInterrupts, DisableAllInterrupts, GetSpinlock, TryToGetSpinlock, MK_AcquireLock — acquire a lock and raise the calling thread's priority

Synopsis

#include <MicroOs.h>

```
StatusType GetResource(ResourceType resourceId);

void SuspendOSInterrupts(void);

void SuspendAllInterrupts(void);

void DisableAllInterrupts(void);

StatusType GetSpinlock(SpinlockIdType SpinlockId);

StatusType TryToGetSpinlock(SpinlockIdType
SpinlockId, TryToGetSpinlockType Success);

mk_parametertype_t MK_AcquireLock(mk_lockid_t lockId);
```

Description

`MK_AcquireLock()` acquires the lock object specified by the `lockId` parameter. If the lock object is already occupied by the calling thread, the microkernel increases the nesting count of the lock by one. However, it only increases the nesting count if the lock's configured nesting limit is not exceeded. If the calling thread's current priority is lower than the lock's ceiling priority, the microkernel raises the calling thread's priority to the lock's ceiling priority. Also the microkernel sets the calling thread's interrupt lock level to the lock's interrupt lock level.

If the `lockId` parameter is the ID of a spinlock or an EB-vendor-specific combined lock object, the spinlock component is acquired if possible, but the return value is `MK_E_TRYAGAIN` if the spinlock component of the lock cannot be acquired. `MK_E_TRYAGAIN` is not an error; the `ErrorHook()` is not called. The caller should repeat the call until the return value is `E_OK`. In case of a combined lock, no part of the lock is taken, if the return value is `MK_E_TRYAGAIN`.

`MK_AcquireLock()` is normally not called directly. Instead, use one of the AUTOSAR-compatible APIs described below.

`GetResource()` is a macro that provides an AUTOSAR-compatible API by calling `MK_AcquireLock()` and converting the return value to `StatusType`.

`SuspendOSInterrupts()` is a macro that provides an AUTOSAR-compatible API. The macro expands to two different functions depending on the `MK_USE_INT_LOCKING_CALLOUTS` preprocessor switch. If the macro

MK_USE_INT_LOCKING_CALLOUTS is defined and is non-zero, SuspendOSInterrupts() expands to MK_SuspendCallout(MK_resLockCat2). Otherwise it expands to MK_SuspendInterrupts(MK_resLockCat2). MK_SuspendCallout() can be used to implement the interrupt locking hook.

SuspendAllInterrupts() and DisableAllInterrupts() are both macros that provide an AUTOSAR-compatible API. The macros expand to two different functions depending on the MK_USE_INT_LOCKING_CALLOUTS preprocessor switch. If the macro MK_USE_INT_LOCKING_CALLOUTS is defined and is non-zero, SuspendAllInterrupts() and DisableAllInterrupts() expand to MK_SuspendCallout(MK_resLockCat1). Otherwise they expand to MK_SuspendInterrupts(MK_resLockCat1). MK_SuspendCallout() can be used to implement the interrupt locking hook.

GetSpinlock() is a library function that provides an AUTOSAR-compatible API by repeatedly calling MK_AcquireLock() as long as the return value is MK_E_TRYAGAIN.

TryToGetSpinlock() is a library function that provides an AUTOSAR-compatible API by calling MK_AcquireLock() and converting the return value into a status code and a success/failure indicator. The function executes in the context of the caller so that writing to the Success parameter occurs with the memory protection of the caller in place. No other validation of the Success parameter is performed.

If the value returned by MK_AcquireLock() is E_OK, TryToGetSpinlock() sets the variable referenced by the Success value to TRYTOGETSPINLOCK_SUCCESS and returns E_OK.

If the value returned by MK_AcquireLock() is MK_E_TRYAGAIN, TryToGetSpinlock() sets the variable referenced by the Success value to TRYTOGETSPINLOCK_NOSUCCESS and returns E_OK.

If the value returned by MK_AcquireLock() is any other value, TryToGetSpinlock() returns that value. In this case the variable referenced by the Success parameter remains unchanged.

NOTE

Nesting limits

For AUTOSAR-conformant resources and spinlocks the nesting limit is 1. The interrupt locks MK_RESCAT1 and MK_RESCAT2 have a higher nesting limit.

NOTE

GetResource() and spinlocks

The common API underlying GetResource(), GetSpinlock() and TryToGetSpinlock() means that you can use any of the AUTOSAR APIs regardless of whether the specified lock is a resource, a spinlock, or an EB-specific combined lock. If you use GetResource() to acquire a spinlock or EB-specific combined lock, you must ensure that a return value of MK_E_TRYAGAIN is correctly handled by your software.

Caveat

The services GetSpinlock() and TryToGetSpinlock() are only available on multi-core processors.

Return value

A return value of `E_OK` indicates a successful completion of the function.

A return value of `MK_E_TRYAGAIN` indicates that the attempt to acquire a lock with a spinlock component did not succeed. Your software must not enter the critical section that is protected by this lock. However, if the AUTOSAR-compatible API is used correctly, this return value is not visible to the caller.

Any other return value indicates an error code as described below.

Service identification

`MK_sid_AcquireLock` or `OSServiceId_GetResource`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_InvalidLockId	ASIL	E_OS_ID	The `lockId` parameter does not identify a configured lock.
MK_eid_LockConfiguredForDifferentCore	ASIL	E_OS_ID	The `lockId` parameter identifies a lock that is not configured for the core on which the caller executes.
MK_eid_LockAlreadyOccupied	ASIL	E_OS_ACCESS	The lock is already occupied by another thread.
MK_eid_LockNestingLimitExceeded	ASIL	E_OS_ACCESS	The lock is already occupied by the calling thread.

Name

GetTaskID — gets the ID of the current task

Synopsis

```
#include <MicroOs.h>

StatusType GetTaskID(TaskRefType TaskId);
```

Description

`GetTaskID()` writes the ID of the current task to the variable referred by `TaskId` if there is currently a task running. If no task is running, it writes the value `INVALID_TASK` to that variable. In both cases it returns `E_OK`.

`GetTaskID()` runs with the permissions of the calling thread. This means that any access violations caused by an incorrect `TaskId` are associated with the calling thread. The function does not check the validity of the reference in software, but relies on the memory protection hardware to do the job without a CPU time overhead.

Return value

`E_OK` always.

Service identification

`MK_sid_GetTaskId` or `OSServiceId_GetTaskID`.

Error handling

No errors are reported.

Name

GetTaskState — gets the state of the given task

Synopsis

#include <MicroOs.h>

```
StatusType GetTaskState(TaskType taskId, TaskStateRefType taskStateRef);
```

Description

`GetTaskState()` tries to determine the task state of the task with the given `taskId` and to write that state to the variable referred by `taskStateRef`.

If the caller of this function is the core to which the given task is allocated, it determines the state, i.e. SUSPENDED, READY, RUNNING, or WAITING directly, writes it to the output variable and returns `E_OK`.

If the task is allocated to a different core than the caller, `GetTaskState()` uses a synchronous inter-core get task state request to get the task state.

If the `taskId` parameter is out of range, `GetTaskState()` returns `E_OS_ID` and does not change the variable referred by `taskStateRef`.

`GetTaskState()` runs with the permissions of the calling thread. This means that any access violations caused by an incorrect `taskStateRef` are associated with the calling thread.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_GetTaskState` or `OSServiceId_GetTaskState`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_InvalidTaskId	ASIL	E_OS_ID	The `taskId` parameter does not identify a configured task.

Name

ReleaseResource, ResumeOSInterrupts, ResumeAllInterrupts, EnableAllInterrupts, ReleaseSpinlock, MK_-ReleaseLock — release a lock and lower the calling thread's priority

Synopsis

```
#include <MicroOs.h>

StatusType ReleaseResource(ResourceType resourceId);

void ResumeOSInterrupts(void);

void ResumeAllInterrupts(void);

void EnableAllInterrupts(void);

StatusType ReleaseSpinlock(SpinlockIdType spinlockId);

mk_parametertype_t MK_ReleaseLock(mk_lockid_t lockId);
```

Description

`MK_ReleaseLock()` releases the lock object specified by the `lockId` parameter provided that the lock object is occupied by the calling thread. The nesting count of the lock is reduced by one. If the nesting count reaches zero, the lock is released, provided it is the thread's most recently acquired lock.

When `MK_ReleaseLock()` releases the lock, the thread's priority is set to the priority that is saved by the lock when the thread first acquired it. The thread's interrupt lock level is similarly restored.

If the lock object is the ID of a spinlock or an EB-vendor-specific combined lock object, the spinlock component is released.

`MK_ReleaseLock()` is normally not called directly. Instead, use one of the AUTOSAR-compatible APIs described below.

`ReleaseResource()` is a macro that provides an AUTOSAR-compatible API by calling `MK_ReleaseLock()` and converting the return value to `StatusType`.

`ResumeOSInterrupts()` is a macro that provides an AUTOSAR-compatible API. The macro expands to two different functions depending on the `MK_USE_INT_LOCKING_CALLOUTS` preprocessor switch. If the macro `MK_USE_INT_LOCKING_CALLOUTS` is defined and is non-zero, `ResumeOSInterrupts()` expands to `MK_ResumeCallout(MK_resLockCat2)`. Otherwise it expands to `MK_ResumeInterrupts(MK_resLockCat2)`. `MK_ResumeCallout()` can be used to implement the interrupt locking hook.

`ResumeAllInterrupts()` and `EnableAllInterrupts()` are both macros that provide an AUTOSAR-compatible API. The macros expand to two different functions depending on the `MK_USE_INT_LOCKING_CALLOUTS` preprocessor switch. If the macro `MK_USE_INT_LOCKING_CALLOUTS` is defined and is non-zero, `ResumeAllInterrupts()` and `EnableAllInterrupts()` expand to `MK_ResumeCallout(MK_resLockCat1)`. Otherwise they expand to `MK_ResumeInterrupts(MK_resLockCat1)`. `MK_ResumeCallout()` can be used to implement the interrupt locking hook.

`ReleaseSpinlock()` is a macro that provides an AUTOSAR-compatible API by calling `MK_ReleaseLock()` and converting the return value to `StatusType`.

NOTE

Nesting limits
For AUTOSAR-conformant resources and spinlocks the nesting limit is 1. The interrupt locks `MK_RESCAT1` and `MK_RESCAT2` have a higher nesting limit.

NOTE

`ReleaseResource()` and spinlocks
The common API underlying `ReleaseResource()` and `ReleaseSpinlock()` means that you can use either of the AUTOSAR APIs regardless of whether the specified lock is a resource, a spinlock, or an EB combined lock.

Caveat

The service `ReleaseSpinlock()` is only available on multi-core processors.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_ReleaseLock` or `OSServiceId_ReleaseResource`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_InvalidLockId	ASIL	E_OS_ID	The `lockId` parameter does not identify a configured lock.

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_LockConfiguredForDiffer-entCore	ASIL	E_OS_ID	The `lockId` parameter identifies a lock that is not configured for the core on which the caller executes.
MK_eid_LockNotOccupied	ASIL	E_OS_NOFUNC	The lock is not acquired.
MK_eid_LockNotOccupiedByCaller	ASIL	E_OS_NOFUNC	The lock is not acquired by the calling thread.
MK_eid_LockReleaseSequenceEr-ror	ASIL	E_OS_NOFUNC	Another lock is acquired more recently by the calling thread.

Name

Schedule — yields the processor to higher-priority threads

Synopsis

```
#include <MicroOs.h>

StatusType Schedule(void);
```

Description

`Schedule()` reduces the priority of the calling thread from its current value to the thread's configured value. Thus it temporarily renounces the thread's *non-preemptive* status and releases all internal resources that are allocated to the thread. This permits threads with a higher configured priority to run.

If the thread occupies one or more locks, i. e. resources, interrupt locks or combined locks, the resulting priority is not lower than the ceiling priority of the highest-priority lock that is occupied.

When the thread regains the CPU, its priority gets restored to its previous value.

Return value

A return value of `E_OK` indicates a successful completion of the function.

Service identification

`MK_sid_Schedule` or `OSServiceId_Schedule`.

Error handling

No errors are reported.

Name

SetEvent, MK_AsyncSetEvent — send an event to a task

Synopsis

#include <MicroOs.h>

```
StatusType SetEvent(TaskType taskId, EventMaskType mask);

StatusType MK_AsyncSetEvent(TaskType taskId, EventMaskType mask);
```

Description

`SetEvent()` sends the events specified by the `mask` parameter to the task specified by the `taskId` parameter. The task to which the events are sent must be an EXTENDED task and must be in the state WAITING, READY or RUNNING.

If the task is in the WAITING state, it gets reawakened and becomes eligible to use the CPU.

If the task `taskId` is allocated to a different core than the caller, `SetEvent()` uses a synchronous inter-core set event request to set the event on that core.

`MK_AsyncSetEvent()` behaves like `SetEvent()` with the exception that it uses an asynchronous inter-core set event request if the task is allocated to another core. This means that `MK_AsyncSetEvent()` does not wait for the result of the operation and therefore cannot return the error code `E_OS_STATE`.

See also WaitEvent and MK_WaitGetClearEvent.

On single-core processors `MK_AsyncSetEvent()` is an alias for `SetEvent()`.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_SetEvent` or `OSServiceId_SetEvent`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_InvalidTaskId	ASIL	E_OS_ID	The `taskId` parameter does not identify a configured task.
MK_eid_TaskIsNotExtended	ASIL	E_OS_ACCESS	The task identified by the `taskId` parameter is a BASIC task.
MK_eid_Quarantined	ASIL	E_OS_STATE	The caller does not have access to the OS-Application of the given task.

Name

ShutdownAllCores — shuts down all cores controlled by the microkernel

Synopsis

```
#include <MicroOs.h>

void ShutdownAllCores(StatusType error);
```

Description

`ShutdownAllCores()` sends a core shutdown request with the provided `error` to each other core which is controlled by the microkernel. `ShutdownAllCores()` then shuts down the core, on which it is executed. It does not wait for replies for the sent shutdown requests.

If a `ShutdownHook` is configured, `ShutdownAllCores()` provides the `error` as parameter to the `ShutdownHook`.

Caveat

On single-core processors ShutdownAllCores is an alias for ShutdownOS.

Return value

None.

Service identification

`MK_sid_ShutdownAllCores` or `OSServiceId_ShutdownAllCores`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_WithoutPermission	ASIL	E_OS_ACCESS	The calling application does not have the appropriate permission.

Name

ShutdownOS — shuts down the operating system on the current core

Synopsis

#include <MicroOs.h>

```
StatusType ShutdownOS(StatusType error);
```

Description

`ShutdownOS()` shuts down the system on the calling core. It terminates all threads on that core, then it activates a new idle thread. By default this thread executes an endless loop with interrupts disabled.

If the `ShutdownHook()` feature is configured, another thread is activated to execute the user-supplied function `ShutdownHook()`. The `error` given in the call to `ShutdownOS()` is passed to the `ShutdownHook()`.

The priorities of the new threads ensure that the shutdown-idle loop executes when `ShutdownHook()` returns or otherwise terminates.

`ShutdownOS()` does not evaluate its parameter in any way, so you can pass any value that is understood by the `ShutdownHook()`, or simply zero if you did not configure a `ShutdownHook()`.

NOTE

Activation of other threads

You can activate other threads through the actions of `ShutdownHook()` and these threads run in the correct priority order. The shutdown-idle thread runs on completion of the last of these threads.

See also Section 7.6, “Shutdown sequence”.

Return value

None. `ShutdownOS()` never returns if the caller has the permission to call this API.

Service identification

`MK_sid_Shutdown` or `OSServiceId_ShutdownOS`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_WithoutPermission	ASIL	E_OS_ACCESS	The calling application does not have the appropriate permission.

Name

StartCore — starts the given core

Synopsis

```
#include <MicroOs.h>

void StartCore(CoreIdType coreID, StatusType * status);
```

Description

`StartCore()` starts the provided core via a synchronous inter-core core activation request, see Section “Synchronous inter-core requests”.

Caveats

- On single-core processors this service will always cause an error.
- This service shall only be called for cores which are controlled by the microkernel.

Return value

None.

Service identification

`MK_sid_StartCore` or `OSServiceId_StartCore`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_CoreIdOutOfRange	ASIL	E_OS_ID	The service is called with an invalid core index.
MK_eid_CoreIsAlreadyStarted	ASIL	E_OS_STATE	The service is called for a core which already started.

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_CoreIsNotConfigured	ASIL	E_OS_ID	The service is called for a core which is not configured for the microkernel.
MK_eid_CoreIsAutonomous	ASIL	E_OS_ACCESS	The service is called for a core which is configured to not run under the control of the microkernel.

Name

StartOS — starts the operating system on the current core

Synopsis

```
#include <MicroOs.h>

StatusType StartOS(AppModeType appMode);
```

Description

`StartOS()` terminates the calling thread. Then it activates a QM-OS thread to run the *StartOS* function of the QM-OS. The `appMode` is passed to the QM-OS function.

If the `StartupHook()` feature is configured, the QM-OS is responsible to call the `StartupHook()`.

For a typical example of the startup procedure, see Figure 7.4, "Startup procedure of EB tresos Safety OS".

Return value

`StartOS()` never returns to the caller. The return type is a dummy to satisfy AUTOSAR interface specifications.

Service identification

`MK_sid_StartOs` or `OSServiceId_StartOS`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_CoreIsNotRunning	ASIL	E_OS_STATE	The service is called on a core which has not started yet for AUTOSAR operation.
MK_eid_OsInvalidStartMode	ASIL	E_OS_ID	The service is called with an invalid application mode.
MK_eid_StartOsWhileQmOsBusy	ASIL	E_OS_CALLEV-EL	The service is called on a core on which the QM-OS already runs.

ErrorId	**Mode**	**Error code**	**Description**
	QM		Errors reported by the QM-OS: see the EB tresos AutoCore OS documentation [ASCOS_USERGUIDE].

Name

TerminateApplication — terminates the given OS-Application

Synopsis

#include <MicroOs.h>

```
void TerminateApplication(ApplicationType applicationId, RestartType restart);
```

Description

`TerminateApplication()` terminates all threads of the given OS-Application `applicationId`. If any locks are held by these threads, they are automatically released.

Then the QM-OS is triggered to stop alarms and schedule tables that belong to the OS-Application.

After that, if the `restart` parameter is set to RESTART, the restart task of this OS-Application is started. If the `restart` parameter is set to NO_RESTART, the OS-Application is not restarted and quarantined. A quarantined OS-Application remains terminated until a restart of the system.

If the OS-Application `applicationId` is allocated to a different core than the calling core, `TerminateApplication()` uses a synchronous inter-core terminate application request to terminate the application on that core.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Note that `TerminateApplication()` does not return to its caller if it was called for the OS-Application of the caller and finished successfully.

Service identification

`MK_sid_TerminateApplication` or `OSServiceId_TerminateApplication`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_InvalidRestartParameter	ASIL	E_OS_VALUE	The `restart` parameter is different from RESTART or NO_RESTART.

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_WithoutPermission	ASIL	E_OS_ACCESS	The calling OS-Application does not have the permission to terminate the OS-Application `applicationId`.
MK_eid_InvalidOsApplication	ASIL	E_OS_ID	The `applicationId` parameter does not denote a valid OS-Application, i.e. it is out of range.
MK_eid_Quarantined	ASIL	E_OS_STATE	The OS-Application `applicationId` is already in the state APPLICATION_TERMINATED.

Name

TerminateTask — terminates the calling thread

Synopsis

```
#include <MicroOs.h>

StatusType TerminateTask(void);
```

Description

`TerminateTask()` unconditionally terminates the calling thread. If any locks are held by the thread they are automatically released.

Return value

`TerminateTask()` never returns to its caller. The return type is a dummy to satisfy AUTOSAR interface specifications.

Service identification

`MK_sid_TerminateSelf` or `OSServiceId_TerminateTask`.

Error handling

No errors are reported.

Name

WaitEvent — waits for events

Synopsis

#include <MicroOs.h>

```
StatusType WaitEvent(EventMaskType mask);
```

Description

When a thread runs a task that is designated as EXTENDED calls `WaitEvent()`, the thread is blocked until one or more of the events specified by the `mask` parameter are sent to the task.

If the parameter `mask` is zero or contains one or more events that are already pending for the task, the service returns immediately. Otherwise, it records the value of the parameter for the task and places the task into the WAITING state. The task is still active. Its activation counter retains its value of `1`, but the task is removed from its thread and so can no longer be dispatched.

The task remains in the WAITING state until another thread sends one or more of the events for which it waits or until the task gets forcibly terminated.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_WaitEvent` or `OSServiceId_WaitEvent`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_ThreadIsNotATask	ASIL	E_OS_CALLEV-EL	The calling thread is not a task.

ErrorId	Mode	Error code	Description
MK_eid_TaskIsNotExtended	ASIL	E_OS_ACCESS	The calling thread is a BASIC task.

Name

MK_WaitGetClearEvent, MK_GetClearEvent — wait for events, then automatically return and clear them

Synopsis

```
#include <MicroOs.h>

mk_parametertype_t MK_WaitGetClearEvent(mk_uint32_-
    t eventsToWait, mk_uint32_t *eventsReceived);

mk_parametertype_t MK_GetClearEvent(mk_uint32_t *eventsReceived);
```

Description

When a thread runs a task that is designated as an EXTENDED task calls `MK_WaitGetClearEvent()`, the thread becomes blocked until one or more of the events specified by the `eventsToWait` are sent to the task. This is exactly as described for WaitEvent.

In case of a successful call to `MK_WaitGetClearEvent()`, the return value is `E_OK`, the task's pending events are cleared and the received events are placed into the variable referenced by the `eventsReceived` parameter.

If the call is not successful, the variable referenced by `eventsReceived` is not changed and the pending events do not get cleared.

If the `eventsToWait` parameter is zero, `MK_WaitGetClearEvent()` returns immediately, i.e. the task does not enter the waiting state. This is equivalent to calling `GetEvent()` followed by `ClearEvent()`.

`MK_GetClearEvent()` is implemented as a macro that calls `MK_WaitGetClearEvent()` with the `eventsToWait` parameter set to zero.

