



Elektrobit

EB tresos[®] Safety OS safety manual

for TriCore family

Date: 2020-12-09, Document version V2.1.31-MicroOs2.0_TRICORE_inspected_2020cw48, Status:
RELEASED



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

Technical support

<https://www.elektrobit.com/support>

Legal disclaimer

Confidential and proprietary information

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Elektrobit Automotive GmbH.

All brand names, trademarks and registered trademarks are property of their rightful owners and are used only for description.

Copyright 2020, Elektrobit Automotive GmbH.

Table of Contents

1. Document history	7
2. Document information	17
2.1. Objective	17
2.2. Scope and audience	17
2.3. Motivation	17
2.4. Structure	18
2.5. Typography and style conventions	18
3. About EB tresos Safety OS	21
3.1. Architecture of the surrounding system	21
3.1.1. Microcontrollers in the TriCore family	21
3.2. Description of the microkernel	22
3.2.1. Interaction with the surrounding system	23
3.2.2. Outstanding anomalies	24
3.2.3. Backward compatibility	24
3.2.4. Compatibility with other systems	25
3.2.5. Change control	25
3.2.6. Safety mechanism used by the microkernel	25
3.2.7. Identification of files in the microkernel	26
3.2.8. Robustness of the microkernel	26
3.2.8.1. What the microkernel does not do	26
3.2.8.2. Robustness against hardware faults	27
3.2.8.3. Robustness against systematic software errors	27
3.2.8.4. Robustness against configuration errors	27
3.2.8.5. Robustness against resource conflicts	27
3.2.8.6. Robustness against interrupt overload	28
3.2.8.7. Robustness against input errors	28
4. Using EB tresos Safety OS safely	30
4.1. Applicability of this document	30
4.2. Prerequisites	30
4.3. Installing EB tresos Safety OS	31
4.4. Verifying the integrity of the microkernel sources	31
4.5. Using EB tresos Studio	32
4.5.1. Configuring the operating system module	32
4.5.2. Generating the operating system module	32
4.5.3. Verifying generated files	33
4.6. Implementing your system	33
4.6.1. Protect against non-lockstep cores	33
4.6.2. Avoid symbols reserved by EB tresos Safety OS	34
4.6.3. EB tresos Safety OS services	34

4.6.3.1. Services with integrity category 1	35
4.6.3.2. Services with integrity category 2	35
4.6.3.3. Services with integrity category 3	35
4.6.4. Writing callout functions	36
4.6.4.1. Callout functions that are called directly	36
4.6.4.2. Callout functions that are called in a thread	36
4.6.4.3. Alarm callback functions	37
4.6.5. Starting the microkernel	37
4.6.6. Shutting down the microkernel	38
4.6.7. Testing the memory protection mechanism	39
4.6.8. Accuracy of simple schedule table	40
4.7. Writing the linker script	41
4.8. Building your system	41
4.9. Verifying the binary image	42
4.9.1. Ensure that the microkernel contains no low-integrity software	42
4.9.2. Ensure that code is correctly placed	42
4.9.3. Ensure that variables are correctly placed	43
4.9.4. Ensure that the representation of the memory regions is correct	43
4.9.5. Ensuring the integrity of the binary image	45
4.10. Implications of the compiler settings for use with the microkernel	45
4.11. Steps outside the scope of EB	46
5. Application constraints and requirements	47
5.1. Safety Element out of Context (SEooC) definition	47
5.1.1. Functional scope of the SEooC	47
5.1.1.1. Provided functionality	47
5.1.1.2. Assumed hardware functionality	48
5.1.1.3. Assumed employment	48
5.1.1.4. Assumed compiler confidence level	48
5.1.1.5. Assumed external functionality	48
5.1.2. Implementation scope of the SEooC	49
5.2. Definitions	49
5.2.1. Process words	49
5.2.2. Term definitions	50
5.2.2.1. Term: freedom from interference	53
5.2.2.1.1. Freedom from interference in ISO 26262	53
5.2.2.1.2. Non-interference between software elements in IEC 61508	53
5.2.3. Requirements	54
5.3. Level 1 requirements	54
5.4. Level 2 requirements	55
5.4.1. Operating system requirements	55
5.4.2. Freedom from interference requirements	58
5.5. Assumptions	60

5.5.1. Defining assumptions	60
5.5.2. Assumptions on the hardware	60
5.5.3. Assumptions on the surrounding system	61
5.6. Overview	62
6. Configuration verification criteria	64
6.1. Verification criteria for Mk_gen_global.c	65
6.2. Verification criteria for Mk_gen_config.h	66
6.2.1. Verification criteria for global settings	67
6.2.2. Verification criteria for microkernel executables	69
6.2.3. Verification criteria for the QM-OS	70
6.2.4. Verification criteria for trusted function threads	72
6.2.5. Verification criteria for hook functions	73
6.2.6. Verification criteria for interrupts and ISRs	75
6.2.7. Verification criteria for internal interrupts	79
6.2.8. Verification criteria for tasks	79
6.2.9. Verification criteria for threads and job queues	84
6.2.10. Verification criteria for OS-Applications	85
6.2.11. Verification criteria for locks	86
6.2.12. Verification criteria for memory partitions	89
6.2.13. Verification criteria for simple schedule tables	93
6.2.14. Verification criteria for advanced logical core identifiers	94
6.2.15. Additional configuration criteria for TriCore family processors	95
6.2.15.1. Verification criteria for stacks on TriCore family processors	95
6.2.15.2. Verification criteria for interrupts on TriCore family processors	96
6.2.15.3. Verification criteria for memory regions on TriCore family processors	96
6.2.15.4. Verification criteria for TriCore family processors with multiple cores	97
6.2.15.5. Verification criteria for simple schedule table on TriCore family processors	97
6.3. Verification criteria for Mk_gen_user.h	98
6.4. Verification criteria for Mk_gen_addons.h	99
6.5. Verification criteria for Mk_board.h	99
7. TriCore family supplement	102
7.1. Using TriCore processors safely	102
7.2. Safety of OS callout functions specific to TriCore	104
7.2.1. MK_ProtectRamFromExternal	105
7.3. Core special function registers (CSFRs)	105
7.4. Memory protection unit	109
7.5. Protection from other bus masters	110
7.6. Interrupt router	111
7.7. Code placement	112
7.8. Linking requirements	112
7.9. Verification measures for hardware	113
7.10. Hardware timers	114

7.10.1. The System Timer (STM)	114
7.10.2. The Temporal Protection System (TPS)	115
7.11. TriCore-specific processor status	115
7.12. Start-up preconditions	116
7.13. Supported TriCore family processors	117
7.14. Safety-related documentation used during development	132
7.14.1. TC22xL	132
7.14.2. TC23xL	132
7.14.3. TC27xT	132
7.14.4. TC29xT	133
7.14.5. TC33xL	133
7.14.6. TC36xD	133
7.14.7. TC37xT	133
7.14.8. TC38xT	134
7.14.9. TC38xQ	134
7.14.10. TC39xX	134
7.14.11. TC39xQ	134
A. Reserved identifiers	136
B. Safety of OS services	139
B.1. Safety of short-duration time services	144
B.2. Safety of asynchronous QM-OS services	144
C. Safety of OS callout functions	145
C.1. Implementation notes	146
D. Document configuration information	147
Glossary	149
Bibliography	152

1. Document history

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

Version	Date	State	Description
V2.0	2015-07-29	DRAFT	Initial version. This version is derived from the safety manual version 1.14 and does not include the multi-core features.
V2.0.1	2015-08-06	DRAFT	Rebuild for release without changes.
V2.0.2	2015-08-11	DRAFT	Rebuild for release without changes.
V2.0.3	2015-11-17	DRAFT	Rebuild for development drop without changes.
V2.0.4	2015-12-15	DRAFT	Re-phrased criterion [VC.Mk_gen_config.ISR.4.15] to be more strict.
V2.0.5	2016-02-17	DRAFT	Rebuild for release without changes.
V2.0.6	2016-02-24	DRAFT	Rebuild for release without changes.
V2.0.7	2016-04-14	DRAFT	Update reference to ISO26262 in SEooC chapter.
V2.0.8	2016-04-29	PROPOSED	Proposed safety manual for the multi-core microkernel.
V2.0.9	2016-05-02	PROPOSED	Proposed safety manual for the multi-core microkernel with updated tracing links.
V2.0.10	2016-06-22	PROPOSED	Proposed safety manual, incorporating rework checks and other tickets.
V2.0.11	2016-06-28	PROPOSED	Incorporate rework from TE review.
V2.0.12	2016-06-28	PROPOSED	Incorporate further rework from inspection 1.
V2.0.12	2016-06-28	RELEASED	Two minor typographical changes. The document has passed its verification; see [SM2INSP1] .
V2.0.13	2016-08-09	PROPOSED	<ul style="list-style-type: none">▶ Add the following new verification criteria:<ul style="list-style-type: none">▶ [VC.Mk_gen_config.global.25]▶ [VC.Mk_gen_config.QM.10]▶ [VC.Mk_gen_config.QM.11]▶ [VC.Mk_gen_config.QM.12]▶ [VC.Mk_gen_config.QM.13]▶ [VC.Mk_gen_config.QM.14]▶ [VC.Mk_gen_config.QM.15]▶ [VC.Mk_gen_config.QM.16]