A call to `MK_WaitGetClearEvent(EVENTS_EXPECTED, &eventsReceived);` is equivalent to

```
GetTaskID(&myId);
WaitEvent(EVENTS_EXPECTED);
GetEvent(myId, &eventsReceived);
ClearEvent(eventsReceived);
```

A call to `MK_GetClearEvent(&eventsReceived);` is equivalent to

```
GetTaskID(&myId);
```

```
        GetEvent(myId, &eventsReceived);
        ClearEvent(eventsReceived);
```

The advantage of using these APIs is that they have less processing time overhead than the standard AUTOSAR APIs.

Return value

A return value (or status code) of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_WaitGetClearEvent`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_ThreadIsNotATask	ASIL	E_OS_CALLEVEL	The calling thread is not a task.
MK_eid_TaskIsNotExtended	ASIL	E_OS_ACCESS	The calling thread is a BASIC task.

Name

MK_EnableInterruptSource — enables the interrupt source that belongs to the given ISR

Synopsis

#include <MicroOs.h>

```
mk_parametertype_t MK_EnableInterruptSource(mk_objectid_t isrId);
```

Description

`MK_EnableInterruptSource()` enables the interrupt source that is associated with the ISR given by the `isrId` parameter. If the given ISR belongs to an OS-Application that is in the state `APPLICATION_TERMINATED`, an error is reported and the interrupt source is not enabled.

The caller and the ISR must both be allocated to the same core.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_EnableInterruptSource`

Error handling

ErrorId	Mode	Error code	Description
MK_eid_InvalidIsrId	ASIL	E_OS_ID	The `isrId` parameter does not denote a valid ISR.
MK_eid_Quarantined	ASIL	E_OS_STATE	The `isrId` parameter denotes an ISR that belongs to an OS-Application that is in the state `APPLICATION_TERMINATED`.
MK_eid_UnsupportedXCor-eRequest	ASIL	E_OS_-SERVICEID	The `isrId` parameter denotes an ISR that belongs to another core.

Name

MK_DisableInterruptSource — disables the interrupt source that belongs to the given ISR

Synopsis

#include <MicroOs.h>

```
mk_parametertype_t MK_DisableInterruptSource(mk_objectid_t isrId);
```

Description

`MK_DisableInterruptSource()` disables the interrupt source that is associated with the ISR given by the `isrId` parameter.

The caller and the ISR must both be allocated to the same core.

Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

Service identification

`MK_sid_DisableInterruptSource`

Error handling

ErrorId	Mode	Error code	Description
MK_eid_InvalidIsrId	ASIL	E_OS_ID	The `isrId` parameter does not denote a valid ISR.
MK_eid_Quarantined	ASIL	E_OS_STATE	The `isrId` parameter denotes an ISR that belongs to an OS-Application that is in the state `APPLICATION_TERMINATED`.
MK_eid_UnsupportedXCor-eRequest	ASIL	E_OS_-SERVICEID	The `isrId` parameter denotes an ISR that belongs to another core.

Name

MK_ErrorGetParameter, MK_ErrorGetParameterForCore — get a parameter for the last reported error

Synopsis

#include <MicroOs.h>

```
mk_parametertype_t MK_ErrorGetParameterForCore(mk_objectid_t coreIndex, mk_int_fast16_t paramNo);

mk_parametertype_t MK_ErrorGetParameter(mk_int_fast16_t paramNo);
```

Description

`MK_ErrorGetParameterForCore()` returns the parameter number `paramNo` that is stored with the error that was most recently reported on the core specified by `coreIndex`.

If the `coreIndex` is negative, `MK_ErrorGetParameterForCore()` returns the information for the core on which the function is called.

`paramNo` must be in the range 1 to 4. If `paramNo` is outside this range, the result is `-1` cast to `mk_parametertype_t`.

`MK_ErrorGetParameter()` is a macro that calls `MK_ErrorGetParameterForCore()`, which passes `-1` to the `coreIndex`.

`MK_ErrorGetParameter()` is used to implement the AUTOSAR services listed below. You can also call it from your own threads.

Return value

`MK_ErrorGetParameterForCore()` and `MK_ErrorGetParameter()` return the value of the specified parameter that was passed to the API that reported the error. If no error was reported on the specified core since the microkernel started, the return value is undefined.

The return value is also undefined if the service that reported the error does not have the specified parameter. For example, if `ActivateTask()` reported an error, parameter 1 is the task ID and parameters 2, 3, and 4 are undefined.

If you used `MK_ReportError()` to report an error from one of your own threads, parameters 1 and 2 contain the values you passed to `MK_ReportError()`. Parameters 3 and 4 contain `-1`.

The return value is `-1` if `coreIndex` or `paramNo` are out of range.

Service identification

None.

Error handling

No errors are reported. An invalid value for `coreIndex` or `paramNo` is handled via the return value.

Services that use `MK_ErrorGetParameter()`

Service name	Parameter no.
`OSError_CallTrustedFunction_FunctionIndex()`	1
`OSError_CallTrustedFunction_FunctionParams()`	2
`OSError_ActivateTask_TaskID()`	1
`OSError_ChainTask_TaskID()`	1
`OSError_GetTaskID_TaskID()`	1
`OSError_GetTaskState_TaskID()`	1
`OSError_GetTaskState_State()`	2
`OSError_GetResource_ResID()`	1
`OSError_ReleaseResource_ResID()`	1
`OSError_SetEvent_TaskID()`	1
`OSError_SetEvent_Mask()`	2
`OSError_ClearEvent_Mask()`	1
`OSError_WaitEvent_Mask()`	1
`OSError_GetEvent_TaskID()`	1
`OSError_GetEvent_Event()`	2
`OSError_GetAlarm_AlarmID()`	1
`OSError_GetAlarm_Tick()`	2
`OSError_GetAlarmBase_AlarmID()`	1
`OSError_GetAlarmBase_Info()`	2
`OSError_SetRelAlarm_AlarmID()`	1

Service name	Parameter no.
`OSError_SetRelAlarm_increment()`	2
`OSError_SetRelAlarm_cycle()`	3
`OSError_SetAbsAlarm_AlarmID()`	1
`OSError_SetAbsAlarm_start()`	2
`OSError_SetAbsAlarm_cycle()`	3
`OSError_CancelAlarm_AlarmID()`	1
`OSError_StartOS_Mode()`	1
`OSError_ShutdownOS_Error()`	1
`OSError_IncrementCounter_CounterID()`	1
`OSError_StartScheduleTableRel_ScheduleTableID()`	1
`OSError_StartScheduleTableRel_Offset()`	2
`OSError_StartScheduleTableAbs_ScheduleTableID()`	1
`OSError_StartScheduleTableAbs_Tickvalue()`	2
`OSError_StartScheduleTableSynchron_ScheduleTableID()`	1
`OSError_NextScheduleTable_SchedTableID_current()`	1
`OSError_NextScheduleTable_SchedTableID_next()`	2
`OSError_StopScheduleTable_ScheduleTableID()`	1
`OSError_SyncScheduleTable_SchedTableID()`	1
`OSError_SyncScheduleTable_GlobalTime()`	2
`OSError_SetScheduleTableAsync_ScheduleID()`	1
`OSError_GetCounterValue_CounterID()`	1
`OSError_GetCounterValue_Value()`	2
`OSError_TerminateApplication_Application()`	1
`OSError_TerminateApplication_RestartOption()`	2

Table 8.4. Mapping of AUTOSAR error parameter services to parameter numbers

Name

OSErrorGetServiceId, MK_ErrorGetServiceId, MK_ErrorGetServiceIdForCore — get the service ID of the last reported error

Synopsis

#include <MicroOs.h>

```
mk_serviceid_t MK_ErrorGetServiceIdForCore(mk_objectid_t coreIndex);

mk_serviceid_t MK_ErrorGetServiceId(void);

OSServiceIdType OSErrorGetServiceId(void);
```

Description

`MK_ErrorGetServiceIdForCore()` returns the `serviceId` from the error information structure on the specified core, if one is configured. If the `coreIndex` parameter is negative, the error information structure of the caller's core is used.

`MK_ErrorGetServiceId()` is a macro that returns the service ID for the core on which it is called, by calling `MK_ErrorGetServiceIdForCore(-1)`.

`OSErrorGetServiceId()` is a macro that provides an AUTOSAR-compatible API by calling `MK_ErrorGetServiceId()` and converting the return value to `OSServiceIdType`.

Return value

`MK_ErrorGetServiceIdForCore()`, `MK_ErrorGetServiceId()`, and `OSErrorGetServiceId()` return the service ID as specified for each API function of this reference chapter. The return value is `MK_sid_UnknownService` if the `coreIndex` parameter is out of range. The return value is also `MK_sid_UnknownService` if no errors were reported since the microkernel started.

Service identification

None.

Error handling

No errors are reported. An invalid value for `coreIndex` is handled via the return value.

Name

MK_ConditionalGetResource — acquires a resource if necessary

Synopsis

#include <MicroOs.h>

```
StatusType MK_ConditionalGetResource(mk_lockid_t resourceId);
```

Description

`MK_ConditionalGetResource()` acquires the lock with index `resourceId`:

- Either if the current priority of the calling thread is lower than the ceiling priority of the lock.
- Or if the lock is a spinlock or has a spinlock assigned to it.

To do this, it calls `MK_AcquireLock()` and returns the result from that call.

If none of the conditions are fulfilled, `MK_ConditionalGetResource()` does not acquire a lock and returns `E_OS_NOFUNC`. This is no error condition and therefore the `ErrorHook()` is not called.

If `resourceId` is out of range, a resource configured for a core other than the caller's core, a spinlock, or a combined lock, `MK_ConditionalGetResource()` always calls `MK_AcquireLock()`. This ensures correct operation and also means that errors are reported by `MK_AcquireLock()`, not by `MK_ConditionalGetResource()` itself.

Return value

`E_OK` if the resource was acquired successfully.

`E_OS_NOFUNC` if no resource was acquired as the conditions were not fulfilled.

Other return values reflect errors that result from executing the function `MK_AcquireLock()`, see Lock Acquisition Services.

Service identification

None.

Error handling

`MK_ConditionalGetResource()` does not report errors directly. Since `MK_AcquireLock()` is called internally, `MK_ConditionalGetResource()` may report errors of `MK_AcquireLock()`.

Name

MK_IsScheduleNecessary — checks if `Schedule()` activates a higher-priority thread

Synopsis

#include <MicroOs.h>

```
mk_boolean_t MK_IsScheduleNecessary(void);
```

Description

`MK_IsScheduleNecessary()` returns `MK_TRUE` if there is another thread in the thread queue on the caller's core with a higher configured priority than the current thread. Otherwise it returns `MK_FALSE`.

Return value

`MK_FALSE` if a call to schedule is not necessary.

`MK_TRUE` if a call to schedule is necessary.

Service identification

None.

Error handling

No errors are reported.

Name

MK_ScheduleIfNecessary — yields the processor to a higher-priority thread

Synopsis

#include <MicroOs.h>

```
mk_parametertype_t MK_ScheduleIfNecessary(void);
```

Description

`MK_ScheduleIfNecessary()` is a macro that calls `MK_IsScheduleNecessary()`. If this returns `MK_TRUE`, `Schedule()` is called.

The AUTOSAR API `Schedule()` yields the CPU to a thread with a higher configured priority that is blocked because the current thread is configured to be non-preemptable or it uses an internal resource.

You can use this function to replace the AUTOSAR API `Schedule()` to avoid the overhead of calling `Schedule()` in cases where there is normally no thread waiting to execute.

Return value

A return value of `E_OK` indicates a successful completion of the function.

Service identification

`MK_sid_Schedule` or `OSServiceId_Schedule`.

Error handling

No errors are reported.

Name

MK_ReportError — reports an error

Synopsis

#include <MicroOs.h>

```
mk_parametertype_t MK_ReportError(mk_serviceid_t MK_sid, mk_error-
id_t MK_eid, mk_parametertype_t MK_p1, mk_parametertype_t MK_p2);
```

Description

`MK_ReportError()` reports an error via the microkernel's error handling. It is intended to be used by the QM-OS threads and by the microkernel's own library functions that run in the thread's context. You can also call it from your own threads.

To call it from your own thread, call it either directly or indirectly via a microkernel library function. To report the error, the microkernel activates the `ErrorHook()` thread if the `ErrorHook()` feature is enabled and the `ErrorHook()` thread is available. The parameters passed to `MK_ReportError()` are placed into the `MK_errorInfo()` structure. The `MK_eid` is translated into an AUTOSAR standard error code. An unknown error code results in `MK_E_ERROR`. The translated error code is passed as a parameter to the `ErrorHook()` function if it gets activated. It is also returned as a return value from `MK_ReportError()`.

If you pass service codes or error codes which are not between `0` and `32767`, the service code reported by `MK_errorInfo()` is `MK_sid_ReportError` and the respective error code is either `MK_eid_InvalidServiceId` or `MK_eid_InvalidErrorId`.

The semantics of `MK_ReportError()` when called from a QM-OS thread are not documented in detail here. In summary, the QM-OS thread gets terminated. The `ErrorHook()` gets activated if possible. The translated error code is returned to the QM-OS thread's parent thread, which is the thread that made the call to the QM-OS service, if the call was a synchronous call.

The error is reported on the core of the caller, and the `ErrorHook()` runs on the same core.

Return value

If `MK_ReportError()` returns, its return value is the translated error code.

Service identification

`MK_sid_ReportError`.

Error handling

`MK_ReportError()` reports only the errors that it is instructed to report.

Name

SetRelAlarm, MK_AsyncSetRelAlarm, SetAbsAlarm, MK_AsyncSetAbsAlarm, CancelAlarm, MK_AsyncCancelAlarm, GetAlarm — call the alarm services in the QM-OS

Synopsis

```
#include <MicroOs.h>

StatusType SetRelAlarm(AlarmType AlarmId, TickType increment, TickType cycle);

StatusType MK_AsyncSetRelAlarm(AlarmType AlarmId, TickType increment, TickType cycle);

StatusType SetAbsAlarm(AlarmType AlarmId, TickType start, TickType cycle);

StatusType MK_AsyncSetAbsAlarm(AlarmType AlarmId, TickType start, TickType cycle);

StatusType CancelAlarm(AlarmType AlarmId);

StatusType MK_AsyncCancelAlarm(AlarmType AlarmId);

StatusType GetAlarm(AlarmType AlarmId, TickRefType Tick);
```

Description

`SetRelAlarm()`, `SetAbsAlarm()`, and `CancelAlarm()` are macros and `GetAlarm()` is a function that wraps the associated QM-OS call for that service into an interface provided by the microkernel.

They activate the QM-OS thread on the core, to which the alarm `AlarmId` is allocated if necessary and trigger the execution of the corresponding QM-OS function in that thread. After that, they wait for the QM-OS function to finish and then return the result to the caller. If the alarm is allocated to a different core than the caller, they use synchronous inter-core requests to do this.

`MK_AsyncSetRelAlarm()`, `MK_AsyncSetAbsAlarm`, and `MK_AsyncCancelAlarm()` behave like their non-`MK_Async` counterparts with the exception that they use asynchronous inter-core requests if the alarm is allocated to another core. This means that they do not wait for the result of the corresponding operation and therefore cannot return error codes from the QM-OS services in this case but return `E_OK`.

Return value

A return value of `E_OK` indicates that the QM-OS service was called and returned `E_OK`.

A return value of `E_OS_CALLEVEL` could indicate that the QM-OS service could not be started, as described below.

Any other return value is reported by the QM-OS service.

Service identification

`MK_sid_SetRelAlarm` or `OSServiceId_SetRelAlarm`.

`MK_sid_SetAbsAlarm` or `OSServiceId_SetAbsAlarm`.

`MK_sid_CancelAlarm` or `OSServiceId_CancelAlarm`.

`MK_sid_GetAlarm` or `OSServiceId_GetAlarm`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_OsCallFromWrongContext	ASIL	E_OS_CALLEVEL	The call is made from a thread that is not allowed to call QM-OS services, e.g. the QM-OS thread.
	QM		Error codes returned by the QM-OS.

Name

IncrementCounter, MK_AsyncIncrementCounter, GetCounterValue — call the counter services in the QM-OS

Synopsis

```
#include <MicroOs.h>

StatusType IncrementCounter(CounterType CounterID);

StatusType MK_AsyncIncrementCounter(CounterType CounterID);

StatusType GetCounterValue(CounterType CounterID, TickRefType Value);
```

Description

`IncrementCounter()` is a macro and `GetCounterValue()` a function that wraps the associated QM-OS call for that service into an interface provided by the microkernel.

They activate the QM-OS thread on the core, to which the counter `CounterID` is allocated if necessary and trigger the execution of the corresponding QM-OS function in that thread. After that, they wait for the QM-OS function to finish and then return the result to the caller. If the counter is allocated to a different core than the caller, they use synchronous inter-core requests to do this.

`MK_AsyncIncrementCounter()` behaves like `IncrementCounter()` with the exception that it uses an asynchronous inter-core increment counter request if the counter is allocated to another core. This means that it does not wait for the result of the operation and therefore cannot return error codes from the QM-OS service in this case but returns `E_OK`.

Return value

A return value of `E_OK` indicates that the QM-OS service was called and returned `E_OK`.

A return value of `E_OS_CALLEVEL` could indicate that the QM-OS service could not be started, as described below.

Any other return value is reported by the QM-OS service.

Service identification

`MK_sid_IncrementCounter` or `OSServiceId_IncrementCounter`.

`MK_sid_GetCounterValue` or `OSServiceId_GetCounterValue`.

Error handling

ErrorId	Mode	Error code	Description
MK_eid_OsCallFromWrongContext	ASIL	E_OS_CALLEVEL	The call is made from a thread that is not allowed to call QM-OS services, e.g. the QM-OS thread.
	QM		Error codes returned by the QM-OS.

Name

StartScheduleTableAbs, MK_AsyncStartScheduleTableAbs, StartScheduleTableRel, MK_AsyncStartScheduleTableRel, NextScheduleTable, MK_AsyncNextScheduleTable, StopScheduleTable, MK_AsyncStopScheduleTable — call the schedule table services in the QM-OS

Synopsis

#include <MicroOs.h>

```
StatusType StartScheduleTableAbs(ScheduleTableType
    ScheduleTableId, TickType Start);

StatusType MK_AsyncStartScheduleTableAbs(ScheduleTableType
    ScheduleTableId, TickType Start);

StatusType StartScheduleTableRel(ScheduleTableType
    ScheduleTableId, TickType Offset);

StatusType MK_AsyncStartScheduleTableRel(ScheduleTableType
    ScheduleTableId, TickType Offset);

StatusType NextScheduleTable(ScheduleTableType Sched-
    uleTableId_From, ScheduleTableType ScheduleTableId_To);

StatusType MK_AsyncNextScheduleTable(ScheduleTableType Sched-
    uleTableId_From, ScheduleTableType ScheduleTableId_To);

StatusType StopScheduleTable(ScheduleTableType ScheduleTableId);

StatusType MK_AsyncStopScheduleTable(ScheduleTableType ScheduleTableId);
```

Description

`StartScheduleTableAbs()`, `StartScheduleTableRel()`, `NextScheduleTable()`, and `StopScheduleTable()` are macros that wrap the associated QM-OS call for that service into an interface provided by the microkernel.

They activate the QM-OS thread on the core, to which the schedule table `ScheduleTableId` is allocated if necessary and trigger the execution of the corresponding QM-OS function in that thread. After that, they wait for the QM-OS function to finish and then return the result to the caller. If the schedule table is allocated to a different core than the caller, they use synchronous inter-core requests to do this.

MK_AsyncStartScheduleTableAbs(), MK_AsyncStartScheduleTableRel(), MK_AsyncNextScheduleTable(), and MK_AsyncStopScheduleTable() behave like their non-MK_Async counterpart with the exception that they use asynchronous inter-core requests if the schedule table is allocated to another core. This means that they do not wait for the result of the operation and therefore cannot return error codes from the QM-OS service in this case but return E_OK.

Return value

A return value of E_OK indicates that the QM-OS service was called and returned E_OK.

A return value of E_OS_CALLEVEL could indicate that the QM-OS service could not be started, as described below.

Any other return value is reported by the QM-OS service.

Service identification

MK_sid_StartScheduleTableAbs or OSServiceId_StartScheduleTableAbs.

MK_sid_StartScheduleTableRel or OSServiceId_StartScheduleTableRel.

MK_sid_StopScheduleTable or OSServiceId_StopScheduleTable.

MK_sid_NextScheduleTable or OSServiceId_NextScheduleTable.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_OsCallFromWrongContext	ASIL	E_OS_CALLEVEL	The call is made from a thread that is not allowed to call QM-OS services, e.g. the QM-OS thread.
	QM		Error codes returned by the QM-OS.

Name

StartScheduleTableSynchron, MK_AsyncStartScheduleTableSynchron, SyncScheduleTable, MK_-AsyncSyncScheduleTable, SetScheduleTableAsync, MK_AsyncSetScheduleTableAsync — call the schedule table synchronization services in the QM-OS

Synopsis

```
#include <MicroOs.h>

StatusType StartScheduleTableSynchron(ScheduleTableType ScheduleTableId);

StatusType MK_AsyncStartScheduleTableSynchron(ScheduleTableType
                  ScheduleTableId);

StatusType SyncScheduleTable(ScheduleTableType ScheduleTableId, TickType Value);

StatusType MK_AsyncSyncScheduleTable(ScheduleTableType
                  ScheduleTableId, TickType Value);

StatusType SetScheduleTableAsync(ScheduleTableType ScheduleTableId);

StatusType MK_AsyncSetScheduleTableAsync(ScheduleTableType ScheduleTableId);
```

Description

`StartScheduleTableSynchron()`, `SyncScheduleTable()`, and `SetScheduleTableAsync()` are macros that wrap the associated QM-OS call for that service into an interface provided by the microkernel.

They activate the QM-OS thread on the core, to which the schedule table `ScheduleTableId` is allocated if necessary and trigger the execution of the corresponding QM-OS function in that thread. After that, they wait for the QM-OS function to finish and then return the result to the caller. If the schedule table is allocated to a different core than the caller, they use synchronous inter-core requests to do this.

`MK_AsyncStartScheduleTableSynchron()`, `MK_AsyncSyncScheduleTable()` and `MK_-AsyncSetScheduleTableAsync()` behave like their non-`MK_Async` counterpart with the exception that they use asynchronous inter-core requests if the schedule table is allocated to another core. This means that they do not wait for the result of the operation and therefore cannot return error codes from the QM-OS service in this case but return `E_OK`.

Return value

A return value of `E_OK` indicates that the QM-OS service was called and returned `E_OK`.

A return value of `E_OS_CALLEVEL` could indicate that the QM-OS service could not be started, as described below.

Any other return value is reported by the QM-OS service.

Service identification

`MK_sid_StartScheduleTableSynchron` or `OSServiceId_StartScheduleTableSynchron`.

`MK_sid_SyncScheduleTable` or `OSServiceId_SyncScheduleTable`.

`MK_sid_SetScheduleTableAsync` or `OSServiceId_SetScheduleTableAsync`.

Error handling

ErrorId	**Mode**	**Error code**	**Description**
MK_eid_OsCallFromWrongContext	ASIL	E_OS_CALLEVEL	The call is made from a thread that is not allowed to call QM-OS services, e.g. the QM-OS thread.
	QM		Error codes returned by the QM-OS.

Name

MK_GetErrorInfo, MK_GetErrorInfoForCore — provide information about an error

Synopsis

#include <public/Mk_error.h>

```
mk_errorinfo_t *MK_GetErrorInfoForCore(mk_objectid_t coreIndex);

mk_errorinfo_t *MK_GetErrorInfo(void);
```

Description

The microkernel maintains *error information structures* that provide information about the nature of the most recent error that occurred. There is an error information structure for each configured core.

`MK_GetErrorInfoForCore()` is a library function that returns a pointer to the error information structure for the specified core. If the `coreIndex` parameter is negative, `MK_GetErrorInfoForCore()` returns a pointer to the error information structure for the core on which the caller runs.

`MK_GetErrorInfo` is a macro that returns a pointer to the error information structure for the core on which the caller runs.

The error information structure is declared as follows:

```
typedef struct mk_errorinfo_s mk_errorinfo_t;
struct mk_errorinfo_s
{
    mk_serviceid_t serviceId;
    mk_errorid_t errorId;
    mk_osekerror_t osekError;
    mk_objectid_t culpritId;
    mk_objecttype_t culpritType;
    const char *culpritName;
    mk_culprit_t culprit;
    mk_parametertype_t parameter[MK_MAXPARAMS];
};
```

The data fields in the structure are:

serviceId
    An enumeration that identifies the service that detected the error.

errorId
: An enumeration that identifies the error. This identifier provides more details than the standard AUTOSAR/OSEK/VDX status code.

osekError
: An enumeration that gives the AUTOSAR/OSEK/VDX status code for the error. The numerical value of the enumeration is the same as the `StatusType` parameter passed to the `ErrorHook()` function.

culpritId
: The numerical identifier of the executable object that caused the error.

culpritType
: The type of the executable object such as Task, ISR etc. that caused the error.

culpritName
: A pointer to a constant string which contains the name of the executable object that caused the error.

culprit
: A pointer to the thread that causes the error. This is declared with an incomplete structure type so the contents are not available to the application. The information is intended to be used in a debugger. The pointer is only valid for the duration of the `ErrorHook()`.

parameter
: An array of up to four parameters that represents the numerical values of the parameters to the service that detects the error.

The enumerated types are declared in the header files `public/Mk_error.h` and `public/Mk_public_types.h`.

The information in the error information structure is valid and consistent for as long as the `ErrorHook()` instance on the core that caused the error executes. After your `ErrorHook()` terminated, the `culprit` field is no longer valid. The remaining fields are valid but consistency is not guaranteed because a subsequent error can occur at any time and cause the structure to be updated. If you need consistency outside the `ErrorHook()` it is therefore recommended that you add code to your `ErrorHook()` function that copies the information to a private variable.

The microkernel does not guarantee the consistency of the information in the error information structure of another core.

To access the error information structure your configuration must grant your thread read-only access to the microkernel's variables.

Return value

A return value of `MK_NULL` indicates that the specified core index is out of range.

A return value other than `MK_NULL` is a pointer to a valid error information structure. This does not guarantee that you can access the information from your function.

Error handling

No errors are reported.

Name

MK_GetExceptionInfo, MK_GetExceptionInfoForCore — provide information about an exception

Synopsis

#include <public/Mk_error.h>

```
mk_hwexceptioninfo_t *MK_GetExceptionInfoForCore(mk_objectid_t coreIndex);

mk_hwexceptioninfo_t *MK_GetExceptionInfo(void);
```

Description

The microkernel maintains hardware-specific *exception information structures* that provide information about the nature of the most recent processor exception that occurred. There is an exception information structure for each configured core.

`MK_GetExceptionInfoForCore()` is a library function that returns a pointer to the exception information structure for the specified core. If the `coreIndex` parameter is negative, `MK_GetExceptionInfoForCore()` returns a pointer to the exception information structure for the core on which the caller runs.

`MK_GetExceptionInfo` is a macro that returns a pointer to the exception information structure for the core on which the caller runs.

The exception information structure is hardware-specific and is described in MK_exceptionInfo.

To access the exception information structure your configuration must grant your thread read-only access to the microkernel's variables.

Return value

A return value of `MK_NULL` indicates that the specified core index is out of range.

A return value other than `MK_NULL` is a pointer to a valid exception information structure. This does not guarantee that you can access the information from your function.

Error handling

No errors are reported.

Name

MK_GetPanicExceptionInfo, and MK_GetPanicExceptionInfoForCore — provides information about an exception which caused a kernel panic

Synopsis

#include <public/Mk_error.h>

```
mk_hwexceptioninfo_t *MK_GetPanicExceptionInfoForCore(mk_objectid_t coreIndex);
```

```
mk_hwexceptioninfo_t *MK_GetPanicExceptionInfo(void);
```

Description

Whether your hardware supports the `MK_GetPanicExceptionInfo` feature is described in MK_exceptionInfo.

The microkernel maintains hardware-specific *exception information structures* that provide information about the nature of the processor exception that occurred in kernel context and caused a kernel panic. There is an exception information structure for each configured core.

`MK_GetPanicExceptionInfoForCore()` is a library function that returns a pointer to the exception information structure for the specified core. If the `coreIndex` parameter is negative, `MK_GetPanicExceptionInfoForCore()` returns a pointer to the exception information structure for the core on which the caller runs.

`MK_GetPanicExceptionInfo` is a macro that returns a pointer to the exception information structure for the core on which the caller runs.

The exception information structure is hardware-specific and is described in MK_exceptionInfo.

To access the exception information structure your configuration must grant your thread read-only access to the microkernel's variables.

The obtained exception information is only valid, if `MK_GetPanicExceptionInfo` is called during the `ShutdownHook()` and `MK_GetPanicReason()` returns `MK_panic_ExceptionFromKernel`.

You can inspect the panic exception information in your debugger, by accessing the data structures referenced by `MK_panicExceptionInfo` where the core index is the key to this table.

If the microkernel encounters another exception while handling a panic which was caused by an exception, it is undefined to which exception the obtained information belongs. Note that in this case the microkernel goes to `MK_PanicStop()` instead of invoking the `ShutdownHook()`.

Return value

A return value of `MK_NULL` indicates that the specified core index is out of range, or that the exception information structure is not provided for this hardware.

A return value other than `MK_NULL` is a pointer to a valid exception information structure. This does not guarantee that you can access the information from your function.

Error handling

No errors are reported.

Name

MK_ReadTime — reads the time

Synopsis

```
#include <MicroOs.h>

void MK_ReadTime(mk_time_t *timeRef);
```

Description

`MK_ReadTime()` reads the current value of a long-duration timer. The current value is placed into a variable provided by the caller. The address of the caller is specified by the `timeRef` parameter. The caller must have permission to modify this variable.

You can call `MK_ReadTime()` from any thread, regardless of the privilege level. You must ensure on some systems that the calling threads have read access to the hardware timer that provides the time, see Section 9.6, "Timers on TriCore family processors".

`MK_ReadTime()` is a macro with a hardware-dependent implementation.

Return value

None.

Service identification

None.

Error handling

No errors are reported.

Name

MK_DiffTime, MK_DiffTime32 — calculate the length of an interval between two times

Synopsis

```
#include <MicroOs.h>

void MK_DiffTime(mk_time_t *diffTime, const mk_-
    time_t *newTime, const mk_time_t *oldTime);

mk_uint32_t MK_DiffTime32(const mk_time_t *newTime, const mk_time_t *oldTime);
```

Description

`MK_DiffTime()` calculates the difference newTime - oldTime, i.e. the duration of the interval that starts at oldTime and ends at newTime. The two input values are variables provided by the caller whose addresses are passed as parameters. The result is placed into the variable whose address is specified by the `diffTime` parameter. The caller must have permission to modify this variable.

`MK_DiffTime32()` calculates the difference newTime - oldTime as in `MK_DiffTime()` but returns the result as a 32-bit number. If the time difference is too large to be represented in 32 bits, the function returns the maximum value `0xffffffff` that can be represented.

Return value

`MK_DiffTime()` has no return value. `MK_DiffTime32()` returns the time difference.

Service identification

None.

Error handling

No errors are reported.

Name

MK_ElapsedTime, MK_ElapsedTime32 — calculate the time elapsed since a previous call

Synopsis

```
#include <MicroOs.h>

void MK_ElapsedTime(mk_time_t *elapsedTime, mk_time_t *previousTime);

mk_uint32_t MK_ElapsedTime32(mk_time_t *previousTime);
```

Description

`MK_ElapsedTime()` places the amount of time that has elapsed since a previous call to the function into the output variable whose address is passed in the `elapsedTime` parameter. `MK_ElapsedTime()` obtains the time of the previous call from the variable whose address is passed in the `previousTime` parameter. After the calculation is performed, `MK_ElapsedTime()` updates the previous time variable with the time of this call. The caller must have permission to modify both variables.

`MK_ElapsedTime32()` calculates the elapsed time as in `MK_ElapsedTime()` but returns the result as a 32-bit number. If the time difference is too large to be represented in 32 bits, the function returns the maximum value `0xffffffff` that can be represented. The caller must have permission to modify the referenced variable.

To initialize the `previousTime` variable, call `MK_ElapsedTime()` or `MK_ElapsedTime32()` and discard the result.

Return value

`MK_ElapsedTime()` has no return value. `MK_ElapsedTime32()` returns the elapsed time.

Service identification

None.

Error handling

No errors are reported.

Name

MK_ElapsedMicroseconds — calculates the time elapsed since a previous call, scaled by a given factor

Synopsis

#include <MicroOs.h>

```
void MK_ElapsedMicroseconds(mk_uint32_t *previous-
Time, mk_uint32_t *elapsedTime, mk_uint16_t factor);

void MK_ElapsedTime1u(mk_uint32_t *previousTime, mk_uint32_t *elapsedTime);

void MK_ElapsedTime10u(mk_uint32_t *previousTime, mk_uint32_t *elapsedTime);

void MK_ElapsedTime100u(mk_uint32_t *previousTime, mk_uint32_t *elapsedTime);
```

Description

WARNING ⚠

Deprecated

The function `MK_ElapsedMicroseconds()` as well as the related macros `MK_ElapsedTime1u()`, `MK_ElapsedTime10u()`, and `MK_ElapsedTime100u()` are deprecated. Use `MK_ElapsedTime()` or `MK_ElapsedTime32()` instead.

`MK_ElapsedMicroseconds()` places the amount of time that has elapsed since a previous call to the function into the output variable whose address is passed in the `elapsedTime` parameter. `MK_ElapsedMicroseconds()` obtains the time of the previous call from the variable whose address is passed in the `previousTime` parameter. After the calculation is performed, `MK_ElapsedMicroseconds()` updates the previous time variable with the time of this call. The caller must have permission to modify both variables.

The input and output values are scaled by dividing the time obtained by calling `MK_ReadTime()`. The scaling factor is supplied in the `factor` parameter.

`MK_ElapsedTime1u()`, `MK_ElapsedTime10u()`, and `MK_ElapsedTime100u()` are macros that are implemented by calls to `MK_ElapsedMicroseconds()`. These macros pass the configured constants `MK_timestampClockFactor1u`, `MK_timestampClockFactor10u`, and `MK_timestampClockFactor100u` respectively as the `factor` parameter.

Caveat

These services have no overflow detection, and the scaling operation can result in large rounding errors.

Return value

None.

Service identification

None.

Error handling

No errors are reported.

Name

MK_GetPanicReason, MK_GetPanicReasonForCore — get the panic reason

Synopsis

#include <public/Mk_error.h>

```
mk_panic_t MK_GetPanicReasonForCore(mk_objectid_t coreIndex);

mk_panic_t MK_GetPanicReason(void);
```

Description

`MK_GetPanicReasonForCore()` returns the panic reason for the given core `coreIndex`. If `coreIndex` is a negative number, the function returns the panic reason for the calling core.

The panic reason is an enum value.

`MK_GetPanicReason()` returns the panic reason for the calling core and is implemented as a macro that calls `MK_GetPanicReasonForCore(-1)`.

Return value

The panic reason for the given core if that core is valid and controlled by the microkernel. Negative `coreIndex` values refer to the calling core.

`MK_panic_Unknown` if the given core is invalid, i. e. too large or if the given core is not controlled by the microkernel.

Error handling

No errors are reported.

Name

MK_GetProtectionInfo, MK_GetProtectionInfoForCore — provide information about a protection fault

Synopsis

#include <public/Mk_error.h>

```
mk_protectioninfo_t *MK_GetProtectionInfoForCore(mk_objectid_t coreIndex);
```

```
mk_protectioninfo_t *MK_GetProtectionInfo(void);
```

Description

The microkernel maintains *protection fault information structures* that provide information about the nature of the most recent protection fault that occurred. There is a protection fault information structure for each configured core.

`MK_GetProtectionInfoForCore()` is a library function that returns a pointer to the protection fault information structure for the specified core. If the `coreIndex` parameter is negative, `MK_GetProtectionInfoForCore()` returns a pointer to the protection fault information structure for the core on which the caller runs.

`MK_GetProtectionInfo` is a macro that returns a pointer to the protection fault information structure for the core on which the caller runs.

The protection fault information structure is declared as follows:

```
typedef struct mk_protectioninfo_s mk_protectioninfo_t;
struct mk_protectioninfo_s
{
    mk_serviceid_t serviceId;
    mk_errorid_t errorId;
    mk_osekerror_t osekError;
    mk_culprit_t culprit;
    mk_objectid_t culpritApplicationId;
};
```

The data fields in the structure are:

serviceId
: An enumeration that identifies the service that detected the protection fault. This is normally `MK_sid_ExceptionHandling`.

errorId
: An enumeration that identifies the protection fault. This identifier provides more detail than the standard AUTOSAR/OSEK/VDX status code.

osekError
: An enumeration that gives the AUTOSAR/OSEK/VDX status code for the protection fault. The numerical value of the enumeration is the same as the `StatusType` parameter passed to the `ProtectionHook()` function.

culprit
: A pointer to the thread that caused the error. This is declared with an incomplete structure type so the contents are not available to the application. The information is intended to be used in a debugger. The pointer is only valid for the duration of the `ProtectionHook()`. Moreover, this pointer becomes invalid if the culprit is terminated.

culpritApplicationId
: The identifier of the *OS-Application* which caused the error. The numerical value of this identifier equals the respective `ApplicationType` value.

The enumerated types are declared in the header file `public/Mk_error.h`.

The information in the protection fault information structure is valid and consistent for as long as the `ProtectionHook()` instance on the core that caused the fault executes. After your `ProtectionHook()` terminated, the `culprit` field is no longer valid. The remaining fields are valid but consistency is not guaranteed because a subsequent protection fault can occur at any time and cause the structure to be updated. If you need consistency outside the `ProtectionHook()`, it is therefore recommended that you add code to your `ProtectionHook()` function that copies the information to a private variable.

The microkernel does not guarantee the consistency of the information in the protection fault information structure of another core.

To access the protection fault information structure, your configuration must grant your thread read-only access to the microkernel's variables.

Return value

A return value of `MK_NULL` indicates that the specified core index is out of range, or that the protection fault information structure for the given core is omitted by the configuration.

A return value other than `MK_NULL` is a pointer to a valid error information structure. This does not guarantee that you can access the information from your function.

Error handling

No errors are reported.

Name

MK_InitSyncHere — initializes microkernel data for core synchronization

Synopsis

#include <MicroOs.h>

```
void MK_InitSyncHere(void);
```

Description

`MK_InitSyncHere()` initializes the synchronization data which are used to synchronize the different cores controlled by the microkernel in a multi-core environment.

Caveat

- This service shall be called before any of the cores controlled by the microkernel transfers control to the label `MK_Entry2`.

  WARNING **Do not call this service after control is transferred to `MK_Entry2`**
  Never call this service after the control is transferred to `MK_Entry2` on any core.

- This service shall only be called on one core in a multi-core environment.
- This service requires a minimal C-environment to be present when it is called. In particular this means:
  - It must be possible to access data and code. Note that the data need not be initialized, they must only be accessible.
  - It must be possible to read and write function-local data, e.g. in stack-based systems, a valid stack must be present.
  - It must be possible to call other functions, e.g. in systems with hardware context storage for calls, this context storage must be set up.

Return value

None.

Service identification

None.

Error handling

No errors are reported.

Name

MK_MulDiv — multiplication followed by division, with saturation

Synopsis

#include <MicroOs.h>

```
mk_uint32_t MK_MulDiv(mk_uint32_t in, mk_uint16_t m, mk_uint16_t d);
```

Description

`MK_MulDiv()` multiplies the input number `in` by the multiplier `m` and divides the intermediate result by the divisor `d`. If the resulting value can be represented in 32 bits, `MK_MulDiv()` returns that value. Otherwise the return value is `0xffffffff`.

The input parameter `in` can be any 32-bit unsigned number. The multiplication and division factors `m` and `d` must be in the range 0..65535 (d != 0). The range is *not* checked. Out-of-range values can result in incorrect results.

The result is guaranteed to be free from errors caused by internal overflow, provided that the parameters are in range.

Return value

`(in * m)/d`, or `0xffffffff`.

Service identification

None.

Error handling

No errors are reported. In particular, out-of-range values for `m` and `d` are not detected.

Name

MK_SuspendInterrupts — is activated when one of the suspend interrupt lock functions is used

Synopsis

```
#include <MicroOs.h>

void MK_SuspendInterrupts(mk_lockid_t resourceId);
```

Description

The function combines different AUTOSAR specific suspend interrupt function calls (`SuspendOSInterrupts()`, `SuspendAllInterrupts()` and `DisableAllInterrupts()`) to one generic interface. Internally the function compares the queueing priority of the caller with the ceiling priority of the provided lock and calls the `MK_UsrAcquireLock()` syscall if the priority is lower otherwise it does nothing.

The `resourceId` parameter shows which resource should be acquired. The following values are possible:

- `MK_resLockCat1` for `SuspendAllInterrupts()` and `DisableAllInterrupts()`
- `MK_resLockCat2` for `SuspendOSInterrupts()`

The function must not be called directly instead use always the provided AUTOSAR interface.

Return value

None.

Name

MK_ResumeInterrupts — is activated when one of the resume interrupt lock functions is used

Synopsis

```
#include <MicroOs.h>

void MK_ResumeInterrupts(mk_lockid_t resourceId);
```

Description

The function combines different AUTOSAR specific resume interrupt function calls (`ResumeOSInterrupts()`, `ResumeAllInterrupts()` and `EnableAllInterrupts()`) to one generic interface. Internally the function compares the queueing priority of the caller with the ceiling priority of the provided lock and calls the `MK_UsrReleaseLock()` syscall if the priority is lower otherwise it does nothing.

The `resourceId` parameter shows which resource should be released. The following values are possible:

- `MK_resLockCat1` for `ResumeAllInterrupts()` and `EnableAllInterrupts()`
- `MK_resLockCat2` for `ResumeOSInterrupts()`

The function must not be called directly instead use always the provided AUTOSAR interface.

Return value

None.

Name

MK_SstStartCounter — starts a counter and its associated simple schedule table

Synopsis

#include <public/Mk_sst_api.h>

```
StatusType MK_SstStartCounter(mk_objec-
tid_t counterId, mk_parametertype_t delay);
```

Description

`MK_SstStartCounter` starts the counter and associated simple schedule table specified by the `counterId` parameter. The schedule table starts executing after `delay` ticks of the counter. If the schedule table has an expiry point at offset `0` it is processed at this time.

If the counter is configured to be driven by a hardware timer interrupt, `MK_SstStartCounter` initializes the hardware to trigger interrupts at the configured frequency.

The `delay` parameter must be in the range `1` to the `OsCounterMaxAllowedValue` configured for the counter.

Return value

A return value of `E_OK` indicates that the counter has been successfully started. Any other return value indicates an error code as described below.

Error handling

Status code	Description
E_OS_ID	The `counterId` parameter does not identify a configured counter, or is not assigned to the core of the caller.
E_OS_VALUE	The `delay` is out of range.
E_OS_STATE	The counter is in a state where it cannot be started, i.e. it is not in state STOPPED.

Name

MK_SstAdvanceCounter — advances a counter and its associated simple schedule table

Synopsis

#include <public/Mk_sst_api.h>

```
StatusType MK_SstAdvanceCounter(mk_objec-
tid_t counterId, mk_parametertype_t nTicks);
```

Description

`MK_SstAdvanceCounter()` advances the counter and associated simple schedule table specified by the `counterId` parameter. The counter's value increases by `nTicks` ticks (wrapping around as necessary). All expiry points on the schedule table that become due are processed.

The `nTicks` parameter must be in the range `1` to the `OsCounterMaxAllowedValue` configured for the counter.

`MK_SstAdvanceCounter()` reports errors that are detected while it processes the expiry points, but continues to process expiry points. For further details, see Section 7.3, “Error handling”.

Return value

A return value of `E_OK` indicates that the counter has been successfully advanced and that no immediate errors were detected during the processing of expiry points. However, if this processing involves cross-core operations, the ErrorHook might be called on remote cores to report errors encountered there. Any other return value indicates an error code as described below.

Note, that only the *first* error detected during activate task or set event operations is returned by this function. These operations are executed because of expiry point actions, which become due as a consequence of advancing a SST counter. Any subsequent errors are lost.