Version	Date	State	Description
			<ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.QM.17] ▶ [VC.Mk_gen_config.QM.18] ▶ [VC.Mk_gen_config.QM.19] ▶ Correct the following verification criteria: <ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.Hook.ErrorHook.1.3] ▶ [VC.Mk_gen_config.Hook.ShutdownHook.1.2] ▶ [VC.Mk_gen_config.Locks.8] ▶ [VC.TRICORE.Startup.3] ▶ Correct names and descriptors in the following verification criteria: <ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.IRQ.1] ▶ [VC.Mk_gen_config.IRQ.2] ▶ [VC.Mk_gen_config.ISR.4.3] ▶ [VC.Mk_gen_config.ISR.4.17] ▶ [VC.Mk_gen_config.Task.4.6] ▶ [VC.Mk_gen_config.Task.4.20] ▶ [VC.Mk_gen_config.Locks.1] ▶ [VC.Mk_gen_config.Locks.2] ▶ [VC.Mk_gen_config.MemoryPartition.2] ▶ [VC.Mk_gen_config.MemoryPartition.5.1] ▶ Clarify the following verification criteria: <ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.MK.5] ▶ [VC.Mk_gen_config.MK.9] ▶ [VC.Mk_gen_config.IRQ.2.3] ▶ [VC.Mk_gen_config.ISR.4.4] ▶ [VC.Mk_gen_config.Locks.4] ▶ [VC.Mk_gen_config.Locks.5] ▶ [VC.Mk_gen_config.Locks.6] ▶ [VC.Mk_gen_config.MemoryPartition.7.A] ▶ Remove the following duplicate verification criteria: <ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.Task.4.8.1]

Version	Date	State	Description
			<ul style="list-style-type: none"> ▶ Add chapter 4.6.1 concerning multi-core processors which lack lockstep cores. ▶ Add an application note concerning internal shutdowns to chapter 4.6.6. ▶ Add a description of <i>memory partition indexes</i> to chapter 6.2.
V2.0.14	2016-08-11	PROPOSED	<ul style="list-style-type: none"> ▶ Clarify the following verification criteria: <ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.IRQ.1] ▶ [VC.Mk_gen_config.IRQ.2.3] ▶ [VC.Mk_gen_config.ISR.4.4.1] ▶ Remove the following verification criteria: <ul style="list-style-type: none"> ▶ [VC.Mk_gen_config.global.20]
V2.0.14	2016-08-11	RELEASED	The document has passed its verification; see [SM2INSP2] .
V2.0.15	2016-08-11	PROPOSED	<p>Correct occurrences of <code>MK_InitHwAfterData()</code> to the correct name <code>MK_InitHardwareAfterData()</code>. This includes the following verification criteria:</p> <ul style="list-style-type: none"> ▶ [VC.TRICORE.STM.1]
V2.0.16	2016-09-02	PROPOSED	[TRICORE]: "The System Timer (STM)": Clarify, that the STM must not get disabled or reset and recommend no write permissions for its registers.
V2.0.17	2016-09-06	PROPOSED	Fix some grammatical errors.
V2.0.17	2016-09-07	RELEASED	The document has passed its verification; see [SM2INSP3] .
V2.0.18	2016-12-08	PROPOSED	<ul style="list-style-type: none"> ▶ Removed duplicate entry for <code>MK_InitHardwareBeforeData()</code> from table of callout functions. ▶ Changed <code>Mk_gen_src.c</code> to <code>Mk_gen_global.c</code>. ▶ Changed <code>MK_cx_taskThreads[n]</code> to <code>MK_cx_taskThreads[i]</code>. ▶ Added an instruction describing how to handle <code>OS_SYSTEM_n</code> OS-Applications. ▶ Inserted the missing <code>CFG_</code> into some macro names. ▶ Added more VCs for the OS-Application configuration. ▶ Replaced all instances of "startup" in text with "start-up". ▶ Replaced all instances of "timestamp" in text with "time stamp".

Version	Date	State	Description
V2.0.18	2016-12-08	RELEASED	The document has passed its verification; see [SM2INSP4] .
V2.0.19	2017-02-01	PROPOSED	<ul style="list-style-type: none"> ▶ EB tresos Safety OS does no longer implement StartNonAutosarCore. ▶ Added <code>coreIndex</code> parameter to <code>MK_IRQCFG</code>. ▶ Added some clarifications and language improvements. ▶ Added note about CSA limitations. ▶ [TRICORE]: Added verification criteria [VC.TRICORE.Hw-Ps.*] for hardware specific processor status. ▶ [TRICORE]: Added section about the interrupt router.
V2.0.20	2017-02-10	PROPOSED	<ul style="list-style-type: none"> ▶ Corrected spelling mistakes. ▶ Referenced the user's guide concerning "API access protection".
V2.0.20	2017-02-24	RELEASED	The document has passed its verification; see [SM2INSP5] .
V2.0.21	2017-05-08	PROPOSED	<ul style="list-style-type: none"> ▶ [RH850] First complete version for RH850F1H ▶ Added verification criterion [VC.FASTLOCKING.1]. ▶ Added verification criterion VC.Mk_board.13.
V2.1	2017-06-21	PROPOSED	<ul style="list-style-type: none"> ▶ Integrates simple schedule table. ▶ Reformulated statement about low-integrity software. ▶ Renamed [VC.FASTLOCKING.1] to [VC.LOCKINGCALL-OUTS.1] and updated text. ▶ Added non-preemptive ISRs. ▶ Added [VC.Mk_gen_config.ISR.4.32], [VC.Mk_gen_config.ISR.4.33].
V2.1.1	2017-06-30	PROPOSED	<ul style="list-style-type: none"> ▶ Corrected language mistakes. ▶ Added some helpful cross-references.
V2.1.1	2017-07-03	RELEASED	The document has passed its verification; see [SM2INSP6] .
V2.1.2	2018-08-10	PROPOSED	<ul style="list-style-type: none"> ▶ [CORTEXM] Add a dummy SM for CORTEXM to be released with the first development drop. ▶ [TRICORE] Reformulate [VC.TRICORE.Startup.3] to clarify that this covers TC1.6P's functional deviation CPU_TC.123. ▶ [TRICORE] Adapt to TC1.6.2 code region alignment. Add [VC.TRICORE.MPU.4].

Version	Date	State	Description
			<ul style="list-style-type: none"> ▶ [RH850] Add PSW.NP bit handling to the supported thread modes. ▶ [TRICORE] Add TC22XL and TC23XL. ▶ Add [VC.Mk_gen_config.ISR.5], [VC.Mk_gen_config.ISR.4.-34], [VC.Mk_gen_config.ISR.4.35]. Adapt all Verification Criteria for MK_ISRCFG to also support MK_QMOSISRCFG.
V2.1.3	2018-09-27	PROPOSED	<ul style="list-style-type: none"> ▶ [TRICORE] Update [VC.TRICORE.SST.1] and [VC.TRICORE.SST.2] ▶ Update [VC.Mk_gen_config.ISR.5].
V2.1.3	2018-09-27	RELEASED	The document has passed its verification; see [SM2INSP7] .
V2.1.4	2018-10-02	PROPOSED	rework after TE review.
V2.1.4	2018-10-02	RELEASED	The document has passed its verification; see [SM2INSP7] .
V2.1.5	2018-10-10	PROPOSED	[RH850] Improve comprehensibility and grammar of RH850 supplement.
V2.1.5	2018-10-11	RELEASED	The document has passed its verification; see [SM2INSP8] .
V2.1.6	2018-11-29	PROPOSED	<ul style="list-style-type: none"> ▶ Added support for RH850P1HC: <ul style="list-style-type: none"> ▶ Revised section "Using RH850 processors safely" ▶ Revised section "The INTC1/INTC2 interrupt controllers" ▶ Extended and revised section "Hardware timers" ▶ Extended and revised section "Supported RH850 family processors" ▶ Extended section "Safety-related documentation used during development" ▶ Removed [VC.RH850.SST.1] ▶ Added [VC.RH850.SST.1.TAUJ] ▶ Added [VC.RH850.SST.1.STM] ▶ Revised [VC.RH850.SST.2]
V2.1.7	2019-01-24	PROPOSED	Rework after inspection.
V2.1.8	2019-01-31	PROPOSED	Rework after inspection.
V2.1.8	2019-02-01	RELEASED	The document has passed its verification; see [SM2INSP9] .
V2.1.9	2019-02-27	PROPOSED	<p>Rework after SAR walkthrough:</p> <ul style="list-style-type: none"> ▶ Revised section "Code Placement"

Version	Date	State	Description
			▶ Revised [VC.RH850.Startup.1]
V2.1.9	2019-03-01	RELEASED	The document has passed its verification; see [SM2INSP9] .
V2.1.10	2019-06-17	PROPOSED	<ul style="list-style-type: none"> ▶ Added [VC.Mk_gen_config.MemoryPartition.13] ▶ Corrected the Integrity category of the following services: GetCoreID, StartOS, and ShutdownOS. ▶ Added the service ShutdownAllCores to the list. ▶ [RH850] Renamed SafetyManual.RH850.RH850P1HC.ECM to SafetyManual.RH850.ECM ▶ [RH850] Added VC.RH850.InternalISR.0 and VC.RH850.InternalISR.1 ▶ [RH850] Extended section "OSTM-based timing protection timers" ▶ [RH850] Revised section "The time stamp timer" ▶ [RH850] Revised section "The execution timer" ▶ [RH850] Added subsection for RH850D1X to section "Supported RH850 family processors" ▶ [TRICORE] Add TC3xx ▶ [TRICORE] Add TC38XT ▶ Revised [VC.Mk_gen_config.SST.3] ▶ Revised [VC.Mk_gen_config.SST.11] ▶ Revised [VC.Mk_gen_config.SST.13] ▶ Revised [VC.Mk_gen_config.SST.14] ▶ Revised [VC.Mk_gen_config.SST.15] ▶ Incorporated TE review findings ▶ [TRICORE] Update references to hw safety manual ▶ [ARM] Initial version of safety manual for ARM on microkernel 2.0.
V2.1.11	2019-07-01	PROPOSED	Rework after inspection.
V2.1.11	2019-07-01	RELEASED	The document has passed its verification; see [SM2INSP10] .
V2.1.11- ^a	2019-07-10	PROPOSED	[RH850] Rework after inspection
V2.1.12- ^a	2019-07-12	PROPOSED	[RH850] Rework after inspection
V2.1.12- ^a	2019-07-15	RELEASED	[RH850] The document has passed its verification; see [SM2INSP11] .