Error handling

Status code	Description
E_OS_ID	The `counterId` parameter does not identify a configured counter, or is not assigned to the core of the caller.

Status code	Description
E_OS_VALUE	The parameter `nTicks` is out of range.
E_OS_LIMIT	A task that was due to be activated could not be activated because it has reached its activation limit.
E_OS_STATE	The counter is not in a state where it can be advanced, i.e. it is not in state STARTED or RUNNING. It may also mean, that at least one set event operation failed, because the OS application of the task for which to set the event is quarantined.
E_OS_ACCESS	The core of the activated task or for which events are set, is not an AUTOSAR core.
MK_E_INTERNAL	The core of the activated task or for which events are set, is corrupted.

Name

MK_SstStopCounter — stops a counter and its associated simple schedule table

Synopsis

#include <public/Mk_sst_api.h>

```
StatusType MK_SstStopCounter(mk_objectid_t counterId);
```

Description

`MK_SstStopCounter` stops the counter and associated simple schedule table specified by the `counterId` parameter. The counter stops immediately and no more expiry points on the schedule table are processed.

If the counter is configured to be driven by a hardware timer interrupt, `MK_SstStopCounter` stops the hardware timer.

Return value

A return value of `E_OK` indicates that the counter has been successfully stopped. Any other return value indicates an error code as described below.

Error handling

Status code	**Description**
E_OS_ID	The `counterId` parameter does not identify a configured counter, or is not assigned to the core of the caller.
E_OS_NOFUNC	The counter is in a state where it cannot be stopped, i.e. it is not in state STARTED or RUNNING.

Name

MK_NsToTicks_f etc. — converts between nanoseconds and ticks

Synopsis

```
#include <Mk_timeconversion.h>

mk_uint32_t MK_NsToTicks_f(mk_uint32_t ns);

mk_uint32_t MK_TicksToNs_f(mk_uint32_t tk);

mk_uint32_t MK_NsToTicksF_f(mk_uint32_t ns);

mk_uint32_t MK_TicksToNsF_f(mk_uint32_t tk);
```

Description

The macros `MK_NsToTicks_f()` and `MK_NsToTicksF_f()` convert the number of nanoseconds given by `ns` to the equivalent number of ticks of a clock running at frequency f.

The macros `MK_TicksToNs_f()` and `MK_TicksToNsF_f()` convert the number of ticks of a clock which runs at frequency f, given by `tk`, to the equivalent number of nanoseconds. If the resulting value is too large to be represented in 32 bits, the result is `0xffffffff`.

`MK_NsToTicks_f()` and `MK_TicksToNs_f()` are implemented in such a way that you can evaluate them at compile time, so you can use them to initialize constants.

`MK_NsToTicksF_f()` and `MK_TicksToNsF_f()` in most cases use MK_MulDiv to achieve better accuracy. You should not use them to initialize constants or in other places where CPU time is limited.

Return value

The converted value in all cases.

Service identification

None.

Error handling

No errors are reported.

Name

MK_TimestampNsToTicks etc. — converts between nanoseconds and ticks for the time stamp timer

Synopsis

#include <Mk_board.h>

```
mk_uint32_t MK_TimestampNsToTicks(mk_uint32_t ns);

mk_uint32_t MK_TimestampTicksToNs(mk_uint32_t tk);

mk_uint32_t MK_TimestampNsToTicksF(mk_uint32_t ns);

mk_uint32_t MK_TimestampTicksToNsF(mk_uint32_t tk);
```

Description

The macros `MK_TimestampNsToTicks()` and `MK_TimestampNsToTicksF()` convert the number of nanoseconds given by `ns` to the equivalent number of ticks of the time stamp timer.

The macros `MK_TimestampTicksToNs()` and `MK_TimestampTicksToNsF()` convert the number of ticks of the time stamp timer given by `tk` to the equivalent number of nanoseconds. If the resulting value is too large to be represented in 32 bits, the result is `0xffffffff`.

`MK_TimestampNsToTicks()` and `MK_TimestampTicksToNs()` are implemented in such a way that you can evaluate them at compile time, so you can use them to initialize constants.

`MK_TimestampNsToTicksF()` and `MK_TimestampTicksToNsF()` in most cases use MK_MulDiv to achieve better accuracy. You should not use them to initialize constants or in places where CPU time is limited.

Note

The frequency of the time stamp timer depends on the hardware used. The macros are defined in the board header file Mk_board.h, which you must provide or adapt.

Return value

The converted value in all cases.

Service identification

None.

Error handling

No errors are reported.

Name

mk_time_t — data type for time services

Synopsis

```
#include <MicroOs.h>

typedef struct mk_time_s mk_time_t
```

Description

`mk_time_t` is a 64-bit data type that you can use to store and manipulate absolute time values and long-duration intervals without the need to take special care of overflow. It is implemented by a structure that contains two unsigned 32-bit integers; `timeHi` and `timeLo`. For future compatibility, the endianness of the processor determines the order of the two members in the structure.

Return value

Not applicable.

Service identification

Not applicable.

Error handling

Not applicable.

Name

mk_statusandvalue_t — a data type for API services that returns a status code and a requested value

Synopsis

#include <MicroOs.h>

```
typedef struct mk_statusandvalue_s mk_statusandvalue_t

struct mk_statusandvalue_s { mk_parametertype_t sta-
tusCode; mk_parametertype_t requestedValue; }
```

Description

`mk_statusandvalue_t` is a structure data type that is used as the return value from microkernel API functions that need to return a status code and some requested information.

Many API services defined by AUTOSAR have a status code return value and place requested information into a variable whose address is passed by the caller. The microkernel cannot assume that it has write access to variables that the thread uses, including variables on the stack. Even if the microkernel has access, validating pointer parameters incurs a large CPU time overhead.

The microkernel API takes the following approach: The system call handler places the two parts of the return value into the thread's registers. Then the assembly language stub function converts the return value into a C-structure return value. A library function provides the AUTOSAR interface. The library function calls the corresponding microkernel API, places the requested information into the referenced variable and returns the status code.

Return value

Not applicable.

Service identification

Not applicable.

Error handling

Not applicable.

8.3. Microkernel callout reference

The microkernel supports four types of callout functions that are specified by AUTOSAR:

- The global startup hook
- The global shutdown hook
- The global error hook
- The protection hook

NOTE

No support for OS-Application-specific hook functions

OS-Application-specific hook functions are not supported.

The following additional callout functions are provided by the microkernel:

- The interrupt locking callout
- The interrupt unlocking callout

The callout functions execute on the cores on which they are started. On systems with more than one core, this means that there can be several instances of the functions which execute in parallel. Your implementation must synchronize access to resources, including variables with global lifetime.

The callout functions are described in the following sections.

Name

StartupHook — is activated by the StartOS service

Synopsis

#include <MicroOs.h>

```
void StartupHook(void);
```

Description

If the startup hook is enabled in the configuration, the microkernel expects that the QM-OS calls `StartupHook()` during startup. In that case, the startup hook is executed in the QM-OS thread, see also Figure 7.4, “Startup procedure of EB tresos Safety OS”.

The system designer must provide `StartupHook()`.

Return value

None.

Name

ShutdownHook — is activated at shutdown

Synopsis

```
#include <MicroOs.h>

void ShutdownHook(StatusType errorCode);
```

Description

If the shutdown hook is enabled in the configuration, the microkernel activates a thread to run the `ShutdownHook()` function whenever the system shuts down.

The `errorCode` parameter indicates the nature of the error that causes the shutdown. Note that this is an error code specified by AUTOSAR and does not give full details about the error. More information are normally available in the MK_errorInfo structure.

If the shutdown is initiated by a call to ShutdownOS, the parameter is the parameter passed to this service. In this case, no additional information is available.

The system designer must provide `ShutdownHook()`.

Return value

None.

Name

ErrorHook — is activated when an error is reported

Synopsis

#include <MicroOs.h>

```
void ErrorHook(StatusType errorCode);
```

Description

If the error hook is enabled in the configuration, the microkernel activates a thread to run the `ErrorHook()` function whenever an error is detected in a microkernel or QM-OS service. The microkernel also activates this thread when a thread calls MK_ReportError. The thread runs before control returns to the caller of the service.

The `errorCode` parameter indicates the nature of the error. Note that this is an error code specified by AUTOSAR and does not give full details about the error. More information are normally available in the MK_errorInfo structure.

The system designer must provide `ErrorHook()`.

Return value

None.

Name

ProtectionHook — is activated when a protection fault is detected

Synopsis

```
#include <MicroOs.h>

ProtectionReturnType ProtectionHook(StatusType errorCode);
```

Description

If the protection hook is enabled in the configuration, the microkernel activates a thread to run the `ProtectionHook()` function whenever the microkernel detects a protection fault. The thread runs immediately. Its return value is used to determine the system's response to the protection fault.

The `errorCode` parameter indicates the nature of the protection fault. Note that this is an error code specified by AUTOSAR and does not give full details about the fault. More information are available in the MK_protectionInfo structure.

The system designer must provide the `ProtectionHook()`.

Return value

`ProtectionHook()` should return one of the values described in Section 7.3.3, “Protection hook return values”.

Name

MK_SuspendCallout — project-specific interrupt locking

Synopsis

#include <MicroOs.h>

```
void MK_SuspendCallout(mk_lockid_t resourceId);
```

Description

The function is a user provided call-out which can be used to implement project-specific interrupt locking. If you define the macro `MK_USE_INT_LOCKING_CALLOUTS` to a non-zero value, the microkernel calls this function instead of its normal implementations of `SuspendOSInterrupts()`, `SuspendAllInterrupts()` and `DisableAllInterrupts()`.

The `resourceId` parameter shows which resource should be acquired. The following values are possible:

- `MK_resLockCat1` for `SuspendAllInterrupts()` and `DisableAllInterrupts()`
- `MK_resLockCat2` for `SuspendOSInterrupts()`

If you wish to implement the interrupt locking functionality from this function by using microkernel services, call `MK_SuspendInterrupts(resourceId)`.

Do not call `MK_SuspendCallout()` directly. Always use the AUTOSAR API.

WARNING

Possible dangers of `MK_SuspendCallout()`

The application can call this function from any executable of any safety integrity level, including ISRs and hook functions. The calls could be nested, both within a single executable and between exectutables. Also, the function could execute simultaneously on two or more cores.

Return value

None.

Name

MK_ResumeCallout — project-specific interrupt unlocking

Synopsis

#include <MicroOs.h>

```
void MK_ResumeCallout(mk_lockid_t resourceId);
```

Description

The function is a user provided call-out which can be used to implement project-specific interrupt unlocking. If you define the macro `MK_USE_INT_LOCKING_CALLOUTS` to a non-zero value, the microkernel calls this function instead of its normal implementations of `ResumeOSInterrupts()`, `ResumeAllInterrupts()` and `EnableAllInterrupts()`.

The `resourceId` parameter shows which resource should be released. The following values are possible:

- `MK_resLockCat1` for `ResumeAllInterrupts()` and `EnableAllInterrupts()`
- `MK_resLockCat2` for `ResumeOSInterrupts()`

If you wish to implement the interrupt unlocking functionality from this function by using microkernel services, call `MK_ResumeInterrupts(resourceId)`.

Do not call `MK_ResumeCallout()` directly. Always use the AUTOSAR API.

WARNING

Possible dangers of `MK_ResumeCallout()`

The application can call this function from any executable of any safety integrity level, including ISRs and hook functions. The calls could be nested, both within a single executable and between exectutables. Also, the function could execute simultaneously on two or more cores.

Return value

None.

8.4. Microkernel configuration reference

This section describes the configuration macros that you can use to configure the microkernel. If you use EB tresos Studio to configure your system, you can find the definitions in `Mk_gen_config.h`, unless explicitly noted otherwise. Note that EB tresos Studio does not allow you to configure everything that is documented in this section. In many cases the default values of the macros that are provided by the microkernel are correct. In some cases, overriding the default value can result in a system that is not strictly compatible with the AUTOSAR specification.

The configuration macros are divided into three categories:

- System-wide configuration parameters
- Core-specific configuration parameters for which no defaults exist
- Core-specific configuration parameters for which suitable defaults are defined

The system-wide configuration macros are named `MK_CFG_<ConfigItem>`. These macros are described in Section 8.4.1, “System-wide configuration macros”.

The core-specific configuration macros are named `MK_CFG_Cx_<ConfigItem>`. These macros are described in Section 8.4.2, “Core-specific configuration parameters without default values” and Section 8.4.3, “Core-specific configuration parameters with default values”.

You must define all of the macros using C-preprocessor directives (`#define`). In some cases you must provide additional configuration such as function prototypes, variable declarations etc.

8.4.1. System-wide configuration macros

The configuration macros described in this section define the configuration of the system as a whole, and affect all cores in a multi-core configuration. Unless stated otherwise, these macros must be defined for every configuration; there are no default values.

MK_CFG_HWMASTERCOREINDEX. The physical core index of the core that is considered to be the master core. Valid values lie in the range from `0` to `(MK_MAXCORES-1)`. `MK_MAXCORES` is defined in the microkernel header files. For single-core processors `MK_CFG_HWMASTERCOREINDEX` must be `0`. You can choose the master core freely from among the configured cores. There are no hardware-defined restrictions.

MK_CFG_COREMAP_n. Each of these macros, for `n=0,1,2,...` specifies, for a set of 32 of the available physical core indices, which of these cores is configured as part of the system. Each macro must be defined as either `0` or a bitwise `OR` of `MK_COREMAPBIT_Cx`, where `x` is in the range `(n*32)` to `(n*32)+31`. There must be at least one non-zero bit in the entire set of macros. There is no currently-supported hardware with more than 32 cores, so you need only define `MK_CFG_COREMAP_0` as a bitwise `OR` of bits for cores 0 to `(MK_MAXCORES-1)`.

MK_CFG_MAXMSG. The length of the inter-core message queues. Valid values lie in the range from 2 upwards. The higher the number, the more buffer space is available for nested and asynchronous inter-core requests before the microkernel blocks itself while waiting for space. Increasing the length of the queues uses more memory.

MK_SCHEDULERPRIO. The scheduler priority. This is the priority of the `RES_SCHEDULER` resource. It should be at least as high as every task priority but lower than the QM-OS thread.

MK_CAT2LOCKPRIO. The priority of the resource that locks category 2 interrupts. It should be at least as high as every category 2 ISR priority but lower than the category 1 ISR priorities.

MK_CAT2LOCKLEVEL. The interrupt lock level to disable all category 2 ISRs.

MK_CAT1LOCKPRIO. The priority of the resource that locks category 1 and 2 interrupts. It should be at least as high as every category 1 ISR priority but lower than the error hook and protection hook.

MK_CAT1LOCKLEVEL. The interrupt lock level to disable all category 1 and 2 ISRs.

MK_CFG_NADDONS. The number of configured add-ons.

MK_HAS_USERPANICSTOP. Set to `1` if the configuration macro `MK_USERPANICSTOP` provides a user-defined startup panic stop function. Set to `0` otherwise.

MK_USERPANICSTOP(reason). If the configuration macro `MK_HAS_USERPANICSTOP` enables the user-defined startup panic stop function, this macro also defines the function. The function runs if the kernel reaches a fatal state during startup in which no normal shutdown is possible. The parameter `reason`, which might be handed on the user-defined startup panic stop function, is of type `mk_panic_t`. The user-defined startup panic stop function shall never return. If it returns anyway, the processor enters an endless loop.

WARNING

Startup panic stop mechanism

The startup panic stop mechanism is called only in very fatal cases during the startup of the system, e.g. misconfigurations or hardware defects. Never expect that you can do *anything useful* in the user-defined startup panic stop function.

MK_CFG_NAPPLICATIONS. The number of *OS-Applications* that are present in your configuration. Valid values lie in the range from `0` to `32767`, although the QM-OS and available memory space impose lower restrictions.

MK_CFG_APPLIST. A comma-separated list of application configurations, each of which is created by invoking the macro `MK_APPCONFIG(d, x, rt)` with suitable parameters. The parameters specify the address of the dynamic state variables for the application (`c`), the physical index of the core on which the application is configured (`x`), and the restart task (`rt`), respectively. The parameter `d` must be `&MK_cx_appDynamic[i]`, where `x` is the index of the core on which the OS-Application is configured. This value `x` must be the same as the parameter `x` in the invocation of `MK_APPCONFIG`. You must choose `i` so that no two OS-Applications use the same state variables. The parameter `rt` is the ID of the restart task for this OS-Application. A value of `INVALID_TASK` indicates that no restart task is configured for the OS-Application. The index of an OS-

Application in this list must match the OS-Application's ID that you have defined in `Mk_gen_user.h`. If `MK_-CFG_NAPPLICATIONS` is zero, `MK_CFG_APPLIST` is not used and you can omit its definition.

MK_CFG_NTRUSTEDFUNCTIONS. The total number of trusted functions that are configured.

MK_CFG_TRUSTEDFUNCTIONLIST. A comma-separated list of trusted function configurations as defined in Section 8.4.12, "Configuration of trusted functions". You might find it helpful to define a macro for each trusted function and place invocations of those macros in this list.

MK_CFG_NIRQS. The number of configured interrupt sources in the system. This may be greater than the number of ISRs.

MK_CFG_IRQLIST. A comma-separated list of interrupt source configurations as defined in Section 8.4.5, "Configuration of interrupt sources". You might find it helpful to define a macro for each interrupt source and place invocations of those macros in this list. If `MK_CFG_NIRQS` is zero, `MK_CFG_IRQLIST` is not used and you can omit its definition.

MK_CFG_NISRS. The number of AUTOSAR ISRs that are configured. The value includes all category 1 and 2 ISRs as well as the QM-OS ISRs for handling hardware counters.

MK_CFG_ISRLIST. A comma-separated list of ISR configurations as defined in Section 8.4.6, "Configuration of ISRs". You might find it helpful to define a macro for each ISR and place invocations of those macros in this list. The positions of the ISRs in this array must match the indices that are defined as the ISR identifiers in `Mk_gen_user.h`.

MK_SOFTVECTOR_nnnn. Each interrupt source has a hardware vector number. For each configured interrupt, a macro of this form connects the vector to the interrupt handling in the microkernel. If the interrupt source is configured as an AUTOSAR ISR, its `MK_SOFTVECTOR_nnnn` should be defined as `MK_VECTOR_ISR(i)` with `i` being the index of the ISR in the ISR table, i.e. the ISR identifier as defined in `Mk_gen_user.h`. If the interrupt source is handled directly by the microkernel its `MK_SOFTVECTOR_nnnn` should be defined as `MK_-VECTOR_INTERNAL(f,p)` where `f` is the name of the function that handles the interrupt and `p` is an integer parameter that gets passed to the function when the interrupt occurs.

MK_CFG_NTASKS. The total number of tasks that are configured.

MK_CFG_TASKLIST. A comma-separated list of task configurations as defined in Section 8.4.4, "Configuration of tasks". You might find it helpful to define a macro for each task and place invocations of those macros in this list. The positions of the tasks in this array must match the indices that are defined as the task identifiers in `Mk_gen_user.h`.

MK_CFG_NMEMORYPARTITIONS. The number of memory partitions in the memory partition table. If the value is zero or less the microkernel does not enable memory protection.

MK_CFG_NMEMORYREGIONS. The number of memory regions in the memory region table.

MK_CFG_NMEMORYREGIONMAPS. The number of mappings in the region map table.

MK_CFG_MEMORYPARTITIONCONFIG. A comma-separated list of memory partition descriptors. Each descriptor is represented by an invocation of the macro `MK_MEMORYPARTITION()` as described in Section 8.4.8, “Configuration of memory partitions”. The array contains memory partitions for all threads; hook function threads, idle threads, QM-OS threads etc. as well as for tasks and ISRs. The first partition in the array is the global partition that is permanently resident in the MPU. This partition specifies the microkernel's regions as well as the regions that are valid for all threads.

MK_CFG_MEMORYREGIONMAPCONFIG. A comma-separated list of the memory-partition to memory-region mappings. A memory partition is a contiguous block of these mappings. Each element of the memory region map table is defined by an invocation of the `MK_MEMORYREGIONMAP()` macro as defined in Section 8.4.9, “Configuration of memory region mappings”.

MK_CFG_MEMORYREGIONCONFIG. A comma-separated list of memory-region descriptors. Each element of the memory region table is a memory region descriptor defined by an invocation of the macro `MK_MEMORYREGION()`, the macro `MK_MR_INIT()`, the macro `MK_MR_STACK()`, or the macro `MK_MR_NOINIT()`.

MK_CFG_NLOCKS. The total number of locks used in the configured system.

MK_CFG_NGLOBALLOCKS. The number of global locks used in the configured system.

MK_CFG_LOCKLIST. A comma-separated list of lock configurations as defined in Section 8.4.11, “Configuration of locks”. You might find it helpful to define a macro for each lock and place invocations of those macros in this list.

8.4.2. Core-specific configuration parameters without default values

The configuration macros described in this section define the configuration of core-specific aspects of the system which must be defined for every configuration.

NOTE

Macros start with prefix `MK_CFG_Cx_`

All these macros start with the prefix `MK_CFG_Cx_` where `x` specifies the physical core index of the associated core (`x = 0, 1, ..., (MK_MAXCORES-1)`).

MK_CFG_Cx_KERNEL_STACK_NELEMENTS. The number of stack elements needed by the kernel stack on core `x`.

MK_CFG_Cx_NAPPLICATIONS. The number of *OS-Applications* on core `x`.

MK_CFG_Cx_NTASKS. The total number of tasks that are configured on core `x`.

MK_CFG_Cx_NETASKS. The number of EXTENDED tasks that are configured on core `x`.

MK_CFG_Cx_NTASKTHREADS. The number of threads that are required for the configured tasks on core `x`. This value determines the size of an array called `MK_cx_taskThreads` whose elements can be used for configuring the tasks.

MK_CFG_Cx_NTASKREGISTERS. The number of distinct register stores that are needed for the tasks and task threads on core `x`. This value determines the size of an array called `MK_cx_taskRegisters` whose elements can be used to configure the tasks.

MK_CFG_Cx_NISRS. The number of ISRs that are configured on core `x`.

MK_CFG_Cx_NISRTHREADS. The number of threads that are required for the configured ISRs on core `x`. This value determines the size of an array called `MK_cx_isrThreads` whose elements can be used to configure the ISRs.