Version	Date	State	Description
V2.1.12	2019-07-22	PROPOSED	[ARM] Prepared initial version of safety manual for ARM on micro-kernel 2.0.
V2.1.13	2019-08-22	PROPOSED	Rework after inspection.
V2.1.14	2019-08-26	PROPOSED	Rework after inspection.
V2.1.14	2019-08-26	RELEASED	The document has passed its verification; see [SM2INSP12] .
V2.1.15	2019-08-30	PROPOSED	<ul style="list-style-type: none"> ▶ Added support for the feature Advanced Logical Core Identifiers (ALCI). ▶ [TRICORE] Added support for Compiler Stack Smashing Protection (Diab Compiler). ▶ [TRICORE] Updated document references. ▶ [TRICORE] Added support for interrupt router override.
V2.1.16	2019-09-19	PROPOSED	Rework after inspection.
V2.1.17	2019-09-20	PROPOSED	Rework after inspection.
V2.1.17	2019-09-20	RELEASED	The document has passed its verification; see [SM2INSP13] .
V2.1.18	2019-09-25	PROPOSED	<ul style="list-style-type: none"> ▶ [PA] Initial version for PA. ▶ [PA] Add SPC58XG.
V2.1.19-MicroOs2.0_-TRI-CORE_PA_-inspected_2019cw35	2019-11-20	PROPOSED	Rework after inspection.
V2.1.19	2019-11-22	PROPOSED	<ul style="list-style-type: none"> ▶ [ARM] Added derivative RCARV3MCR7. ▶ [ARM] Added derivative RCARV3HCR7. ▶ [TRICORE] Update references to hw safety manual. ▶ Rework after TE review. ▶ Fix some spelling mistakes. ▶ [ARM64] Prepare document for ARM64, first derivative RCARV3M.
V2.1.20-MicroOs2.0_-TRI-CORE_PA_-	2019-11-26	PROPOSED	Rework after inspection.

Version	Date	State	Description
inspect-ed_2019cw35			
V2.1.20-MicroOs2.0_-TRI-CORE_PA_-inspect-ed_2019cw35	2019-11-27	RELEASED	The document has passed its verification; see [SM2INSP14] .
V2.1.20	2020-01-10	PROPOSED	Rework after inspection.
V2.1.20	2020-01-14	RELEASED	The document has passed its verification; see [SM2INSP15] .
V2.1.21	2020-02-13	DRAFT	[ARM64] Draft for RCARV3M.
V2.1.22	2020-02-24	RELEASED	The document has passed its verification; see [SM2INSP16] .
V2.1.23	2020-03-24	PROPOSED	<ul style="list-style-type: none"> ▶ [ARM] Added derivative ZUXEVCR5. ▶ [ARM64] Proposed version for RCARV3M. ▶ [TRICORE] Added derivative TC36XD.
V2.1.24	2020-04-22	PROPOSED	▶ [ARM] Reworked inspection findings for ZUXEVCR5.
V2.1.25	2020-04-24	PROPOSED	▶ [ARM] Reworked inspection findings for ZUXEVCR5.
V2.1.25	2020-04-28	RELEASED	The document has passed its verification; see [SM2INSP17] .
V2.1.26	2020-04-28	PROPOSED	<ul style="list-style-type: none"> ▶ [TRICORE] Added derivative TC33XL. ▶ Section 3.2.6. Safety mechanism used by the microkernel: updated text for clarity. ▶ [TRICORE] Added missing TC3 derivatives in chapter 3.1.1. ▶ [TRICORE] Update references to hw safety manual. ▶ [TRICORE] Added derivative TC37XT.
V2.1.27	2020-06-03	PROPOSED	▶ [TRICORE] Reworked inspection findings for TC33XL, TC36XD and TC37XT.
V2.1.27	2020-06-04	RELEASED	The document has passed its verification; see [SM2INSP19] .
V2.1.28-MicroOs2.0_-ARM64_-inspection_2020cw24	2020-06-09	PROPOSED	<ul style="list-style-type: none"> ▶ [ARM64] Revised section "Interruption by higher exception levels". ▶ [ARM64] Revised [VC.ARM64.Startup.3]. ▶ [ARM64] Revised section "Reliable execution environment". ▶ [ARM64] Revised section "Memory management unit". ▶ [ARM64] Revised section "Interrupt controller".

Version	Date	State	Description
			<ul style="list-style-type: none"> ▶ [ARM64] Revised section "Hardware-specific mechanisms". ▶ [ARM64] Added [VC.ARM64.Startup.14]. ▶ [ARM64] Revised section "The Renesas R-Car family". ▶ [ARM64] Revised [VC.ARM64.Startup.6].
V2.1.28	2020-08-14	PROPOSED	<ul style="list-style-type: none"> ▶ [ARM64] Revised section "Interruption by higher exception levels". ▶ [ARM64] Revised [VC.ARM64.Startup.3]. ▶ [ARM64] Revised section "Reliable execution environment". ▶ [ARM64] Revised section "Memory management unit". ▶ [ARM64] Revised section "Interrupt controller". ▶ [ARM64] Revised section "Hardware-specific mechanisms". ▶ [ARM64] Added [VC.ARM64.Startup.14]. ▶ [ARM64] Revised section "The Renesas R-Car family". ▶ [ARM64] Revised [VC.ARM64.Startup.6]. ▶ [TRICORE] Merged TC275 and TC277 to TC27XT. ▶ [TRICORE] Update references to AURIX TC3XX safety manual v1.09. ▶ [TRICORE] Update references for TC36XD. ▶ Added explicit note about usage of MK_USE_TRACE.
V2.1.29	2020-08-26	PROPOSED	<ul style="list-style-type: none"> ▶ [TRICORE] Reworked inspection findings for TC39XX, TC36XD and TC38XQ.
V2.1.29	2020-08-28	RELEASED	The document has passed its verification; see [SM2INSP21] .
V2.1.29-MicroOs2.0_-ARM64_-inspection_2020cw24	2020-11-03	PROPOSED	<ul style="list-style-type: none"> ▶ [ARM64] Added handling of MK_ARM64TmuSpinlock. ▶ [ARM64] Rework after inspection.
V2.1.30-MicroOs2.0_-ARM64_-inspection_2020cw24	2020-11-09	PROPOSED	[ARM64] Rework after inspection.
V2.1.30-MicroOs2.0_-	2020-11-10	RELEASED	The document has passed its verification; see [SM2INSP20]

Version	Date	State	Description
ARM64_- inspec- tion_2020cw24			
V2.1.31	2020-11-26	PROPOSED	<ul style="list-style-type: none"> ▶ [ARM] Updated references to hardware safety manual for RCARM3NCR7. ▶ [TRICORE] Added derivative TC39XQ. ▶ [TRICORE] Update references to AURIX TC3XX User's Manual v1.6.0. ▶ [ARM64] Reintegrated inspection branch MicroOs2.0_-ARM64_inspection_2020cw24. ▶ [TRICORE] Removed [VC.TRICORE.Spinlock.1]. ▶ [ARM64] Rework based on current SAR: this includes a new part about SError interrupts, the new [VC.ARM64.Startup.15], update of [VC.ARM64.RCARGEN3.Startup.4] and sections "Hardware timers" and "Interruption by EL2/EL3".
V2.1.31	2020-12-09	RELEASED	The document has passed its verification; see [SM2INSP22] .

^aASCMICROOS-5214_RH850D1X_inspection

Table 1.1. Document history

2. Document information

2.1. Objective

The objective of this document is to provide you with all the information necessary to ensure that EB tresos Safety OS for the TriCore family is used in a safe way in your project.

2.2. Scope and audience

This manual describes the usage of EB tresos Safety OS in system applications which have safety allocations up to ASIL-D. It is valid for all projects and organizations which use EB tresos Safety OS in a safety-related environment.

The intended audience of this document consists of:

- ▶ Software architects.
- ▶ Safety engineers.
- ▶ Application developers.
- ▶ Software integrators.

The persons with these roles are responsible for performing the verification methods defined in this manual. They shall have the following knowledge and abilities:

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

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

2.3. Motivation

This manual provides information on how to use EB tresos Safety OS correctly in safety-related projects. If EB tresos Safety OS is used differently, the resulting system might not comply with the assumed requirements of EB tresos Safety OS that are defined in [Chapter 5, “Application constraints and requirements”](#). You must take appropriate actions to ensure that your safety requirements are fulfilled.

2.4. Structure

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

[Chapter 3, “About EB tresos Safety OS”](#) describes EB tresos Safety OS and the microkernel in particular and provides a list of the supported microcontrollers.

[Chapter 4, “Using EB tresos Safety OS safely”](#) describes how to use the EB tresos Safety OS safely.

[Chapter 5, “Application constraints and requirements”](#) describes the application constraints and the assumed requirements.

[Chapter 6, “Configuration verification criteria”](#) lists the criteria that you must use to verify that the EB tresos Safety OS is configured correctly for safe use. Safe use can only be determined for the *item* in which EB tresos Safety OS is used. You must perform a hazard analysis and risk assessment for the item. Refer to [\[ISO26262-3_1ST\]](#), clause 7.

[Chapter 7, “TriCore family supplement”](#) provides details of processor-specific safety considerations.

[Appendix A, “Reserved identifiers”](#) lists the program identifiers that are reserved for use by EB tresos Safety OS.

[Appendix B, “Safety of OS services”](#) lists the API services provided by the EB tresos Safety OS and the freedom from interference provided for each service.

[Appendix C, “Safety of OS callout functions”](#) lists the callout (hook) functions and the freedom from interference provided by the EB tresos Safety OS for each service.




Finally, the [“Bibliography”](#) lists the documents that are referenced in the text.

2.5. Typography and style conventions

Throughout the documentation you see that words and phrases are displayed in bold or italic font, or in Mono-space 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 <i>basic building blocks</i> 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 <i>mixed version</i> .
Arial italics	to indicate that a term is explained in the glossary	...exchanges <i>protocol data units (PDUs)</i> with its peer instance of other ECUs.

Convention	Item is used	Example
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 <code>function_name/abc-folder</code>
Monospace font (Courier)	for code	<pre>for (i=0; i<5; i++) { /* now use i */ }</pre>
Monospace font (Courier)	for function names, methods, or routines	The <code>cos</code> function finds the cosine of each array element. Syntax line example is <code>MLGetVar ML_var_name</code>
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	<code>insertBefore [<opt>]</code>
Curly brackets { }	for mandatory parameters; for command syntax with mandatory parameters (in curly brackets)	<code>insertBefore {<file>}</code>
Three dots ...	for further parameters	<code>insertBefore [<opt>...]</code>
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	<code>allowinvalidmarkup {on off}</code>

Convention	Item is used	Example
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.