MK_CFG_Cx_NISRREGISTERS. The number of register stores that are required for the configured ISRs on core `x`. This value determines the size of an array called `MK_cx_isrRegisters` whose elements can be used to configure the ISRs.

MK_CFG_Cx_NJOBQUEUES. The total number of job queues that are needed by the configured objects on core `x`.

MK_CFG_Cx_JOBQUEUEBUFLEN. The total number of elements in the `MK_cx_jobQueueBuffer[]` array. Sections of this array can be used by the job queues for their ring buffers.

MK_CFG_Cx_JOBQUEUECONFIG. A comma-separated list of job queue configurations on core `x`, each of which is an invocation of the macro `MK_JOBQUEUECFG` as described in Section 8.4.7, "Configuration of job queues".

MK_CFG_Cx_NMEMORYPARTITIONS. The number of memory partitions in the memory partition table on core `x`.

MK_CFG_Cx_INIT_MEMPART. The index of the memory partition to be used by the init thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_IDLE_MEMPART. The index of the memory partition to be used by the idle thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_QMOS_MEMPART. The index of the memory partition to be used by the QM-OS thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_ERRORHOOK_MEMPART. The index of the memory partition to be used by the error hook thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_PROTECTIONHOOK_MEMPART. The index of the memory partition to be used by the protection hook thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_SHUTDOWNHOOK_MEMPART. The index of the memory partition to be used by the shutdown hook thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_SHUTDOWN_MEMPART. The index of the memory partition to be used by the shutdown thread on core `x`. If no memory partition is required, the value is `-1`.

MK_CFG_Cx_FIRST_MEMORYPARTITION. The index of the first memory partition on core `x` within the global memory partition configuration.

MK_CFG_Cx_NLOCKS. The number of used locks on core `x`.

8.4.3. Core-specific configuration parameters with default values

The configuration macros described in this section define the configuration of core specific aspects of the system which can be defined for every configuration. If they are not defined, default values are used.

NOTE

Macros start with prefix `MK_CFG_Cx_`

All these macros start with the prefix `MK_CFG_Cx_` where `x` specifies the physical core index of the associated core (`x = 0, 1, ..., (MK_MAXCORES-1)`).

MK_CFG_Cx_MSG_INVALIDHANDLER. The handler which is called by the microkernel if an invalid inter-core request is identified on core `x`:

- `MK_CountInvalidXcoreMessage` (default): the error is ignored but it is counted in the microkernel field `MK_cx_coreVars.nDroppedMessages`.
- `MK_IgnoreInvalidXcoreMessage`: the error is ignored.
- `MK_PanicInvalidXcoreMessage`: the microkernel reports the panic `MK_panic_InvalidCrossCoreMessage`.

MK_CFG_Cx_IDLE_FUNCTION. These macros define the name of the function that runs in the idle thread on core `x`. The normal value is `MK_Idle` but you can use a user-defined function. If you use a user-defined function, you need to provide its prototype in `Mk_board.h`.

WARNING

Do not terminate the idle thread

The idle thread must never terminate.

MK_CFG_Cx_IDLE_MODE. The starting value of the processor mode for the idle thread on core `x`. Choose the value that should select the least-privileged processor mode unless the chosen idle function has other needs. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_IDLE_STACK_NELEMENTS. The number of stack elements needed by the idle thread on core `x`, rounded to take alignment into account. The same stack is used for the shutdown-idle thread, so it must be large enough for both purposes.

MK_CFG_Cx_INIT_FUNCTION. The name of the function that runs in the initialization thread on core `x`. This is the function that ultimately calls `StartOS` in a normal AUTOSAR system. Under normal circumstances the value should be `main()`. Since the prototype of `main()` is not defined for freestanding C, it needs to be provided in `Mk_board.h`.

WARNING **No support for function parameters**

The microkernel essentially treats the initialization thread function as a `void ()(void)` function. If the function you configure as `MK_CFG_Cx_INIT_FUNCTION` expects parameters, the values of these parameters are *undefined* when the function is invoked. Do not use these parameters.

MK_CFG_Cx_INIT_MODE. The starting value of the processor mode for the initialization thread on core `x`. Choose the value that should select the least-privileged processor mode unless the `MK_CFG_Cx_INIT_FUNCTION` function has other needs. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_INIT_QPRIO. The queueing priority of the initialization thread on core `x`.

MK_CFG_Cx_INIT_RPRIO. The running priority of the initialization thread on core `x`.

MK_CFG_Cx_INIT_STACK_NELEMENTS. The number of stack elements needed by the init thread on core `x`.

MK_CFG_Cx_SHUTDOWN_FUNCTION. The name of the function that runs in the idle thread on core `x` during shutdown. The normal value is `MK_Idle` but you can use a user-defined function. If you use a user-defined function, you must provide its prototype in `Mk_board.h`.

WARNING **Do not terminate the idle thread**

The idle thread must never terminate.

MK_CFG_Cx_SHUTDOWN_MODE. The starting value of the processor mode for the idle thread on core `x` during shutdown. Choose the value that should select the least-privileged processor mode unless the chosen shutdown-idle function has other needs. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_SHUTDOWN_STACK_NELEMENTS. The number of stack elements needed by the idle thread on core `x` during shutdown.

MK_CFG_Cx_HAS_ERRORHOOK. `1` if the error hook on core `x` is enabled, `0` if it is disabled. You can omit the following macros to configure the error hook thread if the `ErrorHook()` is disabled.

MK_CFG_Cx_ERRORHOOK_MODE. The starting value of the processor mode for the error hook thread on core `x`. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_ERRORHOOK_LEVEL. The interrupt lock level for the error hook thread on core `x`. You should disable all category 1 and 2 interrupts.

MK_CFG_Cx_ERRORHOOK_QPRIO. The queueing priority of the error hook thread on core `x`. The value should be lower than the queueing priority of the protection hook thread but higher than both priorities of any other thread.

MK_CFG_Cx_ERRORHOOK_RPRIO. The running priority of the error hook thread on core `x`. The value should be higher than both priorities of any other thread except the protection hook thread.

MK_CFG_Cx_ERRORHOOK_STACK_NELEMENTS. The number of stack elements required by your error hook function on core `x`.

MK_CFG_Cx_HAS_PROTECTIONHOOK. `1` if the protection hook on core `x` is enabled, `0` if it is disabled. You can omit the following macros to configure the protection hook thread if the protection hook is disabled.

MK_CFG_Cx_PROTECTIONHOOK_MODE. The starting value of the processor mode for the protection hook thread on core `x`. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_PROTECTIONHOOK_LEVEL. The interrupt lock level for the protection hook thread on core `x`. All category 1 and 2 interrupts should be disabled.

MK_CFG_Cx_PROTECTIONHOOK_QPRIO. The queueing priority of the protection hook thread on core `x`. The value should be higher than both priorities of any other thread.

MK_CFG_Cx_PROTECTIONHOOK_RPRIO. The running priority of the protection hook thread on core `x`. The value should be higher than both priorities of any other thread.

MK_CFG_Cx_PROTECTIONHOOK_STACK_NELEMENTS. The number of stack elements required by your protection hook function on core `x`.

MK_CFG_Cx_QMOS_MODE. The starting value of the processor mode for the QM-OS thread on core `x`. Choose the value that should select the least-privileged processor mode required by the QM-OS. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_QMOS_QPRIO. The queueing priority of the QM-OS thread on core `x`.

MK_CFG_Cx_QMOS_RPRIO. The running priority of the QM-OS thread on core `x`.

MK_CFG_Cx_QMOS_STACK_NELEMENTS. The number of stack elements needed by the QM-OS thread on core `x`.

MK_CFG_Cx_HAS_TRUSTEDFUNCTION. `1` if the trusted function thread on core `x` is enabled, `0` if it is disabled. You can omit the following macros to configure the trusted functions thread for core `x` if you do not configure any trusted function on that core.

MK_CFG_Cx_TF_QPRIO. The queueing priority of the trusted functions thread on core `x`.

MK_CFG_Cx_TF_RPRIO. The running priority of the trusted functions thread on core `x`.

MK_CFG_Cx_TF_STACK_NELEMENTS. The number of stack elements needed by the trusted functions thread on core `x`.

MK_CFG_Cx_HAS_SHUTDOWNHOOK. `1` if the shutdown hook is enabled, `0` if it is disabled. You can omit the following macros to configure the shutdown hook thread if the shutdown hook is disabled.

MK_CFG_Cx_SHUTDOWNHOOK_MODE. The starting value of the processor state for the shutdown hook thread on core `x`. For details on the processor-specific modes, see also Section 9.3, “Processor state mapping on TriCore processors”.

MK_CFG_Cx_SHUTDOWNHOOK_LEVEL. The interrupt lock level of the shutdown hook thread on core `x`. All category 1 and 2 interrupts should be disabled.

MK_CFG_Cx_SHUTDOWNHOOK_QPRIO. The queueing priority of the shutdown hook thread. The value should be at least as high as that of the QM-OS thread and higher than all ISR threads, but lower than the error hook and protection hook threads.

MK_CFG_Cx_SHUTDOWNHOOK_RPRIO. The running priority of the shutdown hook thread on core `x`. The value should be at least as high as that of the QM-OS thread and higher than all ISR threads, but lower than the error hook and protection hook threads.

MK_CFG_Cx_SHUTDOWNHOOK_STACK_NELEMENTS. The number of stack elements required by your shutdown hook function on core `x`.

8.4.4. Configuration of tasks

The macro `MK_CFG_TASKLIST` is a comma-separated list of invocations of either `MK_TASKCFG` or `MK_ETASKCFG`.

`MK_TASKCFG(dynamic, thread, registers, name, core, splimit, initialsp, function, pm, ilvl, ie, fpu, hws, queueprio, runprio, maxact, partIdx, taskid, exBgt, lkBgt, aid)` is used for BASIC tasks.

`MK_ETASKCFG(dynamic, thread, registers, name, core, splimit, initialsp, function, pm, ilvl, ie, fpu, hws, queueprio, runprio, evstat, partIdx, taskid, exBgt, lkBgt, aid)` is used for EXTENDED tasks.

Note the difference in the 16th parameter; for BASIC tasks it is the maximum number of activations and for EXTENDED tasks it is a pointer to the event status block to use.

The parameters are described in the following paragraphs.

dynamic. The address of the state variables for the task. Its value is usually `&MK_cx_taskDynamic[i]` for some value of `i` that is unique for the task and in the range `0 ≤ i < MK_CFG_Cx_NTASKS`.

thread. The address of the thread structure for the thread in which the task runs. Its value is usually `&MK_cx_taskThreads[j]` for some value of `j` in the range `0 ≤ j < MK_CFG_Cx_NTASKTHREADS`. A thread can be shared by several tasks subject to the conditions described in Section 7.4.2.3, “Allocating tasks to threads”.

registers. The address of the register store for this task. Its value is usually `&MK_cx_taskRegisters[k]` for some value of `k` in the range `0 ≤ k < MK_CFG_Cx_NTASKREGISTERS`. A register store can be shared by several tasks subject to the conditions described in Section 7.4.2.6, “Allocating register stores for task and ISR threads”.

name. The name of the task represented as a string, i.e. surrounded with double quotation marks.

core. The physical core index of the processor core on which the task runs.

splimit. The lower limit for the stack pointer for the task.

initialsp. The initial value of the stack pointer for the task. You must observe the alignment requirements of the processor and you must reserve enough space to store registers above the initial stack pointer if the processor's ABI demands it. The microkernel does not make any adjustment for processor requirements. You can use the first address above the configured stack on some processors, but you might prefer to leave some space unused.

function. The name of the outer level function for this task. If you defined your task functions using the `TASK(<taskname>)` construct, the name of the function is `OS_TASK_<taskname>`.

pm. The value of the processor mode when this task is started. Possible values are `MK_THRMODE_USER` (non-privileged mode; recommended) and `MK_THRMODE_SUPER` (privileged mode). Additional processor modes may be available, depending on the hardware. If so, they are described in the hardware-specific part of this document.

ilvl. The value written into the interrupt controller's lock-level register when this task is started. It should be `MK_HWENABLEALLLEVEL` unless there are special circumstances which require a higher lock-level.

ie. The value of the global interrupt enabled flag when this task is started. It should always be `MK_THRIRQ_ENABLE` to enable interrupts. If you really need to disable interrupts use `MK_THRIRQ_DISABLE`.

fpu. This value determines whether the task is allowed to use the floating point unit of the processor. Valid values are `MK_THRFPU_ENABLE` and `MK_THRFPU_DISABLE`. This option has no effect on processors that do not have a floating point unit.

hws. The value of the hardware-specific processor status extensions when this task is started. If not used, the value shall be `MK_THRHWS_DEFAULT`. Further possible values depend on the hardware used and if present are described in the hardware-specific part of this document.

queueprio. The priority at which the task queues. This is directly related to the configured priority of the task. The EB tresos Studio generator might compress the priorities into a continuous range which starts at `1`, but for hand-configured systems you can use the configured priority.

runprio. The priority that the task takes whenever its thread becomes the current thread. The value is at least as high as `queueprio` but can be higher.

maxact. The maximum number of simultaneous activations that the task can accept, whereas the minimum number is `1`. This parameter is only for BASIC tasks. For restrictions on threads and job queues for tasks with more than one activation, see Section 7.4.2.3, "Allocating tasks to threads".

evstat. A pointer to the event status structure for the task. This parameter is only for EXTENDED tasks.

partIdx. The index of the task's memory partition entry in the memory-partition array. A value of `-1` indicates that no memory partition is allocated, but it should be used only on systems with no memory protection.

taskid. The index of the task's entry in the `MK_TASKCONFIG` list. If you use the task's ID macro here, the microkernel can ensure that the ID matches the position in the list.

exBgt. The execution time budget for the task, specified in ticks of the execution timer. You can use the macro `MK_ExecutionNsToTicks()` here to specify the time in nanoseconds. You can use a value of `MK_EXECBUDGET_INFINITE` to configure a task without a limit.

lkBgt. A pointer to an array of lock budgets that contains one entry for each resource in the resource table. Each entry in the array is a time budget that is specified in the same way as the execution time budget. A value of `MK_NULL` for this parameter means that no lock budgets are applied to this task.

NOTE

No support for lock budgets

The microkernel does not implement lock budgets in the current version, so you can specify the value `MK_NULL` for all tasks.

aid. The index of the OS-Application to which this task belongs. Use a value of `MK_APPL_NONE` if the task does not belong to an OS-Application.

8.4.5. Configuration of interrupt sources

The macro `MK_CFG_IRQLIST` is a comma-separated list of invocations of `MK_IRQCFG(control_register, level, coreIndex, flags).`

The parameters are described in the following paragraphs.

control_register. The address of the interrupt controller register that controls the level, enable etc. for the interrupt request.

level. The hardware level for the interrupt request.

coreIndex. The physical index of the core which shall service the interrupt request.

flags. Extra configuration flags for the interrupt request. At the moment the only available flag is *enable-on-startup*, i.e. `MK_IRQ_ENABLE`. In addition to that, in multi-core environments the least significant bits of the flags contain the physical core index. The value `MK_IRQ_ENABLE|1` means for example that the interrupt request is assigned to core 1 and it shall be enabled on startup.

8.4.6. Configuration of ISRs

The macro `MK_CFG_ISRLIST` is a comma-separated list of invocations of `MK_ISRCFG(irqcfg, thread, registers, dynamic, name, core, splimit, initialsp, function, pm, ilvl, ie, fpu, hws, queueprio, runprio, partIdx, isrid, exBgt, lkBgt, aid)`.

The parameters are described in the following paragraphs.

irqcfg. The address of the corresponding interrupt source descriptor. The value is `&MK_irqCfgTable[i]`. `i` is the position in `MK_CFG_IRQLIST` of the interrupt request that is associated with this ISR.

thread. The address of the thread structure for the thread in which the ISR runs. Its value is usually `&MK_cx_isrThreads[j]` for some value of `j` in the range `0 ≤ j < MK_CFG_Cx_NISRTHREADS`. Several ISRs can share a thread subject to the conditions described in Section 7.4.2.4, “Allocating ISRs to threads”.

registers. The address of the register store for this ISR. Its value is usually `&MK_cx_isrRegisters[k]` for some value of `k` in the range `0 ≤ k < MK_CFG_Cx_NISRREGISTERS`. Several ISRs can share a register store subject to the conditions described in Section 7.4.2.6, “Allocating register stores for task and ISR threads”.

dynamic. The address of the state variables for this ISR. Its value is usually `MK_cx_isrDynamic[i]` for some value of `i` that is unique for the ISR and in the range `0 ≤ i < MK_CFG_Cx_NISRS`.

name. The name of the ISR represented as a string, i.e. surrounded with double quotation marks.

core. The physical core index of the processor core on which the ISR runs.

splimit. The lower limit for the stack pointer for the ISR.

initialsp. The initial value of the stack pointer for the ISR. You must observe the alignment requirements of the processor and you must reserve enough space to store registers above the initial stack pointer if the processor's ABI demands it. The microkernel does not make any adjustment for processor requirements. You can use the first address above the configured stack on some processors, but you might prefer to leave some space unused.

function. The name of the outer level function for this ISR. If you defined your ISR functions through the `ISR(<isrname>)` construct or the `ISR1(<isrname>)` construct, the name of the function is `OS_ISR_<isrname>`.

pm. The value of the processor mode when this ISR is started. Possible values are `MK_THRMODE_USER` (non-privileged mode; recommended) and `MK_THRMODE_SUPER` (privileged mode). Additional processor modes may be available, depending on the hardware. If so, they are described in the hardware-specific part of this document.

ilvl. The value written into the interrupt controller's lock-level register when the ISR is started. The value must be greater than or equal to the level configured in the corresponding IRQ descriptor. If the ISR is non-preemptive the value must be equal to the highest `ilvl` among the ISRs of the same category.

ie. The value of the global interrupt enabled flag when the ISR is started. It should always be `MK_THRIRQ_ENABLE` to enable interrupts. If you really need to disable interrupts, use `MK_THRIRQ_DISABLE`.

fpu. This value determines whether the ISR is allowed to use the floating point unit of the processor. Valid values are `MK_THRFPU_ENABLE` and `MK_THRFPU_DISABLE`. This option has no effect on processors that do not have a floating point unit.

hws. The value of the hardware-specific processor status extensions when the ISR is started. If not used, the value shall be `MK_THRHWS_DEFAULT`. Further possible values depend on the hardware used and if present are described in the hardware-specific part of this document.

queueprio. The priority at which the ISR queues. You shall choose the value so that ISRs with increasing IRQ levels are configured with increasing queue-priority.

runprio. The priority that the ISR gains when it first starts executing. The value is at least as high as `queueprio` but can be higher.

partIdx. The index of the ISR's memory partition entry in the memory-partition array. A value of `-1` indicates that no memory partition is allocated, but you should only use it on systems with no memory protection.

isrid. The index of the ISR's entry in the `MK_CFG_ISRLIST` list. If you use the ISR's ID macro here, the microkernel can ensure that the ID matches the position in the list.

exBgt. The execution time budget for the ISR, specified in ticks of the execution timer. You can use the macro `MK_ExecutionNsToTicks()` here to specify the time in nanoseconds. You can use a value of `MK_EXECBUDGET_INFINITE` to configure an ISR without a limit.

lkBgt. A pointer to an array of lock budgets that contains one entry for each resource in the resource table. Each entry in the array is a time budget that is specified in the same way as the execution time budget. A value of `MK_NULL` for this parameter means that no lock budgets are applied to this ISR.

NOTE

Implementation of lock budgets

The microkernel does not implement lock budgets in the current version, so you can specify the value `MK_NULL` for all ISRs.

aid. The index of the OS-Application to which this ISR belongs. Use a value of `MK_APPL_NONE` if the ISR does not belong to an OS-Application.

8.4.7. Configuration of job queues

You need to create a job queue for each task thread that is used by more than one task or a single task with multiple activations. The length of the job queue should be at least the sum of the activations of the tasks that use the thread, minus 1.

Each entry in the `MK_CFG_Cx_JOBQUEUECONFIG` list is an invocation of `MK_JOBQUEUECFG(thread, len, size)`.

thread. The address of the thread with which the job queue is associated. Its value is normally `&MK_cx_taskThreads[t]` for some value of `t`.

len. The length of the job queue, i.e. the number of jobs that can be queued.

size. The size of the job, i.e. the number of `mk_jobelement_t` elements in each job.

8.4.8. Configuration of memory partitions

A memory partition is a set of memory regions.

`MK_CFG_MEMORYPARTITIONCONFIG` is a comma-separated list of invocations of the macro `MK_MEMORYPARTITION(start, nregions)`. The parameters are described in the following paragraphs.

start. The index in the memory region-map array `MK_CFG_MEMORYREGIONMAPCONFIG` of the first memory region reference that belongs to the partition.

nregions. The number of consecutive memory region references that belong to the partition.

8.4.9. Configuration of memory region mappings

The memory region mapping table is a simple list of indices of or references to the memory regions defined by `MK_CFG_MEMORYREGIONCONFIG`. Each memory partition is a contiguous block of these region references. The use of references permits each memory region to be mapped to two or more partitions.

The relationship between memory regions and memory partitions is depicted in Figure 7.3, “Memory partitions and regions”.

`MK_MEMORYREGIONMAP(partition, region, permissions, select)` is a comma-separated list of region references. The parameters are described in the following paragraphs.

partition. The index of the memory partition to which the mapping belongs.

region. The index of the memory region in `MK_CFG_MEMORYREGIONCONFIG` that is mapped into the partition.

permissions. Access permissions (read, write, execute) for the configured region. For details on memory access permissions, see also Section 9.4.2, “Memory permissions on TriCore processors”.

select. Hardware-specific selection criteria. For details, see also Section 9.4.4, “Additional memory region flags for TriCore processors”.