3. About EB tresos Safety OS

EB tresos Safety OS is an implementation of a subset of the Operating System module from the AUTOSAR standard. You can use EB tresos Safety OS in projects where a safety integrity level up to ASIL-D is demanded. This Safety Manual describes EB tresos Safety OS specifically for the TriCore family of microcontrollers.

You can use EB tresos Safety OS to provide spatial freedom from interference between software components, regardless of their safety integrity levels.

EB tresos Safety OS consists of two parts:

- ▶ The *microkernel* with a high safety integrity level
- ▶ The [QM-OS](#) with QM integrity level

3.1. Architecture of the surrounding system

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

EB tresos Safety OS does not depend on the architecture of the surrounding hardware but it uses specific features of the TriCore family microcontroller on which it runs. For information on the processor characteristics and supported derivatives see [Section 3.1.1, “Microcontrollers in the TriCore family”](#).

EB tresos Safety OS is largely independent of the software architecture of the surrounding system. It places only one restriction on the software architecture: Components for which mutual freedom from interference is required must execute in [non-privileged mode](#) in separate threads.

3.1.1. Microcontrollers in the TriCore family

Supported processors in the TriCore family are 32-bit little-endian processors. Integer division always rounds towards zero. The stack grows down from a high memory address. There are three processor privilege levels, two of which are considered [non-privileged](#) and a [privileged-mode](#). The non-privileged modes are called *user-0 mode* and *user-1 mode* and the privileged mode is called *supervisor mode*.

Threads executing in User-0 mode are not permitted to modify core special function registers (CSFR). They are not permitted to access any peripheral hardware, nor are they not permitted to use the `disable` and `enable` instructions.

Threads executing in User-1 mode are likewise not permitted to modify core special function registers. They are permitted to access peripheral hardware subject to memory protection restrictions. They can use the `disable` and `enable` instructions unless you have explicitly forbidden it.

On processors of the TriCore family, the [Program Status Word Register](#) together with the [ICU Interrupt Control Register](#) fulfill the role of the [PSW](#) that is referred to in the hardware-independent configuration verification criteria.

The microkernel is designed for use with the following processors of the TriCore family. This table lists the characteristics of the supported derivatives.

Derivative	Core architecture	Memory protection unit	Remarks
TC22xL	TC1.6E	16 data regions, 8 code regions	none
TC23xL	TC1.6E	16 data regions, 8 code regions	none
TC27xT	TC1.6E/P	16 data regions, 8 code regions	Processors in A-Step are not supported.
TC29xT	TC1.6P	16 data regions, 8 code regions	none
TC33xL	TC1.6.2	18 data regions, 10 code regions	none
TC36xD	TC1.6.2	18 data regions, 10 code regions	none
TC37xT	TC1.6.2	18 data regions, 10 code regions	none
TC38xT	TC1.6.2	18 data regions, 10 code regions	none
TC38xQ	TC1.6.2	18 data regions, 10 code regions	none
TC39xX	TC1.6.2	18 data regions, 10 code regions	none
TC39xQ	TC1.6.2	18 data regions, 10 code regions	none

Table 3.1. TriCore derivatives supported by the microkernel

You can find details about the hardware in the corresponding core reference manuals and the microcontroller reference manuals.

3.2. Description of the microkernel

The microkernel implements a subset of the AUTOSAR Operating System module up to ASIL-D as defined in [\[ISO26262_1ST\]](#). The subset is responsible for managing the CPU and memory of an ECU as shared resources. Spatial freedom from interference between the software components is guaranteed. This means that the local and global variables of any component are free from interference by other components. In addition, the control flow, the relative priorities, and the critical sections are also free from interference. Correct configuration and use of the microkernel according to this document are preconditions to these provisions.

The microkernel also provides an interface to some other services of the AUTOSAR operating system. These services are provided by the QM-OS, an external OS module that is developed to QM standards. The interface to the QM-OS guarantees the spatial freedom from interference for software components. Most QM-OS ser-

vices can be called safely from ASIL-D contexts. The QM-OS is treated like a component of the surrounding system. The microkernel does not guarantee the functionality of the QM-OS.

To find a list of all the OS services defined by AUTOSAR and their implementation status, integrity category etc., see [Appendix B, “Safety of OS services”](#).

The microkernel can limit the execution time of a single invocation of any executable object, but this alone does not provide freedom from interference in the time domain. The microkernel does not guarantee that any specific component executes on time, nor that a component gets enough access to the CPU to perform its functions in a timely manner. If an interference in the time domain must be detected or prevented, you must use a timing protection module in combination with an independent time, such as an external watchdog monitor. The microkernel has support for measuring time intervals that can be used by such a module.

The microkernel does not provide freedom from interference in the communication domain. If you transmit safety-related information over unsafe channels, you must guarantee its integrity.

Whenever this document mentions the *freedom from interference* provided by the microkernel you must understand the term as meaning *spatial* freedom from interference.

3.2.1. Interaction with the surrounding system

The microkernel manages the executable software components of the surrounding system by means of [threads](#). Each thread can use the features of the microcontroller subject to defined privileges and restrictions. Each thread is bound to a [core](#) that is specified by the configuration. The threads that are bound to the same core execute concurrently but not simultaneously subject to the priority rules defined by AUTOSAR. If you configure your system to occupy two or more cores of a multi-core microcontroller, threads that are bound to different cores can execute simultaneously regardless of their relative priorities.

Once successfully started, the microkernel is modeless. The state of the system at any time is defined by the states of the threads that are active. The surrounding system is responsible for maintaining any operating modes that are required for the system. This surrounding system also switches the system to a safe state if a safety-critical error is detected.

The software components can interact with EB tresos Safety OS by using the public API as defined in the EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

The microkernel reports errors and protection faults to the surrounding system by means of three hook functions defined by the AUTOSAR standard: `ErrorHook()`, `ProtectionHook()`, and `ShutdownHook()`. Each of these hook functions executes in a thread to provide freedom from interference between the hook functions and the other software components. There are instances of the threads present on each configured core. The microkernel activates the hook function *on the core on which the error is detected*. This means that the hook functions can execute simultaneously on two or more cores. Your implementation of these hook functions must therefore use suitable mutual exclusion mechanisms to ensure the consistency of common data. Interrupt locks are not sufficient for this purpose.

In the very last instance, if all attempts to notify the surrounding system of a fault fail, the microkernel starts a defined shutdown thread. This thread, unless otherwise configured, executes an endless loop with interrupts disabled. This state is called the [emergency stop state](#). It is managed in the same way as any other thread. It is assumed that the lack of activity caused by executing this thread is a [safe state](#). If this is not the case, you must apply system-specific measures to recover from this state.

There is one further emergency stop state that can occur as a result of a repeated serious fault. This is the emergency stop inside the kernel, and occurs if the attempt to report a serious fault or internal inconsistency in the microkernel results in a further serious fault or internal inconsistency. It can also occur if specifically requested by the value returned from the `ProtectionHook()`.

The microkernel interacts with the QM-OS by means of a configured set of descriptor tables. The tables contain core mappings for the objects in the QM-OS along with function pointers that correspond to services defined by the microkernel. When one of these services is called, the microkernel starts the corresponding function of the QM-OS in a thread and passes up to 4 parameters from the caller to the QM-OS function. The QM-OS function executes on the core to which the QM-OS object is bound. The QM-OS is permitted to use the services provided directly by the microkernel for the purposes of managing critical sections, managing tasks and reporting errors. When a service provided by the QM-OS returns, its return status and an optional return value are passed back to the original caller of the service. Services provided by the QM-OS are listed in the EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

In order to guarantee freedom from interference, the microkernel handles all requests generated by the hardware. These hardware requests include

- ▶ [Interrupt requests](#) raised by timers and other hardware peripheral devices.
- ▶ The [system-call](#) trap.
- ▶ Other [exception](#) traps that are raised by the hardware in response to errors and other abnormal circumstances.

3.2.2. Outstanding anomalies

The EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#) contains deviations from the AUTOSAR standard. You can find information about all known issues for a particular version of EB tresos Safety OS in EB tresos AutoCore known problems [\[KNOWNPROBLEMS\]](#) corresponding to your release. It is updated regularly.

3.2.3. Backward compatibility

You can find information about backward compatibility and how to migrate from an earlier version in the EB tresos Safety OS release notes [\[RELNOTES\]](#): Section *Migrating EB tresos Safety OS*.

3.2.4. Compatibility with other systems

EB tresos Safety OS consists of the microkernel and the QM-OS. Use only the versions that are part of your release together. No other combinations are supported.

The EB tresos AutoCore Quality Statement Safety OS [\[Q-STATEMENT\]](#) documents the supported toolchain and its configuration.

3.2.5. Change control

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

3.2.6. Safety mechanism used by the microkernel

The microkernel uses and relies upon a memory protection mechanism provided by the microcontroller hardware. The memory protection hardware is programmable at run-time and can be used to permit or deny access to defined regions of memory.

Whenever the microkernel determines that a different thread shall execute, it programs the registers of the memory protection mechanism according to the memory protection configuration.

While the processor is executing in a non-privileged mode, it does not permit the memory protection hardware to be re-programmed.

Switching from a lower privilege level to a higher privilege level can only be achieved by a mechanism that enforces execution of a defined microkernel function after the switch is made. [Interrupt requests](#) and [exceptions](#) cause a switch to the highest level and execute defined microkernel functions.

It is assumed that the microcontroller hardware can be programmed to grant access in a lower privileged mode to hardware peripheral devices such as timers, communication ports, etc. This assumption allows all threads to execute using a lower privileged mode than the microkernel. It also means that no software component has write access to the memory protection hardware, thus the safety analysis is greatly simplified.

In addition, the microcontroller shall provide a [reliable execution environment](#). This implies that sporadic faults in the hardware (e.g. a bit flip in a register or in the RAM) are reliably detected by the microcontroller itself. The microkernel does not provide protection against these faults.

If you use the derivatives of the TriCore family in accordance with the corresponding safety manual from the processor vendor and in accordance with this document, EB tresos Safety OS fulfills the requirements for the memory protection mechanism in general. However, there can be derivative specific restrictions and limitations

especially concerning the requirement for the reliable execution environment. For details on processor-dependent implications and restrictions, see [Chapter 7, “TriCore family supplement”](#). The derivatives of the TriCore family are listed in [Section 3.1.1, “Microcontrollers in the TriCore family”](#).

3.2.7. Identification of files in the microkernel

The microkernel's header files begin with the prefix `Mk_`. The only exception to this rule is the header file `MicroOs.h`. The microkernel's source files that are compiled and linked as part of the application are also prefixed with `Mk_`.

EB tresos Safety OS ships a number of template functions, for example to initialize the PLL. These template functions give examples how certain functionality can be implemented for a particular derivative. The template functions are not part of the microkernel itself and are not developed for use in safety-relevant systems. The names of the template functions end with the suffix `Tpl`. You can find the template functions in the `QM` directories.

NOTE**Use of templates**

If you choose to use a template function in either unmodified form or as a basis for your own implementation, you must ensure that it fulfills the intended functionality at the desired integrity level.

3.2.8. Robustness of the microkernel

To understand how to use EB tresos Safety OS correctly, it is important to understand its limitations as well as its features. This section describes what the microkernel does not do and provides information about its robustness in various circumstances.

3.2.8.1. What the microkernel does not do

The freedom from interference provided by the microkernel covers the spatial domain. This means that the local and global variables that belong to one component cannot be interfered with by any other component unless permission is explicitly given by the configuration. It also means that the control flow of a thread cannot be interfered with, even if the thread is preempted by a thread of higher priority. When control returns to the preempted thread its context remains unchanged. This means that the interruption is transparent to the thread (apart from the time delay and the possibility that shared variables have changed).

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

The microkernel does not provide any means to prevent unintentional service calls from the application¹. If any thread erroneously places a valid service request to the microkernel, for example a task activation, the microkernel will perform this request, because from the operating system's side there is no way to distinguish between intentional and unintentional service requests. Such unintentional service requests can be detected by the application using external control flow monitoring or similar mechanisms.

The freedom from interference covers the time domain only partially. This means that the microkernel does not make any guarantee about the timely activation of threads, nor does it place any bounds on the duration of any interruption that may occur. The microkernel can place an upper bound on the processor time used by a single activation of any thread (*execution budget monitoring*), but full AUTOSAR timing protection is *not* implemented.

The microkernel does not provide freedom from interference in the communication domain.

3.2.8.2. Robustness against hardware faults

The microkernel is susceptible to hardware faults. It is assumed that the hardware uses fault detection mechanisms such as dual redundant processing (e.g. [lockstep mode](#)), [error detection and correction codes](#) (ECC) etc.

3.2.8.3. Robustness against systematic software errors

The microkernel is susceptible to systematic software errors in its own code. Although the microkernel contains numerous checks of the validity of its internal variables, it cannot be guaranteed that a systematic error in the microkernel is detected.

To guard against this risk, the microkernel is developed using the stringent practices required for ASIL-D as demanded by [\[ISO26262_1ST\]](#).

3.2.8.4. Robustness against configuration errors

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

You must verify the configuration presented to the microkernel as described in [Chapter 6, “Configuration verification criteria”](#). The configuration must meet the safety requirements of the surrounding system.

3.2.8.5. Robustness against resource conflicts

The microkernel requires exclusive use of the memory protection hardware. Access by other software is not allowed. Failure to observe this adversely affects the freedom from interference provided by the microkernel.

¹For exceptions, see the "API access protection" section of EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

The microkernel manages the shared resource of the processor (including its internal registers) such that no conflicts take place. The microkernel is able to enforce this freedom from conflicts for all threads running in [non-privileged mode](#). The microkernel may not be able to prevent threads running in [privileged mode](#) from creating conflicts.

The microkernel also requires exclusive use of the parts of the interrupt controller that direct interrupts to the cores on which the microkernel is configured to run. Depending on the system requirements it might not be practical to reserve exclusive use of the interrupt controller. If you share the interrupt controller with other cores or with triggerable hardware such as DMA controllers, you must ensure that the operation of the microkernel and your safety goals are not adversely affected by this sharing.

If you share a peripheral such as a timer with the microkernel you must ensure that the other users cannot adversely affect the microkernel.

3.2.8.6. Robustness against interrupt overload

The microkernel itself is resistant to interrupt overload. Its non-reentrant design means that it can only process one request at a time. Thus, there is no possibility that nested interrupts can cause an overflow of the microkernel's stack.

Furthermore, when an ISR is running, its interrupt source and all other interrupts of the same or lower level are blocked by the interrupt controller's programmable interrupt level.

This robustness, however, does not provide any protection for the system as a whole from interrupt overload. The microkernel does not implement the timing protection feature of arrival rate limit of AUTOSAR. It is still possible for rapidly-occurring interrupts to prevent the microkernel from giving processor time to lower-priority tasks. It is assumed that any safety implications resulting from such situations are detected by an external watchdog.

3.2.8.7. Robustness against input errors

The microkernel does not accept any data input from external sources.

Incorrect parameter values and other errors resulting from calls to microkernel services are detected and reported using the `ErrorHook()`.

NOTE



Limitation of error checking

The detection of errors in microkernel services is restricted to those errors that can adversely affect the freedom from interference provided by the microkernel. Much of the additional error checking specified by AUTOSAR has no safety relevance and is not implemented. Furthermore, there are some aspects of the error detection and handling specified by AUTOSAR that can adversely affect safety. One of these aspects is the lack of error reporting in services that do not return `StatusType`. In the microkernel, *all* errors that are detected are reported through the `ErrorHook()`.

4. Using EB tresos Safety OS safely

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

- ▶ Read and understand the safety manual from the processor vendor.
- ▶ Fulfill all requirements that are stated in the safety manual from the processor vendor.
- ▶ Read and understand the reference manuals for the microcontroller family and derivative that you are using; in particular, those sections that are relevant for the hardware features that you are using.

For details on using TriCore family processors safely, see [Chapter 7, “TriCore family supplement”](#).

EB tresos Safety OS is developed as a safety element out of context (SEooC). Therefore, Elektrobit Automotive GmbH (EB) assumes that the environment meets particular requirements so that EB tresos Safety OS behaves appropriately and safely. See these assumptions in [Chapter 5, “Application constraints and requirements”](#).

Each release of EB tresos Safety OS is tested with a particular MCU derivative. The exact identifier and revision of the derivative is stated in the EB tresos AutoCore Quality Statement Safety OS [\[Q-STATEMENT\]](#). Do not use EB tresos Safety OS on a different derivative.

4.1. Applicability of this document

This document does not refer to a specific version of EB tresos Safety OS. To ensure that you have the correct version of this document, refer to the EB tresos AutoCore Quality Statement Safety OS [\[Q-STATEMENT\]](#) that accompany the software you are using.

4.2. Prerequisites

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

You must assign the software components that need to be free from interference from each other to *separate* OS-objects. [Figure 4.1, “Separation of OS-objects based on the integrity level”](#) shows an example. This means that you *must* assign software components of different integrity levels to separate OS-objects. It is also highly recommended to assign different software components of the same integrity level to different OS-objects. This prevents fault propagation between different control functions that might be integrated on the same ECU.

Memory regions associated with tasks, ISRs, and hook functions define memory protection boundaries between these threads. *Memory partitions* implement this association. The general principle is that tasks, ISRs, and hook functions do not share writable memory regions in order to ensure spatial freedom from interference.

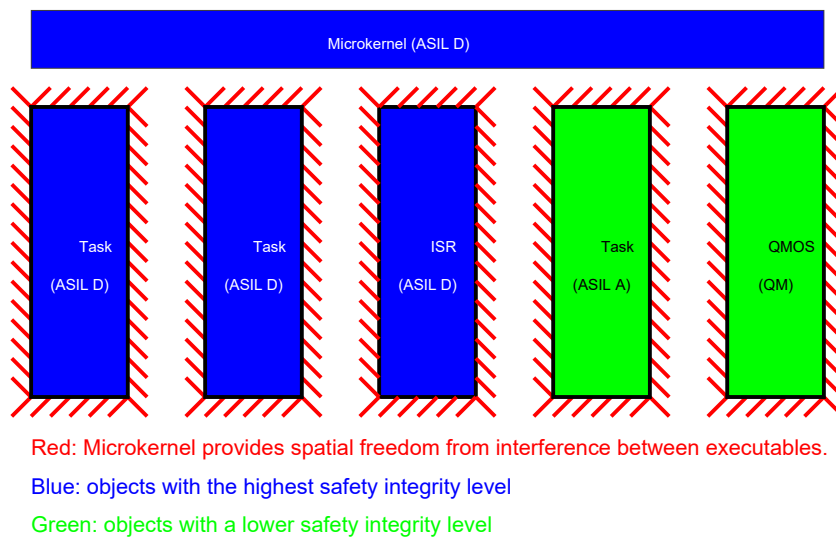


Figure 4.1. Separation of OS-objects based on the integrity level

4.3. Installing EB tresos Safety OS

To install EB tresos Studio, the microkernel plugin, the [QM-OS](#) plugin and all other plugins that you need, follow the instructions in the EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

4.4. Verifying the integrity of the microkernel sources

EB tresos Safety OS delivery contains checksums with MD5 hashes of all files that are part of the shipped package. You can find these files in the `$TRESOS_BASE/checksums` folder. The checksums allow you to verify the integrity of the microkernel sources and thus to detect if any file got corrupted, e.g. by a failing hard drive or an accidental change.

You can process the MD5 checksums automatically with a **md5sum** tool. To perform the check, take the following steps:

- ▶ Set the environment variable `MD5SUMPATH` to the folder that contains the **md5sum** tool.¹
- ▶ Execute `launch.bat` in the project's `util` folder.
- ▶ Call **make checkmd5** to automatically verify the integrity of the shipped files.