8.4.10. Configuration of memory regions

`MK_CFG_MEMORYREGIONCONFIG` is a comma-separated list of memory regions. Each memory region is defined by an invocation of either `MK_MR_INIT`, `MK_MR_STACK` or `MK_MR_NOINIT`

`MK_MR_INIT(name, icore, hwextra)` is used for initialized regions, `MK_MR_STACK(name, icore, hwextra)` is used for stack regions and `MK_MR_NOINIT(name, hwextra)` is used for uninitialized regions. The parameters are described in the following paragraphs.

name. The name of the memory region. The name is used to generate the symbols for several memory addresses used to define the boundaries of the region: `MK_RSA_<name>`, `MK_RLA_<name>`, `MK_BSA_<name>`, and `MK_RDA_<name>`. These symbols should be declared as `extern mk_uint32_t MK_RSA_<name>` etc. in the file .

icore. The physical index of the core which initializes the region.

hwextra. Contains additional hardware-dependent flags for the region, see Section 9.4.4, “Additional memory region flags for TriCore processors”.

8.4.11. Configuration of locks

`MK_CFG_LOCKLIST` is a comma-separated list of invocations of `MK_LOCKCFG(priority, locklevel, nesting, spinlock, core, index)`. The parameters are described in the following paragraphs.

priority. The ceiling priority of the lock. It is calculated to be at least as high as the queueing priority of each task and ISR that uses the lock.

locklevel. The interrupt lock level of the lock. For locks that are used by ISRs the value must be appropriate to lock out all ISRs that use the lock, e.g. at least the highest hardware interrupt level of those ISRs if hardware priorities are ascending. You should choose the value to leave interrupts open for locks used exclusively by tasks. You can use the predefined macro `MK_HWENABLEALLLEVEL` for this purpose.

nesting. The maximum number of times that this resource can be acquired concurrently by the same thread. A maximum value of `32767` is supported. For normal AUTOSAR resources the value should be `1`.

spinlock. The hardware-specific configuration of the spinlock, which shall be used for this lock. Use the macro `MK_HwGetSpinlockCfg(id)` to configure a spinlock or `MK_NULL` otherwise.

core. The physical core index of the core, which uses this lock.

index. Index of the element in the processor core's lock table, which is used for this lock.

8.4.11.1. Special locks

You should add up to three special locks to the list, depending on the needs of the system.

RES_SCHEDULER. If `RES_SCHEDULER` is used in your system, you should include the lock `MK_LOCKCFG(MK_SCHEDULERPRIO, MK_HWENABLEALLLEVEL, 1U, MK_NULL, <core>, <index>)` for each core in the list of locks and define `RES_SCHEDULER` to be the index of this lock.

RESCAT1. If `SuspendAllInterrupts()` or `DisableAllInterrupts()` are used in your system, you should include the lock `MK_LOCKCFG(MK_CAT1LOCKPRIO, MK_CAT1LOCKLEVEL, <nnn>, MK_NULL, <core>, <index>)`, where <nnn> is a suitably-chosen value for the concurrent occupation limit for each core in the list of locks and define `MK_RESCAT1` to be the index of this lock.

NOTE

Omit this lock if there are no category 1 ISRs

If there are no category 1 ISRs in your system, you can omit this lock and define `MK_RESCAT1` to be the same as `MK_RESCAT2`.

RESCAT2. If `SuspendOSInterrupts()` is used in your system, you should include the lock `MK_LOCKCFG(MK_CAT2LOCKPRIO, MK_CAT2LOCKLEVEL, <nnn>, MK_NULL, <core>, <index>)`, where <nnn> is a suitably-chosen value for the concurrent occupation limit for each core in the list of locks and define `MK_RESCAT2` to be the index of this lock.

NOTE

Configuring locks

`RES_SCHEDULER`, `MK_RESCAT1` and `MK_RESCAT2` are global constants. Furthermore, each of those locks must be configured on each core using it. This means that the `<index>` parameter of the lock configurations must be equal to the corresponding constant on all cores so these locks must have the same positions in the core-specific lock configuration lists.

8.4.12. Configuration of trusted functions

`MK_CFG_TRUSTEDFUNCTIONLIST` is a comma-separated list of invocations of `MK_TRUSTEDFUNCTIONCFG(name, core, func, pm, fpu, hws, partIdx)`. The parameters are described in the following paragraphs.

name. The name of the trusted function as configured.

core. The physical core index of the core on which the trusted function runs.

func. A pointer to the trusted function itself.

pm. The value of the processor mode when this trusted function is started. Possible values are `MK_THRMODE_USER` (non-privileged mode; recommended) and `MK_THRMODE_SUPER` (privileged mode). Additional processor modes may be available, depending on the hardware. If so, they are described in the hardware-specific part of this document.

fpu. This value determines whether the trusted function is allowed to use the floating point unit of the processor. Valid values are `MK_THRFPU_ENABLE` and `MK_THRFPU_DISABLE`. This option has no effect on processors that do not have a floating point unit.

hws. The value of the hardware-specific processor status extensions when this function is started. If not used, the value shall be `MK_THRHWS_DEFAULT`. Further possible values depend on the hardware used and if present are described in the hardware-specific part of this document.

partIdx. The index of the trusted functions memory partition entry in the memory-partition array. A value of `-1` indicates that no memory partition is allocated, but it should be used only on systems with no memory protection.

8.4.13. Configuration of simple schedule tables

MK_CFG_NSSTS

`MK_CFG_NSSTS` defines the number of SSTs that you have configured.

Example:

```
#define MK_CFG_NSSTS     3
```

MK_CFG_SSTCOUNTERTABLE

`MK_CFG_SSTCOUNTERTABLE` is a comma-separated list of counter (and associated schedule table) configurations. There must be exactly `MK_CFG_NSSTS` elements in the list. Each element is an invocation of the macro `MK_SSTCOUNTERCONFIG()`.

The macro `MK_SSTCOUNTERCONFIG(ctr, mod, act, nact, tkr, rel, core)` defines a single counter and its associated simple schedule table. Its parameters are described in the following list.

`ctr`
: This parameter specifies the address of the management structure for this counter. A typical value is `&MK_cx_sstCounters[n]` for some `n` in the range `0` to `(MK_CFG_NSSTS-1)`.

`mod`
: This parameter specifies the counter modulus. It is equal to the duration of the schedule table, and is one greater than the *max. allowed value* of the counter. The value of `mod` must lie in the range `2` to `1073741824` = 2^{30}.

`act`
: This parameter specifies the address of an array of *expiry-actions* for the schedule table associated with this counter. A typical value is `&MK_sstActions[n]` for some `n`, where the elements `n..(n+nact-1)` of `MK_sstActions[]` contain the expiry actions for this schedule table.

`nact`
: This parameter specifies the number of expiry actions in the array based at `act`.

`tkr`
: This parameter specifies the id of the hardware-dependent ticker that is used to drive this counter. Valid values are hardware-dependent. A value of `-1` indicates that this counter is not driven by an interrupt.

`rel`
: This parameter specifies the reload interval of the hardware timer that drives the counter. It is a system-specific number of ticks and is typically derived from a number of nanoseconds by means of a *nanoseconds-to-ticks* macro defined in `Mk_board.h`.

`core`
: This parameter specifies the physical index of the core on which the counter runs. It is defined by the core assignment of the owning OS application of a SST.

The expiry actions for any given schedule table must be in ascending order of the counter value at which they occur. It is permitted to have two or more expiry actions with the same counter value in order to model AUTOSAR expiry points with multiple actions. The actions with the same counter value occur logically in the order in which they appear in the array.

An example for a configuration which contains a single schedule table is given below.

```
#define MK_CFG_SSTCOUNTERTABLE \
      MK_SSTCOUNTERCONFIG \
      (   &MK_c0_sstCounters[0],      /* Use the first (only) element of the state array */ \
          10u,                        /* Duration 10 */ \
          &MK_sstActions[0],          /* The portion of the expiry actions array */ \
          5,                          /* 5 expiry actions */ \
          4,                          /* Use channel 4 of the PIT */ \
          MK_NsToTicks_PIT(1000000), /* A PIT interrupt every millisecond */ \
          0                           /* Core id */ \
      )
```

MK_CFG_NSSTACTIONS

`MK_CFG_NSSTACTIONS` is the total number of expiry actions that you have configured for all simple schedule tables in the system.

Example:

```
#define MK_CFG_NSSTACTIONS     5
```

MK_CFG_SSTACTIONTABLE

`MK_CFG_SSTACTIONTABLE` is a comma-separated list of expiry actions. There must be exactly `MK_CFG_-NSSTACTIONS` elements in the list. Each element is an invocation of the macro `MK_SSTACTIONCONFIG()`.

The macro `MK_SSTACTIONCONFIG(ct, task, evt)` defines a single expiry action. Its parameters are described in the following list.

`ct`
: This parameter specifies the value of the counter at which the expiry action is to be performed. Its value is in the range `0` to the duration of the schedule table. Expiry actions with a count value equal to the duration occur at the same time as, but logically before, the expiry actions with a count value of `0` in the next round.

`task`
: This parameter specifies the task for which this action is to be performed. Its value must be a valid task id.

`act`
: This parameter specifies the set of events to set for the task. It contains a bit which represents each event to be set, and is usually specified by means of a bitwise-OR of the eventmask macros created by the OS generator. If this parameter is `0` (i.e., specifies no events to set), the expiry action is a task activation.

An example for a configuration which contains five expiry actions for a single schedule table is given below.

```
#define MK_CFG_SSTACTIONTABLE \
      MK_SSTACTIONCONFIG(0u,  Task10ms_At_0, 0u), /* activate task */ \
      MK_SSTACTIONCONFIG(2u,  Task5ms, Evt_A),    /* set event */ \
      MK_SSTACTIONCONFIG(3u,  Task10ms_At_3, Evt_B|Evt_C), /* set event */ \
      MK_SSTACTIONCONFIG(7u,  Task5ms, Evt_A),    /* set event */ \
      MK_SSTACTIONCONFIG(10u, Task10ms_At_10, 0u) /* activate task */
```

8.4.14. Configuration of add-ons

`MK_CFG_ADDONCONFIG` is a comma-separated list of pointers to the add-on descriptor structures of the used add-ons. These add-on descriptor structures must have the type `mk_addondescriptor_t`. For information about their descriptor structures, see the documentation of the corresponding add-on.

The positions of the add-on descriptor structures in this array must match the indices that are defined as the add-on identifiers in `Mk_gen_user.h`.

If you use EB tresos Studio to configure your system, you can find this definition in `Mk_gen_addons.h`.

8.5. Board-specific configuration

This section describes the configuration macros and functions that are defined in `Mk_board.h` and `Mk_board.c`.

The hardware environment in which the microcontroller is used often has external circuitry and components that affect the operation of the microkernel and the executable objects that it manages. A design goal of the microkernel and its configuration is that it should be possible to compile a pre-configured microkernel to run on any control unit or evaluation board that uses the microcontroller for which the microkernel is designed. To achieve this, you must configure external components that affect the microkernel and specify their characteristics.

This configuration is stored in files called `Mk_board.h` and `Mk_board.c`. The microkernel provides templates of these files and other files. You must, however, develop your own versions of these files that are adapted for the hardware environment in which the microkernel runs. The content of the files is used for safety-relevant functionality. So you must develop and verify the files with a process appropriate for the ASIL that is allocated to the microkernel.

8.5.1. Mk_board.h

This file provides the microkernel with the macros that it needs in order to convert between nanoseconds and ticks for the time stamp timer and the execution timer. The required macros are the following:

MK_TimestampNsToTicks(ns)
: A macro that converts `ns` nanoseconds to the equivalent number of ticks of the hardware timer from which the time stamp is derived. The compiler must be able to evaluate this macro at compile time.

MK_TimestampNsToTicksF(ns)
: A macro that converts `ns` nanoseconds to the equivalent number of ticks of the hardware timer from which the time stamp is derived. This macro can use run-time computation for greater accuracy.

MK_TimestampTicksToNs(tk)
: A macro that converts `tk` ticks of the hardware timer from which the time stamp is derived to the equivalent number of nanoseconds. The compiler must be able to evaluate this macro at compile time.

MK_TimestampTicksToNsF(tk)
: A macro that converts `tk` ticks of the hardware timer from which the time stamp is derived to the equivalent number of nanoseconds. This macro can use run-time computation for greater accuracy.

MK_ExecutionNsToTicks(ns)
: A macro that converts `ns` nanoseconds to the equivalent number of ticks of the hardware timer that is used for the execution budget. The compiler must be able to evaluate this macro at compile time.

You can define the above macros in terms of the appropriate set of timer conversion macros provided with the microkernel. Alternatively, if there is no macro available for your frequency, you can supply your own implementation following the pattern of the available macros.

In addition, the `Mk_board.h` file provides the following macros to configure the scaling factors for the `MK_-ElapsedTimeX` family of functions:

MK_TIMESTAMPCLOCKFACTOR100U
: This macro defines a factor to use to convert ticks of the time stamp timer into units of 100 microseconds. It is used by the function `MK_ElapsedTime100u`. The value must be an unsigned integer. Example: If the timer frequency is 80 MHz, a scaling factor of 8000 is needed.

MK_TIMESTAMPCLOCKFACTOR10U
: This macro defines a factor to use to convert ticks of the time stamp timer into units of 10 microseconds. It is used by the function `MK_ElapsedTime10u`. The value must be an unsigned integer. Example: If the timer frequency is 80 MHz, a scaling factor of 800 is needed.

MK_TIMESTAMPCLOCKFACTOR1U
: This macro defines a factor to use to convert ticks of the time stamp timer into units of 1 microseconds. It is used by the function `MK_ElapsedTime1u`. The value must be an unsigned integer. Example: If the timer frequency is 80 MHz, a scaling factor of 80 is needed.

To enable the use of the respective conversion factors, you should define the macros `MK_HAS_TIMESTAMPCLOCKFACTOR100U`, `MK_HAS_TIMESTAMPCLOCKFACTOR10U`, and `MK_HAS_-TIMESTAMPCLOCKFACTOR1U` as 1.

8.5.2. Mk_board.c

The microkernel does not place any requirements on what functions should be in `Mk_board.c`. However, the microkernel requires that the two callout functions `MK_InitHardwareBeforeData` and `MK_InitHard-wareAfterData` are defined. The template file provided with the microkernel defines these functions, and they are described here.

8.5.2.1. MK_InitHardwareBeforeData

The microkernel's startup code calls this function *before* the memory regions are initialized. At this time, the processor is running in the highest privileged level. Memory protection may be active, but configured such that the processor's access to the available memory and I/O spaces is not restricted. Interrupts are disabled.

Initialization of the memory regions can take considerable time, so you should use this function to configure the processor's clock to the normal running speed and to configure and enable the caches.

In many cases, the processor clock and caches are initialized before control is transferred to the microkernel. If this is the case, you can omit the initialization here.

The function must not assume that variables already have their initial values. Furthermore, information written by this function into statically-allocated variables is erased when the memory regions are initialized, unless the variables are placed in regions for which initialization is disabled.

`MK_InitHardwareBeforeData()` must not make any system calls to the microkernel.

8.5.2.2. MK_InitHardwareAfterData

The microkernel's startup code calls this function *after* the memory regions are initialized. At this time, the processor runs in the highest privilege level. Memory protection may be active, but configured such that the processor's access to the available memory and I/O spaces is not restricted. Interrupts are disabled.

You can use this function to configure hardware features such as access to peripherals when the processor is running in a non-privileged mode.

If processor features such as the clock systems and caches are initialized by software that runs before control is transferred to the microkernel, you should ensure that the settings are correct before you proceed. Run the microkernel-based system in its normal operating mode only if safety-relevant parts of the hardware are correctly initialized.

`MK_InitHardwareAfterData()` must not make any system calls to the microkernel.

8.6. Deviations from the AUTOSAR standard

This section lists the deviations from Specification of Operating System [AUTOSAROS42SPEC].

8.6.1. OS-Applications

The microkernel provides only partial management of OS-Applications. This results in the following deviations:

Deviation.Autosar.ObjectAccess
The `CheckObjectAccess()` service is not implemented and object access rights are not enforced. Deviates: *Autosar.SWS_Os_00271*, *Autosar.SWS_Os_00056*, *Autosar.SWS_Os_00256*, *Autosar.SWS_Os_00272*, *Autosar.SWS_Os_00423*, *Autosar.SWS_Os_00448*, *Autosar.SWS_Os_00450*, *Autosar.SWS_Os_00519*, *Autosar.SWS_Os_00692*, *Autosar.SWS_Os_00700*, *Autosar.SWS_Os_00710*

Deviation.Autosar.ObjectOwnership
The `CheckObjectOwnership` service is not implemented. Deviates: *Autosar.SWS_Os_00017*, *Autosar.SWS_Os_00273*, *Autosar.SWS_Os_00274*, *Autosar.SWS_Os_00520*

Deviation.Autosar.AppHooks
Application-specific `StartupHook()`, `ShutdownHook()` and `ErrorHook()` functions are not called. Deviates: *Autosar.SWS_Os_00246*, *Autosar.SWS_Os_00060*, *Autosar.SWS_Os_00085*, *Autosar.SWS_Os_00112*, *Autosar.SWS_Os_00225*, *Autosar.SWS_Os_00226*, *Autosar.SWS_Os_00236*, *Autosar.SWS_Os_00237*, *Autosar.SWS_Os_00539*, *Autosar.SWS_Os_00543*, *Autosar.SWS_Os_00540*, *Autosar.SWS_Os_00544*, *Autosar.SWS_Os_00541*, *Autosar.SWS_Os_00545*, *Autosar.SWS_Os_00586*

Deviation.Autosar.RestartApplication
If the `ProtectionHook()` returns `PRO_TERMINATEAPPL_RESTART` and no OsRestartTask was configured for the faulty OS-Application, the microkernel terminates the application. The microkernel does not shut down. Deviates: *Autosar.SWS_Os_00557*

8.6.2. Absolute trust

The microkernel does not have any concept of absolute trust. All threads, regardless of their processor mode privileges, run with memory protection enabled and access restricted to configured regions. The microkernel too runs with its access restricted to configured regions. This property of the microkernel means that trusted functions as defined by AUTOSAR cannot be implemented, and results in the following deviations:

Deviation.Autosar.NonTrustedThreads
The microkernel runs all threads with memory protection enabled. Deviates: *Autosar.SWS_Os_00209*, *Autosar.SWS_Os_00211*

Deviation.Autosar.TrustedFunctionMode
The microkernel executes trusted functions in threads. Memory protection settings as well as processor mode can be configured individually per trusted function. Deviates: *Autosar.SWS_Os_00266*

Deviation.Autosar.CheckMemoryAccess
The `CheckTaskMemoryAccess()` and `CheckISRMemoryAccess()` services are not implemented. Deviates: *Autosar.SWS_Os_00267*, *Autosar.SWS_Os_00268*, *Autosar.SWS_Os_00269*, *Autosar.SWS_Os_00270*, *Autosar.SWS_Os_00313*, *Autosar.SWS_Os_00314*, *Autosar.SWS_Os_00449*, *Autosar.SWS_Os_00512*, *Autosar.SWS_Os_00517*, *Autosar.SWS_Os_00513*, *Autosar.SWS_Os_00518*

Deviation.Autosar.IllegalAddress
The error code `E_OS_ILLEGAL_ADDRESS` is never returned. The microkernel cannot normally write to the out-parameters specified by the caller. Information is returned by placing it in registers and copying the information to the out-parameter in a library function where the memory protection boundaries of the caller are active. An *illegal address* therefore results in a memory protection fault that is attributed to the calling thread. Deviates: *Autosar.SWS_Os_00051*, *Autosar.SWS_Os_00566*

Deviation.Autosar.IgnoreShutdown
The microkernel does not determine the right to shutdown a system from the processor mode. Instead, the configuration explicitly states whether an OS-Application is allowed to use `ShutdownOS()` or `ShutdownAllCores()`. Deviates: *Autosar.SWS_Os_00054*, *Autosar.SWS_Os_00716*

For details on access rights, see also Section 6.10, “API access protection”.

Deviation.Autosar.IgnoreTerminateApplication
The microkernel does not determine the right to terminate an OS-Application from the processor mode. Instead, the configuration explicitly states, for each OS-Application which other OS-Applications are allowed to terminate it. Deviates: *Autosar.SWS_Os_00494*, *Autosar.SWS_Os_00535*

Deviation.Autosar.TrustedFunctionTimingProtection
A trusted function does not inherit the timing protection characteristics of its calling thread. Hence, if a trusted function is called from a thread with timing protection enabled, this protection does not extend to the called trusted function. Rather, there are dedicated threads for trusted functions, for which you can configure timing protection separately.
Deviates: *Autosar.SWS_Os_00312*

Deviation.Autosar.AlarmCallbacks
The microkernel runs alarm callbacks inside the QM-OS thread. Therefore the memory protection settings for the QM-OS thread applies for alarm callbacks.
Deviates: *Autosar.SWS_Os_00242*

8.6.3. Calling context restrictions

The microkernel does not enforce the AUTOSAR calling context restrictions unless a restriction is necessary to ensure that the microkernel operates correctly. The only restrictions are the following:

- `WaitEvent()` and `ClearEvent()` may only be called from EXTENDED tasks because the data structures needed for event handling are only configured for EXTENDED tasks.
- Services implemented by the QM-OS part may not be called from the `ProtectionHook()` because a QM-OS thread cannot have a higher or equal priority than the `ProtectionHook()`.

The lack of calling context restrictions results in the following deviations:

Deviation.Autosar.TerminateThread
The services `TerminateTask()` and `ChainTask()` can be called from interrupt service routines, hook functions and other threads. In all cases the thread is terminated and no error relating to the context is reported.
Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, section 12.2.2.*
Deviates: *OSEK.API.TaskManagement.ChainTask.API*, *OSEK.API.TaskManagement.TerminateTask.API*

Deviation.Autosar.CallingContext
With the exception of the event services and requests to the QM-OS listed above, any service can be called from any thread. The behavior of the service is always defined. If no error is reported to the `ErrorHook()` and (where possible) by the return value, the service performs the defined action.
Deviates: *Autosar.SWS_Os_00088*

Deviation.Autosar.GetEvent
`GetEvent()` returns the pending events for a task even after the task is terminated and is in the SUSPENDED state. The main use for `GetEvent()` is for a task to determine which event or events cause it to be released from the WAITING state. In this case, the test for a SUSPENDED task would be redundant.
Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, section 13.5.3.3.*
Deviates: *OSEK.API.EventControl.GetEvent.SuspendedTask*

8.6.4. Interrupt and resource locking services

The interrupt locking services (`SuspendOSInterrupts()`, etc.) are implemented using resources. The resources are configured just like standard resources that are shared with the highest priority ISR of the corresponding category. This means that calling a service with interrupts locked is equivalent to calling that service while occupying a resource of the same priority. It also means that the interrupt locks have proper ceiling priorities that inhibit other tasks.

No restriction is placed on the services that can be used while occupying a resource, including an interrupt-locking resource.

The implementation and lack of restrictions results in the following deviations:

Deviation.Autosar.Schedule
If the `Schedule()` service is called while a resource or interrupt lock is occupied, only tasks with priority higher than the ceiling priority of the occupied resources are permitted to run.
Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, section 13.2.3.4.*
Deviates: *OSEK.API.TaskManagement.Schedule.Resources*

Deviation.Autosar.CallWhileOccupied
If TerminateThread(), ChainTask(), StartOS(), WaitEvent() or MK_WaitGetClearEvent() is called while the thread occupies a resource, the resource is released and no error is reported.
Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, sections 13.2.3.2, 13.2.3.3, 13.5.3.4.*
Deviates: *OSEK.API.TaskManagement.TerminateTask.ExternalResource*, *OSEK.API.TaskManagement.ChainTask.ExternalResource*, *OSEK.API.EventControl.WaitEvent.ExternalResource*

Deviation.Autosar.CallWhileLocked
If a service is called while the thread occupies an interrupt lock, the behavior is the same as if the call is made while occupying a resource.
Deviates: *Autosar.SWS_Os_00093*

Deviation.Autosar.CallWithSpinlock
If a service is called while the thread occupies a spinlock, the behavior is the same as if the call is made while occupying a resource of the same level as that of the spinlock. Deviates: *Autosar.SWS_Os_00622*, *Autosar.SWS_Os_00624*

Deviation.Autosar.LockAPI
The microkernel does not implement separate services for ▶ resources, ▶ interrupt locks and ▶ spinlocks. Instead, these services are handled by the common lock API. This implies that the spinlock functions can be called with resource IDs and that that the resource functions can be called with spinlock IDs. The functions will then behave like their counterpart. Deviates: *OSEK.API.ResourceManagement.GetResource.Extended*, *OSEK.API.ResourceManagement.ReleaseResource.Extended*, *Autosar.SWS_Os_00707*, *Autosar.SWS_Os_00698*, *Autosar.SWS_Os_00689*

Deviation.Autosar.ResourceAccess
The microkernel does not enforce a ceiling priority check when a thread attempts to acquire or release a resource. The acquisition fails if the resource is already occupied, but succeeds otherwise. The release fails if the resource is not successfully acquired by the calling thread. Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, sections 13.4.3.1, 13.4.3.2.* Deviates: *OSEK.API.ResourceManagement.GetResource.LowCeilingPriority*, *OSEK.API.ResourceManagement.ReleaseResource.LowCeilingPriority*

NOTE

Blocking of lower priority tasks

If a thread calls a service while it occupies a resource or interrupt lock and that service activates a task of lower priority than the resource, the task does not run until the resource is released.

NOTE

Interrupt locking resources are nestable

The resources that the generator creates for interrupt locking are *nestable resources*, which means they can be acquired multiple times by the same thread. When releasing such resources, only the release that matches the first acquisition has the effect of releasing the resource.

Deviation.ReleaseSpinLock.NotHeld
If you try to release a spinlock that is not occupied by the calling task, the microkernel returns `E_OS_NOFUNC` instead of `E_OS_STATE`. Deviates: *Autosar.SWS_Os_00699*

8.6.5. Thread termination

Deviation.Autosar.UnconditionalTermination
The `TerminateTask()`, `ChainTask()` and `StartOS()` services terminate the calling thread unconditionally. In the case of `ChainTask()`, termination happens regardless of whether the activation part of the service is successful. Resources, interrupt locks and spinlocks are automatically freed. No error that results from the termination is reported. Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, sections 13.2.3.2, 13.2.3.3.* Deviates: *OSEK.API.TaskManagement.ChainTask.Schedule*, *OSEK.API.TaskManagement.ChainTask.ExternalResource*, *OSEK.API.TaskManagement.ChainTask.ISRC2*, *OSEK.API.TaskManagement.ChainTask.InvalidTask*, *OSEK.API.TaskManagement.ChainTask.TooManyActivations*, *OSEK.API.TaskManagement.TerminateTask.Schedule*, *OSEK.API.TaskManagement.TerminateTask.ExternalResource*, *OSEK.API.TaskManagement.TerminateTask.ISRC2*, *Autosar.SWS_Os_00612*

Deviation.Autosar.TaskReturn
If a task returns from its outer-level function without calling `TerminateTask()` the microkernel terminates the task and releases all the resources, interrupt locks and spinlocks that it occupied. The termination is unconditional with no errors being reported. Deviates: *Autosar.SWS_Os_00069*, *Autosar.SWS_Os_00368*, *Autosar.SWS_Os_00369*

NOTE

Task activation error reporting

If the task activation part of `ChainTask()` reports an error, the `ErrorHook()` gets activated.

8.6.6. OS control

The microkernel is active before the `main()` is called and remains active even after the system shuts down. All executables outside the microkernel run in threads. This results in the following deviations:

Deviation.Autosar.MainThread
The `main()` function runs in a thread, which means that the thread scheduler is already active before `StartOS()` is called and therefore before the `StartupHook()` is invoked. Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, sections 13.8.2.4.* Deviates: *OSEK.API.Hooks.StartupHook.Startup*, *Autosar.SWS_Os_00608*

Deviation.Autosar.Shutdown
The `ShutdownOS()` service terminates the calling thread unconditionally. If it is successful, it also terminates all other threads including the idle thread, activates a shutdown-idle thread and a `ShutdownHook()` thread if the `ShutdownHook()` is configured. This means that the `ShutdownHook()` can continue to call microkernel services; the system does only shut down when there are no more threads eligible for the CPU and the shutdown-idle thread gains the CPU. The shutdown-idle thread contains an endless loop and runs with interrupts disabled, so no further thread switching takes place. Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, sections 13.7.2.3, 13.8.2.5.* Deviates: *OSEK.API.OSControl.ShutdownOS.Shutdown*, *OSEK.API.Hooks.ShutdownHook.Shutdown*, *Autosar.SWS_Os_00618*

Deviation.Autosar.Shutdown.Unsynchronized
The microkernel does not synchronize during shutdown. Deviates: *Autosar.SWS_Os_00587*

Deviation.Autosar.TaskHooks
The microkernel does not call the `PreTaskHook()` or the `PostTaskHook()` because those functions cannot be executed in threads. Comment: *See OSEK/VDX:Operating System Specification, Version 2.2.3, sections 13.8.2.2, 13.8.2.3.* Deviates: *OSEK.API.Hooks*, *OSEK.API.Hooks.PreTaskHook*, *OSEK.API.Hooks.PreTaskHook.API*, *OSEK.API.Hooks.PreTaskHook.TaskChange*, *OSEK.API.Hooks.PostTaskHook*, *OSEK.API.Hooks.PostTaskHook.API*

Deviation.Autosar.ErrorHook
The microkernel always invokes the `ErrorHook()` if an error is detected, regardless of whether the service returns a status indication. Failure to report errors is considered a safety risk; errors in application code could remain undetected. This could be particularly serious if a critical section is not correctly locked because of such an error. Deviates: *Autosar.SWS_Os_00367*, *Autosar.SWS_Os_00681*

Deviation.Autosar.ErrorHook.Core
The microkernel invokes the `ErrorHook()` always on the core on which the error is detected. Deviates: *Autosar.SWS_Os_00599*, *Autosar.SWS_Os_00605*

Deviation.Autosar.StartCore.After.StartOS
It is not possible to start fewer than the configured number of cores because the synchronization in `StartOS()` will wait until all cores reach the synchronization point. If a core has not been started, `StartOS()` waits forever. The microkernel does not keep track if `StartOS()` was called. Therefore, a call to `StartCore()` after `StartOS()` yields the error code `E_OS_STATE` instead of `E_OS_ACCESS`. Deviates: *Autosar.SWS_Os_00678*, *Autosar.SWS_Os_00606*

Deviation.Autosar.StartCore.SingleCore
On single-core processors StartCore will always report E_OS_NOFUNC. This service can't succeed on a single-core processor. It doesn't determine whether the status code should be `E_OS_ID`, `E_OS_ACCESS` or `E_OS_STATE`. Deviates: *Autosar.SWS_Os_00678*, *Autosar.SWS_Os_00606*, *Autosar.SWS_Os_00676*

Deviation.Autosar.Panic
In cases where the microkernel detects a fatal internal error the core on which the error is detected is shut down. Deviates: *Autosar.SWS_Os_00762*

8.6.7. Timing protection

The microkernel implements execution budget monitoring but does not implement lock budgets and arrival rate monitoring.

Deviation.Autosar.ExecutionBudget
Resource and interrupt lock budgets are not enforced. Deviates: *Autosar.SWS_Os_00033*, *Autosar.SWS_Os_00037*

Deviation.Autosar.ArrivalRate
The microkernel does not monitor arrival rates.
Deviates: *Autosar.SWS_Os_00028*, *Autosar.SWS_Os_00048*, *Autosar.SWS_Os_00089*, *Autosar.SWS_-Os_00397*, *Autosar.SWS_Os_00465*, *Autosar.SWS_Os_00466*, *Autosar.SWS_Os_00467*, *Autosar.SWS_-Os_00469*, *Autosar.SWS_Os_00470*, *Autosar.SWS_Os_00471*, *Autosar.SWS_Os_00472*

Deviation.Autosar.ExecutionBudget.CallTrustedFunction
The microkernel stops the execution budget for the caller while a trusted function is executed.
Deviates: *Autosar.SWS_Os_00564*, *Autosar.SWS_Os_00565*

8.6.8. Stack monitoring

The microkernel relies on memory protection to prevent stack overflows and underflows.

Deviation.Autosar.StackMonitoring
The microkernel does not monitor stack use.
Comment: *Stack protection is implemented via the memory protection unit.*
Deviates: *Autosar.SWS_Os_00067*, *Autosar.SWS_Os_00068*, *Autosar.SWS_Os_00396*

8.6.9. AUTOSAR API functions

The microkernel does not provide all AUTOSAR API functions.

Deviation.Autosar.API.StartNonAutosarCore
The microkernel does not provide the API `StartNonAutosarCore()`.
Deviates: *Autosar.SWS_Os_00584*, *Autosar.SWS_Os_00585*, *Autosar.SWS_Os_00680*, *Autosar.SWS_-Os_00682*, *Autosar.SWS_Os_00683*, *Autosar.SWS_Os_00684*, *Autosar.SWS_Os_00685*

Deviation.Autosar.API.ControlIdle
The microkernel does not provide the API `ControlIdle()`.
Deviates: *Autosar.SWS_Os_00769*, *Autosar.SWS_Os_00770*, *Autosar.SWS_Os_00771*, *Autosar.SWS_-Os_00802*

Deviation.Autosar.API.GetNumberOfActivatedCores
The microkernel does not provide the API `GetNumberOfActivatedCores()`.
Deviates: *Autosar.SWS_Os_00626*, *Autosar.SWS_Os_00672*, *Autosar.SWS_Os_00673*

Deviation.Autosar.API.IdleModeType
The microkernel does not provide the type `IdleModeType`. Deviates: *Autosar.SWS_Os_00793*

Deviation.Autosar.API.CallTrustedFunction
The microkernel's function `CallTrustedFunction` returns `E_OS_CALLLEVEL` instead of `E_OS_ACCESS` if the target trusted function is part of an OS-Application on another core. Deviates: *Autosar.SWS_Os_00623*

Deviation.Autosar.API.E_OS_CORE
The microkernel does not return the error code `E_OS_CORE`. Instead the microkernel returns the following error codes if functions that are not allowed to operate cross core are called with parameters that require a cross core operation: ▶ `GetResource():E_OS_ID` ▶ `ReleaseResource():E_OS_ID` ▶ `CallTrustedFunction():E_OS_CALLLEVEL` Deviates: *Autosar.SWS_Os_00589*

Deviation.Autosar.API.IncrementCounter
The QM function `IncrementCounter()` allows to increment a counter that is assigned to a different core than the caller. Deviates: *Autosar.SWS_Os_00629*

Deviation.Autosar.API.CallTrustedFunction
The microkernel schedules trusted function threads at the same priority and interrupt level as the caller of the trusted function. Should tasks or ISRs of the caller's OS-application become eligible to run then the microkernel schedules them according to their priorities. Deviates: *Autosar.SWS_Os_00563*

Deviation.Autosar.API.TerminateApplication
If the state of an OS-Application in a call of `TerminateApplication()` is `APPLICATION_RESTARTING` and the caller does belong to the OS-Application and the RestartOption is equal `RESTART` then `TerminateApplication()` terminates the application and returns `E_OK`. Deviates: *Autosar.SWS_Os_00548*

8.6.10. Spinlocks

The microkernel does not implement spinlocks according to AUTOSAR.

Deviation.Autosar.Spinlock.Order
The microkernel does not support spinlock orders which define sequences, in which multiple spinlocks must be acquired. The microkernel does not check if the spinlock acquisition order is correct or allowed at all, if multiple spinlocks are acquired. Deviates: *Autosar.SWS_Os_00660*, *Autosar.SWS_Os_00661*, *Autosar.SWS_Os_00686*, *Autosar.SWS_Os_00691*, *Autosar.SWS_Os_00709*

Deviation.Autosar.Spinlock.InterferenceDeadlock
`GetSpinlock()` and `TryToGetSpinlock()` do not return `E_OS_INTERFERENCE_DEADLOCK` if the service is called for a spinlock which is already occupied by another task on the same core. Instead, they return `E_OS_ACCESS`. Deviates: *Autosar.SWS_Os_00686*, *Autosar.SWS_Os_00690*, *Autosar.SWS_Os_00708*

Deviation.Autosar.Spinlock.SingleCore
`GetSpinlock()`, `TryToGetSpinlock()` and `ReleaseSpinlock()` are not available on single-core processors. Deviates: *Autosar.SWS_Os_00686*, *Autosar.SWS_Os_00695*, *Autosar.SWS_Os_00703*

8.6.11. Service interfaces

Deviation.Autosar.ServiceInterface.ClientServerInterface
The EB tresos Safety OS provides the client-server interface according to the 4.0.3 release of the Autosar specification. Deviates: *Autosar.SWS_Os_00560*, *Autosar.SWS_Os_00794*

8.7. Limitations

EB tresos Safety OS does not implement the following features of EB tresos AutoCore OS:

- CPU load measurement (functions `GetCpuLoad()`, `GetMaxCpuLoad()` and `InitCpuLoad()`)
- ISR hooks (`PreIsrHook()` and `PostIsrHook()`)

9. Supplementary information for TriCore processors

9.1. Startup conditions for TriCore processors

Startup preconditions are described in detail in the EB tresos Safety OS safety manual for TriCore family [SAFETYMANTRICORE].

9.2. Additional configuration of startup on TriCore processors

The following TriCore-specific configuration macros need to be provided:

- **MK_INITIAL_A0.** The initial value for the global address register A0. The microkernel initializes this register with this value but it does not use it. Therefore set an appropriate value for your application.
- **MK_INITIAL_A1.** The initial value for the global address register A1. The microkernel initializes this register with this value but it does not use it. Therefore set an appropriate value for your application.
- **MK_INITIAL_A8.** The initial value for the global address register A8. The microkernel initializes this register with this value but it does not use it currently. Therefore set an appropriate value for your application.

NOTE

Global register A8 may be restricted

In future versions the register A8 might be restricted for microkernel use.

9.3. Processor state mapping on TriCore processors

The processor state of TriCore processors is represented in the configuration structures by the following data type:

```
typedef struct mk_hwps_s mk_hwps_t;

struct mk_hwps_s
{
    mk_uint32_t psw;
    mk_uint32_t pcxi;
};
```

The `psw` member specifies the initial value of the program status word (PSW), the `pcxi` member specifies the interrupt lock level for the respective thread, represented as part of the `pcxi` processor register where only the previous CPU priority number (`PCPN`) and previous interrupt-enable (`PIE`) fields are used. The microkernel creates instances of this structure using the macro `MK_HWPS_INIT(pm, ilvl, ie, fpu, hwps)`. The parameters affect the struct members as follows:

`pm` - processor mode
: The values `MK_THRMODE_USER`, `MK_THRMODE_USER1` and `MK_THRMODE_SUPER` respectively cause the bits `U0`, `U1` and `S` in the `psw` to be set. This parameter selects the processor mode for a thread.

`ilvl` - interrupt level
: This value is stored in the `PCPN` field of the `pcxi` member. It selects the interrupt level that a thread executes with.

`ie` - interrupt enable
: This value initializes the PIE bit of the `pcxi` member. The value `MK_THRIRQ_ENABLE` causes the PIE bit in the PSW to be set, `MK_THRIRQ_DISABLE` causes it to be cleared. This parameter selects whether a thread is executed with interrupts globally disabled, or not.

`fpu` - use floating point unit
: This parameter is ignored on TriCore processors.

`hwps` - hardware-specific extensions
: This parameter is used to initialize the call-depth counting feature for the thread and to select the protection register set (PRS) used by the thread. You should use one of the following values:

  - `MK_THRHWS_DEFAULT`: Selects protection register set 1 and disables the call-depth counting feature. This is the value which is used for most threads, that use the dynamic memory partition.
  - `MK_THRHWS_PRS<x>`: Selects protection register set `<x>` and disables the call-depth counting feature. This value is used for threads that use the first fast partition. For more information, see Section 9.4.1, “Memory protection basics and fast partitions on TriCore processors”.

These parameters are part of e.g. task and ISR configurations, see Section 8.4.4, “Configuration of tasks” and Section 8.4.6, “Configuration of ISRs”. Additional configuration criteria for these parameters are also given in the EB tresos Safety OS safety manual for TriCore family [SAFETYMANTRICORE].

9.4. Memory protection on TriCore processors

9.4.1. Memory protection basics and fast partitions on TriCore processors

The MPU on TriCore supports several so-called protection register sets, which comprise a subset of the memory region descriptors. The concept is very similar to the memory partitions concept of the microkernel. For more information, see Section 8.4.8, “Configuration of memory partitions”. Assigning a dedicated protection register set (PRS) to a thread allows to dispatch this thread more efficiently than threads that share a PRS.

9.4.1.1. Static partitions, dynamic partitions, static regions, and dynamic regions

To understand the concept and configuration of fast partitions, we need to first introduce the concepts of *static partitions*, *dynamic partitions*, *static regions*, and *dynamic regions*:

- A static partition is a memory partition whose regions are programmed into the MPU during startup time and remain resident until the next reset.
- A dynamic partition on the other hand contains regions that are programmed to the MPU as a thread to that the dynamic partition was assigned is dispatched. As soon as the next thread that is assigned a dynamic partition is dispatched, the regions of the previous dynamic partition are replaced in the MPU.
- Static regions are memory regions that belong to a static partition and are permanently resident in the MPU.
- Dynamic regions are regions that are only resident in the MPU until a thread with a different dynamic partition than the current one is dispatched.

Static partitions only contain static regions. A static region is defined by being part of a static partition, but dynamic regions can contain both static and dynamic regions.

9.4.1.2. Fast partition basics

The memory regions that a thread has access to are defined by the active PRS and the regions assigned to that PRS. When the microkernel dispatches a thread, the regions which belong to the thread's partition must be programmed to the MPU and assigned to the PRS used for the thread. Static regions are always resident in the MPU, so only the dynamic regions must be programmed to the MPU during dispatch. With the concept of fast partitions, unused protection register sets are made available for use by application threads, so the assignment to regions remains also resident in the MPU. This means that no costs incur in the microkernel to set up the MPU if it dispatches a thread that is assigned to a fast partition.

All currently known TC1.6.1 derivatives support four protection register sets and all currently known TC1.6.2 derivatives support six of them. The microkernel requires one PRS that is dynamically reprogrammed at run-time as threads with dynamic memory partitions are dispatched. Another PRS is used for the microkernel. This leaves two (TC1.6.1) or four (TC1.6.2) protection register sets for the fast partitions feature, limiting the number of fast partitions that can be defined. However, depending on the partitioning concept of the system, one fast partition may be shared by several executable objects.

An important limitation with the use of fast partitions is that the amount of static memory regions limits the maximum number of regions that is available to threads using a dynamic memory partition, because the static regions permanently occupy region descriptors in the MPU. The number of static regions is the number of unique regions contained in the global partition, the kernel partitions, and, if configured, the fast partitions.

9.4.1.3. Configuring fast partitions

To configure the use of fast partitions, the following steps are necessary:

- Set the macro `MK_CFG_Cx_NSTATICPARTITIONS` to the number of static partitions on core `x`. The number is 2, i.e. global partition and kernel partition plus the number of fast partitions that shall be used. For processors with 4 protection register sets, this macro has a value in the range of 2 to 4.
- Fast partitions always occupy the partition indices starting from 2, following the global and the kernel partition. The first fast partition uses index 2, the second fast partition uses index 3.
- Any static region uses a smaller region index than the first dynamic region. This means that in the definition of the macro `MK_CFG_MEMORYREGIONCONFIG` all static regions need to be listed before the first dynamic region. A static region may be present in several static partitions, and also in dynamic partitions. To achieve this, refer to the corresponding region index when you assign regions to a partition in the `MK_CFG_MEMORYREGIONMAPCONFIG` macro.
- To assign a fast partition to a thread, assign the index of the respective fast partition to the thread or the task/ISR which occupies the thread, and set the processor status word to select the PRS corresponding to the index of the fast partition. For more information, see Section 9.3, “Processor state mapping on TriCore processors”.

9.4.2. Memory permissions on TriCore processors

The MPU on TriCore has separate region descriptors for executable regions and data regions with read/write permissions. In the microkernel, however, you can assign read, write, and execute permissions to any region. The microkernel appropriately assigns region descriptors of the MPU. If a region has read/write permissions and at the same time executable permissions, it occupies one code region descriptor and one data region descriptor in the MPU.

The following macros exist to set the permissions for a memory region of the microkernel:

`MK_MPERM_U_R`
: The region has read permissions (read-only region).

`MK_MPERM_U_RW`
: The region has read and write permissions (read-write region).