¹You need a GNU-compatible `md5sum` tool. For further details, see <http://www.gnu.org/software/coreutils/manual/coreutils.html>.

If all files are intact, the message `Verification successful. All files are OK.` appears.

If any file is corrupt, the message `*** VERIFICATION FAILED ***` appears.

WARNING

Integrity of delivered files



Use EB tresos Safety OS only if all files are intact.

In case of corrupted files, reinstall the package that contains EB tresos Safety OS, and watch out for any warnings that appear during this process. Contact EB Product Support and Customer Care Team if the verification still fails.

4.5. Using EB tresos Studio

This section describes how to configure the microkernel with the usage of EB tresos Studio.

4.5.1. Configuring the operating system module

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

- ▶ Ensure that you assign the correct priorities to your tasks.
- ▶ Ensure that you assign the correct interrupt levels to your ISRs.
- ▶ Ensure that you select the `OsTaskSchedule` attribute appropriately for each task.
- ▶ Ensure that you configure the correct resources for each task and ISR that uses resources. This is particularly important for internal resources because they are automatically acquired when a task runs. This means that an error check for missing resources at run-time is not possible.
- ▶ Ensure that you enable the `OsTrusted` attribute only for those applications that really need [privileged mode](#). This attribute affects all tasks and ISRs that belong to the application. The tasks and ISRs are executed in privileged mode. If the hardware is configured correctly, privileged mode is needed only for a very few exceptional cases.

It is strongly recommended that you execute all OS-objects, for which this is possible, in [non-privileged mode](#). Threads that execute in privileged mode inherit the highest integrity level of the system.

4.5.2. Generating the operating system module

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

4.5.3. Verifying generated files

EB tresos Studio and its plugins are not developed according to the standards for safety-related projects. Therefore, you must verify, that the configuration files that the [OS generator](#) creates for the microkernel, create the system that is specified.

The OS generator creates three files:

- ▶ `Mk_gen_user.h`
- ▶ `Mk_gen_config.h`
- ▶ `Mk_gen_global.c`

You must verify that the content of these files satisfies all the criteria listed in [Chapter 6, “Configuration verification criteria”](#).

NOTE



Configuration by hand

It is possible to configure the microkernel without the use of EB tresos Studio. If you wish to create your own configuration files, follow the instructions described in the microkernel user's guide. The hand-generated configuration must still satisfy the verification criteria for microkernel configuration files. If the configuration files do not satisfy all the verification criteria, the resulting system might still be safe, but you must assess the safety yourself.

4.6. Implementing your system

The details of how to implement the remaining software in your system are beyond the scope of this Safety Manual. There are however a few rules that you must follow, to ensure that you use the microkernel safely. The rules are given in the following sections.

4.6.1. Protect against non-lockstep cores

The microkernel depends on a reliable execution environment. For more information on that, see [Assumptions.ReliableExecutionEnvironment](#). There are multi-core processors which only provide a partially reliable execution environment as only some of their cores are lockstep cores while others are not.

If you use a non-lockstep core on such a microcontroller in the microkernel, a fault on this non-lockstep core can lead to the following consequences on other cores including the lockstep cores:

- ▶ Another core is shut down.

- ▶ Unintended services are invoked on another core.
- ▶ Another core can wait indefinitely for a result of the non-lockstep core.
- ▶ Spinlocks are corrupted.

While most of these consequences do not violate microkernel safety requirements, they can violate safety requirements of your application. Therefore, you must make sure to use appropriate safety mechanisms to identify and handle such errors if you use non-lockstep cores. This can be for example control flow and deadline monitoring to prevent indefinite waiting for service results but can also require other safety mechanisms.

NOTE



Corrupted spinlocks are a safety risk

If you have safety requirements for spinlocks, you must make sure that those spinlocks cannot be accessed by non-lockstep cores. That means:

- ▶ These spinlocks shall be used to synchronize lockstep cores only.
- ▶ You must prevent access to these spinlocks from non-lockstep cores. You can do this e.g. by configuring a bus MPU in the way that the non-lockstep cores do not have write access to the spinlock memory.

4.6.2. Avoid symbols reserved by EB tresos Safety OS

All symbols that start with `MK_`, `Mk_`, and `mk_` are reserved for the exclusive use of the microkernel. All symbols that start with `OS_`, `Os_` and `os` are reserved for the exclusive use of the QM-OS. All symbols that start with `E_OS_`, `OSServiceId_`, `OSError_`, `OSTICKSPERBASE_`, `OSMAXALLOWEDVALUE_` and `OSMINCYCLE_` are reserved for the exclusive use of AUTOSAR-OS.

In addition to these symbol prefixes, the symbols listed in [Table A.1, “Identifiers reserved for use by AUTOSAR OS”](#) are reserved for use by EB tresos Safety OS as specified by the AUTOSAR Operating System specification. The symbols listed in [Table A.2, “Identifiers reserved for use by EB tresos Safety OS”](#) are reserved for implementation by EB.

You must not define any of the symbols mentioned above in your application. This prohibition includes the naming of objects in your configuration.

4.6.3. EB tresos Safety OS services

The table in [Appendix B, “Safety of OS services”](#) lists all the services provided by an AUTOSAR-compatible operating system module along with some additional services defined by EB. The second column of this table gives the integrity category for each service. You can find a description of the different integrity categories below. [Table 4.1, “Overview of safety integrity categories”](#) contains a summary.

Integrity category	Correctness of functionality verified up to	Spatial freedom from interference enforced
1	ASIL-D	yes
2	QM	yes
3	QM	no / only partially

Table 4.1. Overview of safety integrity categories

4.6.3.1. Services with integrity category 1

You may safely call a service with an integrity category of 1 from any of your tasks, ISRs or other threads whose integrity level is not greater than the integrity level of the system as a whole. For all practical purposes this means you may call these services from any thread.

The functionality of these services is developed to standards appropriate for ASIL-D. This means that if you call the service correctly, it performs as specified. For example a call to `ActivateTask()` either places the task in the correct position in the thread queue or reports an error. If the task to be activated is on the same core as the caller and has a higher priority than the current priority of the caller, the task runs before `ActivateTask()` returns to the caller.

4.6.3.2. Services with integrity category 2

You may safely call a service with an integrity category of 2 from any of your tasks, ISRs or other threads whose integrity level is not greater than the integrity level of the system as a whole. For all practical purposes this means you may call these services from any thread.

The functionality of these services is provided wholly or partly by QM software, so the functionality cannot be guaranteed to any safety integrity level. In particular, it is possible that the service might return incorrect information or might never return, e.g. if the QM software locks up in an endless loop for some reason.

In either case, the microkernel guarantees up to the safety integrity level of the system that the execution of the service does not interfere with the local or global variables of the caller or of any other of your software components.

4.6.3.3. Services with integrity category 3

You must not call a service with an integrity category of 3 from any of your tasks, ISRs or other threads that have a safety integrity level allocation. These services are wholly or partly implemented as QM functions that

are called directly from your thread. This means that they have the freedom to modify or corrupt any local or global variables that the caller itself can modify.

4.6.4. Writing callout functions

The AUTOSAR standard defines a set of [callout functions](#), or hook functions, that the operating system calls in response to defined events. In a microkernel-based system, some of these callout functions run in their own threads rather than being called directly. For a list of the callout functions and their calling methods, see the table in [Appendix C, “Safety of OS callout functions”](#).

4.6.4.1. Callout functions that are called directly

The callout functions that are called directly must have a safety integrity level equal to the highest level of the system. They are executed *within* the microkernel. This means that they run in [privileged mode](#). If the callout function is called before the memory protection hardware is initialized, it runs completely without memory protection. If it is called afterwards, the memory protection settings of the microkernel are in place. For details of where each callout function is called, see [Appendix C, “Safety of OS callout functions”](#). As a consequence, the callouts are not prevented from accessing data and/or peripherals of at least the microkernel or even the whole system. The microkernel does not enable execution budgets for directly called callouts. Additional restrictions apply:

- ▶ The callouts must never call AUTOSAR services.
- ▶ They must not provoke any exception or system call and must not change the CPU state so that the processor mode is changed or flags that disable interrupts or exceptions are modified.

The microkernel is sensitive to errors from callout functions which are called directly. Such an error may cause the system to fail immediately or later. You must take appropriate measures to prevent such errors.

4.6.4.2. Callout functions that are called in a thread

The callout functions that are called in a thread are subject to the same conditions as other threads in the system, such as tasks and ISRs. You can allocate any safety integrity level that is considered appropriate for the system to these callout functions. They may call the services listed in [Appendix B, “Safety of OS services”](#) subject to their safety integrity level.

You can separately enable or disable the threaded callout functions in EB tresos Studio. EB tresos Studio creates a thread configuration for each callout function that you have enabled. The default configuration creates a thread that runs in non-privileged mode. It is possible to modify this configuration to give wider access rights,

provided that the callout function is assigned a safety integrity level in accordance with those rights. You must develop the callout function to the standards appropriate for its safety integrity level.

4.6.4.3. Alarm callback functions

The AUTOSAR standard only permits alarm callback functions in scalability class 1, but EB tresos Safety OS does not enforce this limitation because they too run in a thread. However, unlike the threaded hook functions, alarm callback functions are called directly by the QM-OS. This means that they share the same memory partition as the QM-OS. They are not free from interference by the QM-OS and the QM-OS is not free from interference by alarm callbacks. If you want to perform safety-related functionality in an alarm callback function, you must take measures to ensure the appropriate behavior on system level.

You may add extra memory regions to the memory partition of the QM-OS, or place global variables in the memory region that contains the data that belongs to the QM-OS. However, this data is unprotected from the QM-OS, and software components with a safety integrity allocation must not rely on this data without verification.

4.6.5. Starting the microkernel

To start the microkernel after reset, call or jump to the “function” `MK_Entry2()` on every configured core. On multi-core processors that do not start all cores automatically on reset, this means that your start-up code is responsible for starting the cores that you configure. The microkernel ships an example implementation called `MK_Entry()` that runs after reset and performs sufficient initialization of the hardware to permit tests to execute.