`MK_MPERM_U_RX`
: The region has read and execution permissions (read-execute region).

`MK_TRICORE_PERM_EXECUTE`
: The region has execution permissions (execute-only region).

Currently you cannot assign a region with execute rights to a dynamic partition. This means that you must assign all executable regions of the application to the global partition or the fast partitions.

9.4.3. Memory regions on TriCore processors

Architecture-specific memory region content

`MK_Io`. This region must allow access to the following devices for TriCore processors:

- System Timers (STMx, depending on the configuration - e.g. time stamp timer)
- System Control Unit (SCU)
- Interrupt Router (IR) and Interrupt Router SRC registers

`MK_OsIo`. This region must allow access to the System Timers (STMx) configured for usage by the QM-OS.

Architecture-specific memory regions

In addition to the memory regions described in Section 7.4.2.9.3, “Memory regions when using EB tresos Studio”, the microkernel expects the memory regions listed in Table 9.1, “TriCore-specific memory regions”

Memory region	Description	Access		
`MK_Csa_C<x>`	CSA lists for core `<x>`	r	w	

Table 9.1. TriCore-specific memory regions

9.4.4. Additional memory region flags for TriCore processors

This field is not used.

9.4.5. Additional configuration of memory protection on TriCore processors

The following TriCore-specific configuration macros need to be provided in addition to the configuration described before:

- **MK_CFG_DYNREGIONS_MAX.** The global maximum number of dynamic regions in any dynamic partition on any core. This is the maximum of the `MK_CFG_Cx_DYNREGIONS_MAX` macros of all used processor cores.
- **MK_CFG_Cx_DYNREGIONS_MAX.** The maximum number of dynamic regions in a dynamic partition on core `x`. Static regions that are assigned to a dynamic partition are not part of this number. This is the number of MPU region descriptors that are re-programmed whenever a change of the active dynamic partition occurs.
- **MK_CFG_Cx_NSTATICPARTITIONS.** The number of static partitions on core `x`.
- **MK_CFG_Cx_MPU_REGISTER_CHECK.** Enables or disables the MPU register read back after a change. If this macro is set to 1 the microkernel reads back changed MPU registers and compares them to their expected values as required by some TriCore safety manuals. Leave the macro at its default value 0 if you don't need these checks for your derivative and application.

9.5. Interrupt handling on TriCore processors

The interrupt handling in TriCore processors is different to most other microcontrollers. First it only supports hardware vectoring, i.e. every interrupt has its own entry address. The second and more significant difference is that each of these entry addresses is not associated with an interrupt source, but with an interrupt priority. This section describes the impact on the interrupt configuration of the microkernel.

As a trade-off between ROM usage and ease of linking, the microkernel combines the exception vectors and the interrupt vector in a single object section of 2048 bytes size. The name of this section is `MK_exctable`. This section has an alignment requirement of 2048 bytes assigned by the assembler, so no extra action is needed in the linker script to ensure proper placement. The section only contains a single interrupt vector at its end that is invoked for all types of interrupt requests. This effectively turns the hardware vectoring of the TriCore into software vectoring.

The software vector table of the microkernel maps an interrupt with priority `p` to the entry with the index `(p - 1)` of the microkernel's software interrupt table as defined by `MK_SOFTVECTOR_nnnn`, that is `nnnn` equals `(p - 1)`.

To obtain a correct interrupt configuration, you need to ensure that the hardware interrupt associated with an `MK_SOFTVECTOR_nnnn` entry has the correct priority assignment. To do so, ensure that the `irqcfg` parameter of the `MK_ISRCFG` entry referenced by `MK_SOFTVECTOR_nnnn` points to an `MK_IRQCFG` entry that has its `level` parameter set to `(nnnn + 1)`.

9.6. Timers on TriCore family processors

The microkernel's time services are based on the system timer unit (STM), which provides a free-running counter with at least 56 bits.

The microkernel's execution timing features are based on counter 0 of the processor's temporal protection system. The low word of the system timer that is also used for the time services serves as the time base for this feature.

9.7. Simple schedule table on TriCore processors

All supported TriCore processors have one STM per core. Each of which has two comparators. Such a comparator can be used to drive a SST counter.

The template `Mk_board.h` provides a macro called `MK_TimestampNsToTicks()` that converts nanoseconds to ticks of a STM. This macro can be used to calculate the correct reload values for a SST counter configuration. The existence of the macro is assumed by the generated configuration.

9.8. Exception handling on TriCore processors

The microkernel occupies all exception entry points of the TriCore processor. This is necessary to ensure freedom from interference, because after an exception, the processor automatically switches to supervisor mode. You can find the complete list of exceptions in the TriCore core architecture manual.

Most exceptions are passed on by invoking the `ProtectionHook()`. Some exceptions, however, are handled in a different way, as described below:

FCD

The *free context list depletion* exception is triggered when the core-wide free list of CSAs is nearly used up. Since the microkernel cannot determine which thread has used too many of the CSAs, it shuts down the system.

FCU

The *free context list underflow* exception is triggered, when the core-wide free list of CSAs is completely used up. This should not happen due to the handling of the FCD exception, see above. If this exception happens, the microkernel cannot handle this due to the lack of CSAs. This behavior leads to an endless loop of FCU exceptions.

CSU

The *call stack underflow* exception is used by the microkernel to detect a thread that returns from its entry function. The microkernel handles this as a normal thread termination and continues execution with the next eligible thread from the thread queue.

TAE

The *temporal asynchronous error* exception is used by the microkernel to detect a thread that exceeded its configured execution time budget. The microkernel handles this exception by a regular invocation of the dispatcher, which amongst other things checks whether the current thread exceeded its execution time budget. If this is the case, the corresponding protection fault is reported.

Syscall

The *system call* exception is the designated mechanism of the processor to enter the operating system for the purpose of performing a system service on behalf of the caller. This exception occurs frequently during normal operation, whenever a kernel-level API of the microkernel is invoked from a thread. It causes the context switch to the kernel mode. The microkernel handles this exception by performing the requested service. This exception occurs frequently during normal operation, whenever a kernel-level API of the microkernel is invoked from a thread. It causes the context switch to the kernel mode. The microkernel handles this exception by performing the requested service.

9.9. TriCore API

The TriCore-specific API elements are described as follows.

Name

mk_hwexceptioninfo_t — provides information about an exception

Synopsis

#include <public/Mk_error.h>

```
mk_hwexceptioninfo_t * const MK_exceptionInfo[];

mk_hwexceptioninfo_t * const MK_panicExceptionInfo[];
```

Description

The microkernel implementation for TriCore supports the `MK_GetPanicExceptionInfo` feature.

`MK_exceptionInfo[]` and `MK_panicExceptionInfo[]` are arrays of constant pointers to a structure into which the microkernel places detailed information about an exception that occurred on a specific core. The core index is used as index in the array.

The information in `MK_exceptionInfo[]` is valid while the `ProtectionHook()` executes and remains valid afterwards until the next exception is trapped.

The information in `MK_panicExceptionInfo[]` is valid while the `ShutdownHook()` executes after a kernel panic which was caused by an exception in kernel context.

`MK_exceptionInfo[]` and `MK_panicExceptionInfo[]` point to data structures of the following type:

```
typedef struct mk_exceptioninfo_s mk_hwexceptioninfo_t;

struct mk_exceptioninfo_s
{
  mk_exceptiontype_t type;

  mk_exceptionclass_t excClass;
  mk_exceptiontin_t excTin;

  mk_uint32_t dstr;
  mk_uint32_t datr;
  void * deadd;
  mk_uint32_t pstr;
  mk_uint32_t dietr;
  void * diear;
  mk_uint32_t pietr;
  void * piear;

  mk_lowerctx_t *lowerCtx;
  mk_upperctx_t *upperCtx;
};
```

The individual data fields are described below.

type
: An enumerated type that indicates which exception is trapped. You can find the possible values in the header file `public/TRICORE/Mk_TRICORE_exceptioninfo.h`.

excClass
: The trap class.

excTin
: The trap identification number within the trap class.

dstr
: The content of the Data Synchronous Error Trap Register (DSTR).

datr
: The content of the Data Asynchronous Error Trap Register (DATR).

deadd
: The content of the Data Error Address Register (DEADD).

pstr
: The content of the Program Synchronous Error Trap Register (PSTR).

dietr
: The content of the Data Integrity Error Trap Register (DIETR).

pietr
: The content of the Program Integrity Error Trap Register (PIETR).

piear
: The content of the Program Integrity Error Address Register (PIEAR).

lowerCtx
: The content of the lower context at the time when the exception occurred. Depending on the nature of the exception, you can find the instruction which caused the exception by reading the `pc` field of the lower context. `lowerCtx` may be `MK_NULL`, if no context information was stored. In this case you can probably obtain context information via `MK_protectionInfo`.

upperCtx
: The content of the upper context at the time the exception occurred. This pointer may be `MK_NULL`. See also `lowerCtx`.

The trap classes and the trap identification numbers are described in the TriCore User Manual:

Derivative	Document
TC22XL	[TC_ARCH_161_VOL1]
TC23XL	[TC_ARCH_161_VOL1]
TC27XT	[TC_ARCH_161_VOL1]
TC29XT	[TC_ARCH_161_VOL1]
TC33XL	[TC_ARCH_162_VOL1]
TC36XD	[TC_ARCH_162_VOL1]
TC37XT	[TC_ARCH_162_VOL1]
TC38XQ	[TC_ARCH_162_VOL1]
TC38XT	[TC_ARCH_162_VOL1]
TC39XX	[TC_ARCH_162_VOL1]
TC39XQ	[TC_ARCH_162_VOL1]

The register contents as well as their relevance in case of a certain exception are described in the AURIX User's Manual for your derivative, e.g. [TC27X_UM_C] for the TC27X C-step. Be sure to use the most recent manual for your device.

9.10. Multi-core support on TriCore processors

Number of supported cores

The microkernel supports all processor cores on the supported TriCore multi-core processors. Table 9.2, “Number of supported processor cores (`MK_MAXCORES`)” shows the derivative-specific number of available cores, also stored as constant `MK_MAXCORES` in the microkernel.

Derivative	**`MK_MAXCORES`**
TC22XL	1
TC23XL	1
TC27XT	3
TC29XT	3
TC33XL	1
TC36XD	2
TC37XT	3

Derivative	`MK_MAXCORES`
TC38XQ	4
TC38XT	3
TC39XX	6
TC39XQ	4

Table 9.2. Number of supported processor cores (`MK_MAXCORES`)

Core ID mapping

On the AURIX family of processors, physical core indices are used to assign consecutive numbers to cores, as opposed to physical core IDs which may contain gaps. Therefore physical core indices map to same physical core IDs until 4. Physical core indices 5 or more map to physical core IDs one greater, which means physical core index 5 is mapped to physical core ID 6 and continuing. Logical core IDs map to same physical core indices. This needs to be taken into consideration in many situations, for example: indexing an array that contains an element for each core.

If Advanced Logical Core Identifiers are enabled, the mapping of logical core IDs to physical core indices can be customized. You can find further information on this topic in [ASCOS_USERGUIDE].

Inter-core interrupts

The microkernel uses the GPSR (general purpose service request) nodes to implement the inter-core interrupts. Each core that is controlled by the microkernel needs one inter-core interrupt.

Table 9.3, “Default inter-core interrupts” shows the default inter-core interrupts if EB tresos Studio is used to configure the microkernel. The default IRQ level in the `MK_IRQCFG()` macro is `255` and the interrupt request is enabled on startup. Inter-core interrupts are not available on the single core derivatives TC22XL, TC23XL and TC33XL.

Derivative	Core ID	Service Request Node (SRN)
TC27XT	▶ i (within [0..2])	▶ GPSR2i
TC29XT	▶ i (within [0..2])	▶ GPSR2i
TC36XD	▶ 0 ▶ 1	▶ GPSR16 ▶ GPSR17
TC37XT	▶ i (within [0..2])	▶ GPSR2i
TC38XQ	▶ i (within [0..2]) ▶ 3	▶ GPSR2i ▶ GPSR37

Derivative	Core ID	Service Request Node (SRN)
TC38XT	▶ i (within [0..2])	▶ GPSR2i
TC39XX	▶ i (within [0..2]) ▶ 3 ▶ 4 ▶ 5	▶ GPSR2i ▶ GPSR37 ▶ GPSR41 ▶ GPSR42
TC39XQ	▶ i (within [0..2]) ▶ 3	▶ GPSR2i ▶ GPSR37

Table 9.3. Default inter-core interrupts

9.11. Compiler Stack Smashing Protection on TriCore processors

The Wind River® Diab Compiler Toolchain offers a stack smashing protection feature which can detect stack-based buffer overflow during runtime. To achieve this, the compiler inserts additional code in the function prolog and epilog. In the prolog the tool writes a canary value to the stack at a position in between the local variable area and the return address. In case the canary value is overwritten by faulty or malicious code, a comparison with the known canary value will reveal the corruption in the function epilog. If this happens, an error hook function is called automatically. In EB tresos Safety OS this error hook function is designed to cause a protection fault by default. For more information on the stack smashing protection of the Diab Compiler Toolchain please refer to the Diab Compiler User's Guide for TriCore.

NOTE

Stack smashing feature requires additional macro setting

If the stack smashing protection of the Diab Compiler Toolchain is active, the macro `MK_CCOPT_USE_COMPILER_STACK_PROTECTION` must be set to `1` at compile-time. This is necessary for EB tresos Safety OS to support this feature.

If the stack smashing protection detects a stack corruption during runtime and EB tresos Safety OS support for this feature is switched on, a protection fault will occur and the microkernel will do one of the following actions:

- If the stack corruption did not occur in kernel context, e.g. during task execution, the `ProtectionHook()` function will be called if it was enabled in the configuration. In this case, the `errorId` of the protection fault, provided by `MK_GetProtectionInfo()`, will be `MK_eid_IllegalInstructionException` and the `culprit` address will point to the assembler function `MK_CompilerStackCheckFailLandingPad`. See ProtectionHook for more information on the protection hook function.

  If the `ProtectionHook()` was not enabled in the configuration the microkernel will shut down. If the `ShutdownHook()` was enabled in the configuration it is possible to call `MK_GetProtectionInfo()`

from within the `ShutdownHook()` and obtain information on the nature of the exception as described above.

- If the stack corruption occurred in kernel context, e.g. during a system call, the function `MK_Panic()` will be called and the microkernel will shut down. The origin of the exception, provided by `MK_GetPanicExceptionInfo()`, will point to the assembler function `MK_CompilerStackCheckFailLandingPad`. See Section 7.3.4, “Panics” for more information on microkernel panics.

Appendix A. 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/autosar/asc_MicroOs/trunk/doc/public/userguides.

Filename	Revision
../../project/fragments/cross_references/Cross_References.xml	41608
../../project/fragments/diagrams/mempart.svg	29657
../../project/fragments/glossary/Glossary.xml	39071
../fragments/About_This_Documentation/About_This_Documentation.xml	2500
../fragments/entities/SafetyOS.ent.m4	36173
../fragments/Quality_Statement.xml	36324
ApiRef.xml	36521
BackgroundInformation.xml	39279
BeginHere.xml	30063
BoardConfig.xml	24467
CalloutRef.xml	29153
ConfigRef.xml	41488
Conventions.xml	6871
Deviations.xml	29508
History.xml	41617
HowToUse.xml	39279
images/MemoryRegion.png	6893
images/MemoryRegionRef.png	6893
images/OsIsr.png	6893
images/Resource.png	6893
images/ResourceRef.png	6893
images/ScheduleNon.png	18403
Limitations.xml	24481
Limits.xml	24467
LinkerSymbolsArmToolchain.xml	29980

Filename	Revision
Microkernel_users_guide.xml	38199
Ref_ActivateTask.xml	29506
Ref_AllowAccess.xml	22955
Ref_CallTrustedFunction.xml	24474
Ref_ChainTask.xml	24467
Ref_ClearEvent.xml	22544
Ref_DisableInterruptSource.xml	24094
Ref_EnableInterruptSource.xml	24094
Ref_ErrorGetParameter.xml	24474
Ref_ErrorHook.xml	24479
Ref_GetApplicationId.xml	24433
Ref_GetApplicationState.xml	22544
Ref_GetCurrentApplicationId.xml	24467
Ref_GetEvent.xml	22587
Ref_GetIsrId.xml	22955
Ref_GetTaskId.xml	22955
Ref_GetTaskState.xml	24467
Ref_IsScheduleNecessary.xml	24433
Ref_MK_ConditionalGetResource.xml	29462
Ref_MK_DiffTime.xml	33546
Ref_MK_ElapsedMicroseconds.xml	33546
Ref_MK_ElapsedTime.xml	33546
Ref_MK_GetErrorInfo.xml	24433
Ref_MK_GetExceptionInfo.xml	39088
Ref_MK_GetPanicExceptionInfo.xml	33715
Ref_MK_GetPanicReason.xml	24474
Ref_MK_GetProtectionInfo.xml	26404
Ref_MK_InitSyncHere.xml	24467
Ref_MK_MulDiv.xml	29922
Ref_MK_ReadTime.xml	18403
Ref_MK_ReportError.xml	27313
Ref_MK_ResumeCallout.xml	29269

Filename	Revision
Ref_mk_statusandvalue_t.xml	24433
Ref_MK_SuspendCallout.xml	29269
Ref_mk_time_t.xml	33546
Ref_OsAlarmApi.xml	24474
Ref_OsCounterApi.xml	24474
Ref_OSErrorGetServiceId.xml	24474
Ref_OsLockAcqApi.xml	29260
Ref_OsLockRelApi.xml	29262
Ref_OsScheduleTableApi.xml	24474
Ref_OsScheduleTableSyncApi.xml	24474
Ref_ProtectionHook.xml	24479
Ref_Schedule.xml	24433
Ref_ScheduleIfNecessary.xml	24474
Ref_SetEvent.xml	29506
Ref_ShutdownAllCores.xml	29506
Ref_ShutdownHook.xml	24479
Ref_ShutdownOs.xml	24433
Ref_StartCore.xml	29506
Ref_StartOs.xml	24467
Ref_StartupHook.xml	24433
Ref_TerminateApplication.xml	24474
Ref_TerminateTask.xml	22587
Ref_TimeConversion.xml	29914
Ref_TimestampConv.xml	29914
Ref_WaitEvent.xml	24467
Ref_WaitGetClearEvent.xml	24474
ReferenceManual.xml	22461
Safe_And_Correct_Use.xml	38199
Supplement_TRICORE.xml	41596
UML/Conventions.svg	6831
UML/ErrorHook.svg	6831
UML/Events.svg	6848

Filename	Revision
UML/FPTasks.svg	6831
UML/InterruptLocks.svg	22955
UML/Interrupts.svg	6831
UML/MemProtConsumerProducer.svg	22986
UML/MemProtSimple.svg	22986
UML/NPTasks.svg	6831
UML/ProtectionHook.svg	6831
UML/Resources.svg	6848
UML/Shutdown.svg	6831
UML/Startup.svg	22544
UML/Tasks.svg	22955
UML/TrustedFunctions.svg	22979
UsersGuide_TRICORE.ent	38199

Glossary

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*.
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.
private data	Private data can only be accessed by the *task* that owns the data. These data are declared when the OS is configured.
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*.
system call	A system call is how an OS-object requests a service from an operating system's kernel, with hardware support. A system call separates OS-Applications and the operating system via a context switch. The hardware ensures that the control flow continues in the kernel and switches to the *privileged mode*.
task	A task is an *OS-object* that is started, scheduled and terminated by the operating system.
thread	A thread is an executing instance of a subprogram and forms a logical construct to share the resource *CPU*. Threads provided by the microkernel have the following properties: ▶ The execution can be suspended synchronously and asynchronously.

	▶ After suspension, the execution can be resumed. ▶ Memory protection settings are associated with a thread and are enforced when the thread is executing. ▶ A dedicated stack exists. ▶ A priority allows to prioritize how the resource *CPU* is assigned to different threads.
`trusted OS-Application`	*Trusted* is the AUTOSAR term for *OS-Applications* whose *OS-objects* run in *privileged mode*. Write access to other OS-Application and their related *OS-objects* may be restricted depending on the memory protection configuration. This term does not involve any statement about the ASIL of the code. Opposite of *non-trusted OS-Application*.

Bibliography

[ASCOS_USER-GUIDE]	Elektrobit Automotive GmbH:*EB tresos AutoCore OS documentation*
[AU-TOSAROS42SPEC]	AUTOSAR:*Specification of Operating System*, Version 4.2.2 https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_OS.pdf
[EBMASANNEX]	Elektrobit Automotive GmbH:*EB tresos® maintenance and support annex*
[ISO26262_1ST]	*INTERNATIONAL STANDARD ISO 26262*: Road vehicles - Functional safety, 2011
[OSEKOS223S-PEC]	OSEK/VDX:*Operating System Specification*, Version 2.2.3 https://web.archive.org/web/20160310151905/http://portal.osek-vdx.org/files/pdf/specs/os223.pdf
[SAFETYMANTRI-CORE]	Elektrobit Automotive GmbH:*EB tresos Safety OS safety manual for TriCore family* 8.2_Safety_OS_safety_manual_TRICORE.pdf
[TC_ARCH_161_-VOL1]	Infineon:*TriCore TC1.6P & TC1.6E User Manual (Volume 1)*, V1.0 D10, February 2012
[TC_ARCH_162_-VOL1]	Infineon:*TriCore TC 1.6.2 core architecture manual (Volume 1)*, V1.0, January 2017
[TC27X_UM_C]	Infineon:*AURIX TC27x C-Step User's Manual*, V2.2, December 2014
[UGARM64RR1]	*Review Report 1 of the User's Guide for ARM64 family processors*
[UGARMRR1]	*Review Report 1 of the User's Guide for ARM family processors*
[UGARMRR2]	*Review Report 2 of the User's Guide for ARM family processors*

[UGCORTEXMRR1]	*Review Report 1 of the User's Guide for CORTEXM family processors*
[UGPARR1]	*Review Report 1 of the User's Guide for PA family processors*
[UGRH850RR1]	*Review Report 1 of the User's Guide for RH850 family processors*
[UGTRICOR-ERMK2R1]	*Review Report 1 of the MK 2.0 User's Guide for TRICORE family processors*