You are allowed to run software of any safety integrity level before `MK_Entry2()`. If you implement the software according to the guidelines in [Section 7.12, “Start-up preconditions”](#), the microkernel starts correctly. If you do not follow the guidelines, the behavior of the microkernel is undefined.

However, if the software that executes before `MK_Entry2()` has not been developed to the safety integrity level for your system, you must not make any assumptions about initialization takes place before `MK_Entry2()`. This means that you must repeat this initialization or verify that it has been performed correctly. This is particularly important for the initialization of memory regions, which must be initialized *after* the bus memory protection hardware has been initialized.

[Section 7.12, “Start-up preconditions”](#) defines preconditions of the microkernel's start-up sequence. You can fulfill the preconditions before `MK_Entry2()` is called, provided that you ensure freedom from interference. Alternatively, the microkernel calls two callout functions, `MK_InitHardwareBeforeData()` and `MK_InitHardwareAfterData()`, in which you can initialize the hardware, or verify that the hardware has been correctly initialized. These callout functions are called directly by the microkernel; see [Section 4.6.4.1, “Callout functions that are called directly”](#). The following list describes the callout functions:

```
void MK_InitHardwareBeforeData(void)
```

This function is called on all configured cores before any data is initialized, therefore it is a [restricted C-function](#). It may not rely on any global variables being initialized. It may also not rely on the global variables' values being preserved after the function returns, unless you place the variables in uninitialized memory regions.

You can use it to setup the PLL, RAM or perform other initializations to ensure that the actual RAM initialization (performed later on by the microkernel) runs at the best possible speed.

```
void MK_InitHardwareAfterData(void)
```

This function is called on all cores after global data has been initialized.

You can use this function to initialize drivers or perform other activities.

The microkernel ships an example implementation of both `MK_InitHardwareBeforeData()` and `MK_InitHardwareAfterData()` for the relevant derivative. You must verify the correctness if you use them within your application in either unmodified form or as a basis for your own implementation.

4.6.6. Shutting down the microkernel

You can shut down the microkernel on a single-core microcontroller by calling `ShutdownOS()`. After your `ShutdownHook()` terminates itself (usually by return), the microkernel schedules the *shutdown-idle* thread that runs with interrupts disabled. Thus the microkernel can no longer react and is waiting for power-down or reset of the microcontroller. This type of shutdown is called a *single-core shutdown*.

A single-core shutdown also occurs when

- ▶ `ProtectionHook()` returns `PRO_SHUTDOWN`.
- ▶ `ProtectionHook()` returns `PRO_PANIC`.
- ▶ `ProtectionHook()` returns an invalid return value.
- ▶ You have not enabled the `ProtectionHook()` and a protection fault occurs.
- ▶ The microkernel detects a serious fault or internal inconsistency.

You can shut down the entire microkernel on all cores of a multi-core microcontroller by calling `ShutdownAllCores()`. This API has the effect of simultaneously calling `ShutdownOS()` on all cores.

On a multi-core microcontroller, a single-core shutdown only shuts down the core on which it occurs, leaving the other cores running. When the microkernel finally reaches the shutdown-idle thread it is no longer able to react to inter-core requests of any kind.

If another core continues to send requests to the core that has reached its shutdown-idle thread, it will eventually fill the message queue and will wait until the target core handles some of the messages. This can never happen, and so the microkernel will wait forever, inside the microkernel, with interrupts disabled.

If your requirements specify that it shall be possible to shut down a single core without affecting the other cores, you must ensure that the system never reaches the shutdown-idle thread. The simplest way to achieve this is to implement your `ShutdownHook()` function so that it enters an endless loop and never returns. As long as your `ShutdownHook()` is running, the microkernel on that core will handle all incoming API requests. Tasks will be activated, but will never run. In the case of repeated requests, the `ErrorHook()` will get activated on the core that has shut down.

If QM-OS requests are sent to the core that has shut down, they will be placed in the QM-OS job queue but the QM-OS thread will never execute. In the case of synchronous requests, the status code from the QM-OS service will never get sent back to the caller, so the caller will wait - potentially forever, unless the executable has an execution budget. You must therefore ensure that your executables do not attempt to send synchronous QM-OS requests to a core that is shut down.

NOTE



Shutdowns due to internal errors can corrupt spinlocks

If a shutdown is triggered by the microkernel itself, e.g. because it detected a serious fault, internal microkernel data could be corrupt. This can lead to the situation that the core which shuts down, releases a spinlock that was acquired by another core. This effectively means that the critical section of the other core gets corrupted.

Such a shutdown can also result in the situation that the core which shuts down, does not release the spinlocks which it holds. Other cores therefore cannot acquire these spinlocks.

All in all you can no longer rely on spinlocks that were in use while the core was shut down. In this case you should therefore shut down the other cores as well and reset the microcontroller.

4.6.7. Testing the memory protection mechanism

To guarantee freedom from interference on data level, the microkernel depends on the correct functionality of the memory protection mechanism of the CPU. Faults may arise from incorrect configuration but also if the hardware degrades past its lifetime.

To check the functionality of the memory protection mechanism, use the following procedure:

- ▶ Create a non-trusted task that runs in non-privileged mode.
- ▶ Inside the task, try to write to a memory location that does not belong to the task's accessible data (or the shared data of the [OS-Application](#) of the task). For example, try to change the value of the microkernel's private variable `MK_c0_initTestData`². Under normal conditions the CPU does not write the data and the microkernel calls the protection hook. Therefore, if the non-privileged task is able to change the data in question, the memory protection mechanism does not work correctly.

²The microkernel uses the variable `MK_c0_initTestData` only during start-up. If a failing memory protection mechanism results in a change of the value, the microkernel does not malfunction.

- If the test fails, you must ensure that a safe state is reached, for example by shutting down the microkernel.

You can adapt this procedure for use in a periodical memory protection check if the hardware safety manual demands such a check.

NOTE **This test is only a plausibility check of the memory protection mechanism.**



4.6.8. Accuracy of simple schedule table

A counter that is advanced automatically by a hardware timer interrupt relies on the regular occurrence of the timer interrupt in order to proceed correctly. Disabling interrupts in the application leads to variations in the time interval between any two interrupts.

The microkernel executes with interrupts disabled from the point where a system call, interrupt, or exception occurs to the point where the calling thread (or another thread) is resumed. During this time the timer interrupt is prevented from occurring and runs a short time later.

Disabling interrupts in the tasks and ISRs of your application also results in the timer interrupt being delayed. This applies whether you use the AUTOSAR interrupt management services such as `SuspendOSInterrupts()` or some processor-specific *fast locking* mechanism.

Acquiring a resource by calling `GetResource()` may also delay the timer interrupt until the resource is released (`ReleaseResource()`). This depends on the relative priorities of the timer interrupt and the ISRs (if any) that share the resource. Even if the resource does not cause the interrupt to be delayed, a periodic task might get delayed sufficiently to cause an activation failure at a subsequent timer interrupt.

Hook functions such as the `ErrorHook()` and `ProtectionHook()` also run with disabled interrupts and cause the timer interrupt to be delayed.

Delays caused by critical sections cause the start times of periodically activated tasks to deviate from their design value. The deviations are positive or negative by an amount equal to the longest delay achieved. On average over a long time the sum of all the deviations is close to zero, provided that all task activations have succeeded.

If the delay caused by a critical section exceeds a certain system-specific time, the attempt to activate a task may fail because the task is still executing from a previous activation. If this behavior occurs, the task is not activated but no error is reported.

If the delay is long enough to cause the next interrupt to become due before the pending interrupt is processed, the behavior of the SST is undefined. If, as part of its normal processing, the interrupt routine detects that

the next scheduled time is already in the past, it invokes the `ProtectionHook()`. However, this detection depends on the characteristics of the hardware and is not guaranteed. The hardware may simply lose the interrupt without any error being reported. The SST *does not* monitor the interrupt interval actively.

You must therefore carefully analyze the worst-case timing characteristics of your system to ensure that the following verification criteria are met.

[VC.SSTDELAY.1]

Verify that your system is able to tolerate the jitter and other effects that are caused by the critical sections in the system.

[VC.SSTDELAY.2]

Verify that no critical section in your system causes a timer interrupt to be delayed such that the next timer interrupt becomes due before the previous interrupt has finished execution.

[VC.SSTDELAY.3]

Verify that you have configured the `ProtectionHook()` feature, and provided a `ProtectionHook()` function that reacts correctly to the *in-the-past* error `MK_E_INTHEPAST`.

The *correct* reaction is system-specific. For example, you could shut down the system, or stop and restart the affected counter, or reset the timer hardware such that the next interrupt occurs at some defined time in the future.

[VC.SSTDELAY.4]

Verify that your system uses a timing protection mechanism to detect jitter and other effects that exceed your design requirements.

4.7. Writing the linker script

For information on writing a linker script, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#) and the documentation for your compiler and linker.

It is recommended to place non-writeable regions between writeable regions of any given partition. This practice increases the likelihood of detecting pointer overruns that may result from a programming error.

The standard practice is to place all stack regions together. This practice achieves the separation goal for stacks because a thread (and the microkernel too) only has write access to a single stack. However, the same advice applies to data regions.

4.8. Building your system

For information on building your system, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

4.9. Verifying the binary image

After you build the final binary image that you want to use on the target hardware, you must perform some verification steps to ensure that the binary image really contains what you intend. These steps are additional to any testing and verification activities that are normally required for safety-related systems.

The steps are described in the following sections.

4.9.1. Ensure that the microkernel contains no low-integrity software

There are some additional features - instrumentation, development aids, etc. - that you can optionally link into the microkernel. These features are not developed for use in safety-relevant systems.

The microkernel uses a special naming scheme for low-integrity features that it calls. If the symbol table of your final binary image contains any function symbols that begin with the sequence `mk_qm` in any combination of lower- and upper-case, your system contains low-integrity software. Likewise, if a function symbol that ends with `Tmpl` is present, you linked one of the functions that contains an example implementation.

If your binary image contains functions that match these patterns, you must not use it in production.

An examination of the map file or the symbol table of the ELF file indicates the presence of unsafe software. When you use GNU Binutils,³ you can use a command such as **`nm my-image.elf | grep -i mk_qm`** to detect that unsafe functions from EB tresos Safety OS are present in the resulting binary. You can use **`nm my-image.elf | grep -i tmpl`** to detect if any function of the shipped example implementation is linked with the microkernel by accident. If the microkernel contains no development software, the commands should produce no output.

NOTE



Plausibility check

The command given above produces no output if the binary file does not contain microkernel symbols. To ensure that the binary file does contain microkernel symbols, you should use the command **`nm my-image.elf | grep -i mk_`** first.

4.9.2. Ensure that code is correctly placed

The exact criterion for the correct placement of code depends on the processor and is defined in the processor supplement, [Section 7.7, “Code placement”](#).

³ See <http://www.gnu.org/software/binutils/>.

4.9.3. Ensure that variables are correctly placed

An incorrect linker script, or a fault in the linker, could result in variables which are placed in the wrong memory regions or in memory-region boundaries which are incorrectly defined. Therefore, you must verify the correct placement of variables and region boundaries.

One way to do this is to generate a symbol table sorted by address. The linker's map file might contain such a symbol table. Another way to do this is to use a command such as **nm my_image.elf | sort > my_image.symbols.sorted**.

You must check the sorted symbol table to ensure that *all* the symbols that belong in that region, and *only* those symbols, lie between the region's start and limit addresses. You must perform the check for each memory region. The start and limit addresses for a region called `foo` are `MK_RSA_foo` and `MK_RLA_foo` respectively. The criterion is `MK_RSA_foo <= bar < MK_RLA_foo` for each symbol `bar` that is part of the region.

Do not rely on the order of symbols in the sorted symbol table as the criterion for correctness because the order of symbols with the same address is defined by the names of the symbols.

You must align the regions correctly and you must place sufficient space between the regions to satisfy the properties of the target processor's memory protection hardware (see [Section 7.4, "Memory protection unit"](#)).

4.9.4. Ensure that the representation of the memory regions is correct

To ensure that the regions defined by the symbols in the symbol table are exactly as defined in the symbol table, you must check the memory region table that the microkernel uses.

Locate the memory region table in read-only memory to avoid unauthorized changes at run-time. To ensure this, you can either use ROM, or RAM which is write-protected by the memory protection mechanism.

Create a readable dump of the memory region table. You can use your debugger for this. The array you need to verify is called `MK_memRegion`. It is an array of `mk_memoryregion_t` data structures each of which contains the fields `mr_startaddr`, `mr_bstartaddr`, `mr_limitaddr`, `mr_idata`, `mr_fillpattern` and `mr_initcore` and, depending on the CPU family, `mr_hwextra`.

The fields `mr_startaddr` and `mr_limitaddr` are important for memory protection purposes. You must verify that the memory region table contains the correct values. To do so, compare the memory region with the configured region table in the file `Mk_gen_config.h`.

The following verification criteria apply to the placement of the symbols:

- ▶ Verify that `mr_startaddr` has the proper alignment for your hardware. See [Section 6.2.15.3, "Verification criteria for memory regions on TriCore family processors"](#).

- ▶ Verify that each object that is assigned to a memory region has a symbol whose address that is greater than or equal to the start address `mr_startaddr`.
- ▶ Verify that each object that is assigned to a memory region lies entirely inside the memory region. This means that the object's address plus the size of the object is less than or equal to the limit address `mr_limitaddr`.
- ▶ Verify that there are no unintended overlapping regions. The global read-only region is one example of an overlapping region that is intended.

The fields `mr_idata`, `mr_bstartaddr`, `mr_fillpattern` and `mr_initcore` control, if and how the microkernel should initialize the memory region during start-up. If a memory region contains C data objects, the explicitly initialized objects shall be placed at the start of the region, followed by the implicitly initialized objects.

The following verification criteria apply to all regions:

- ▶ Verify that `mr_startaddr` is less than or equal to `mr_bstartaddr` unless `mr_bstartaddr` equals 0.
- ▶ Verify that `mr_startaddr` is less than or equal to `mr_limitaddr`.
- ▶ Verify that `mr_bstartaddr` is less than or equal to `mr_limitaddr`.

The following verification criteria apply to uninitialized memory regions:

- ▶ Verify that `mr_bstartaddr` is 0.
- ▶ Verify that `mr_idata` is 0.

The following verification criteria apply to initialized memory regions (containing C objects that are either explicitly initialized or implicitly initialized):

- ▶ Verify that `mr_initcore` is the core on which the initialization shall be performed.
- ▶ For memory regions that lie in physically separate memory that is mapped to the same address on all cores (for example, program scratchpad), verify that `mr_initcore` is (-1).
- ▶ Verify that `mr_idata` contains the ROM start address where the initial values of the explicitly initialized objects have been placed. If the region does not contain any explicitly initialized objects, `mr_idata` is 0.
- ▶ Verify that `mr_bstartaddr` contains the address of the portion of the memory region that contains the implicitly initialized objects. If the region does not contain any implicitly initialized objects, `mr_bstartaddr` is either 0 or is equal to `mr_limitaddr`.
- ▶ For each explicitly initialized object, verify that the offset of the initial value relative to `mr_idata` equals the offset of the object relative to `mr_startaddr`.
- ▶ Verify that `mr_fillpattern` is 0.
- ▶ Verify that the initialized memory region does not lie in the memory-mapped IO-space of peripherals.

Stack regions can be either initialized or uninitialized. The following verification criteria apply to initialized stack regions:

- ▶ Verify that `mr_idata` is 0.

- ▶ Verify that `mr_bstartaddr` is equal to `mr_startaddr`.
- ▶ Verify that `mr_fillpattern` is the fill pattern expected by your stack monitoring functions.

4.9.5. Ensuring the integrity of the binary image

If the process of transferring the created binary image to the target ECU is faulty, the executed code on the target MCU as well as the initial values for constants do not match the verified application. You must verify that the binary image has been correctly programmed.

Partial faults like single-bit errors that happen during the transfer can cause the memory protection configuration data to grant unexpected access rights to a task and therefore undermine the freedom from interference. The same is true for errors in the code of the microkernel. Such errors may not be visible immediately.

To guard against the aforementioned faults, you must verify the integrity of the binary image. To achieve this, create a checksum of the content of the binary image. To do so, use a suitable algorithm (e.g. MD5, SHA256) and place it into a constant. During run-time, calculate the checksum again and verify that it matches the precalculated one.

You should perform the online verification either before you start the microkernel or during early start-up with the use of the [callout function](#) `MK_InitHardwareBeforeData()`. If the verification fails, ensure that the microkernel is not started, e.g. by entering an infinite loop. Use `MK_StartupPanic()` for this purpose.

You must also ensure that an intended update of the ECU software has actually taken place, because if an attempted update fails completely, the ECU may contain an older version of the software that still appears to be valid. Such verification is beyond the scope of this document.

4.10. Implications of the compiler settings for use with the microkernel

The compiler, the version, and the exact options that EB supports for a particular release are specified in the corresponding EB tresos AutoCore Quality Statement Safety OS [\[Q-STATEMENT\]](#). The microkernel is tested extensively using these options.

WARNING



Use only tested compilers

Do not use a different compiler, a different version of the compiler or different options than specified in the quality statement.

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



4.11. Steps outside the scope of EB

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

5. Application constraints and requirements

The application constraints and requirements are not relevant to the microkernel. It is assumed that the application requires the services and mechanisms provided by the microkernel and defined in the previous chapters.

The definition of the microkernel as a Safety Element out of Context (SEooC) as well as the assumed requirements for the microkernel and for the hardware are detailed in the following sections.

5.1. Safety Element out of Context (SEooC) definition

EB tresos Safety OS is developed as a Safety Element out of Context (SEooC) according to [\[ISO26262-10_1ST\]](#).

5.1.1. Functional scope of the SEooC

5.1.1.1. Provided functionality

The functional scope of the SEooC is defined by the requirements as specified in [Section 5.3, “Level 1 requirements”](#) and [Section 5.4, “Level 2 requirements”](#). All requirements that have safety class ASIL-D are within the scope of the safety-related part of EB tresos Safety OS.

WARNING

The requirements listed in these sections may not be sufficient to fulfill all safety requirements imposed on the software architecture of the entire system. Depending on the application, additional safety mechanisms which are not part of EB tresos Safety OS may be necessary to guarantee system safety.

NOTE

EB tresos Safety OS does **NOT** provide a safety mechanism itself, but you can use it to implement one.

5.1.1.2. Assumed hardware functionality

EB tresos Safety OS assumes that certain functionality is provided by the hardware. These assumptions are listed in [Section 5.5, “Assumptions”](#).

5.1.1.3. Assumed employment

EB tresos Safety OS assumes that you employ the operating system according to the specification given in the safety manual.

5.1.1.4. Assumed compiler confidence level

EB tresos Safety OS assumes that you use a compiler in a version and with compiler settings that Elektrobit Automotive GmbH supports for a particular release.

EB tresos Safety OS further assumes that you take appropriate measures to create confidence in the compiler (e.g. tool qualification).

5.1.1.5. Assumed external functionality

EB tresos Safety OS

- ▶ provides functionality to ensure **PARTIAL** freedom from interference on the temporal level
- ▶ does **NOT** provide functionality to ensure freedom from interference on the communication level

To ensure freedom from interference on these levels it is assumed that you use additional measures within the entire software architecture surrounding EB tresos Safety OS. Potential measures are

- ▶ a software component providing control flow and deadline monitoring for the supervision of the temporal domain, and/or

- ▶ a software component protecting the communication within the ECU and to other ECUs (end-to-end protection)

The type and implementation of these protection mechanisms depend on the architecture of the application that is developed on top of EB tresos Safety OS.

5.1.2. Implementation scope of the SEooC

The safety-related part of EB tresos Safety OS consists of the following components:

- ▶ Source code in the `lib_src`, `src` and `include` subdirectories of the `MicroOs_TS_<ID>` plugin as installed in EB tresos Studio.
- ▶ The microkernel configuration, written according to the safety manual.
- ▶ The *quality statement*, which contains the following information on EB tresos Safety OS qualification:
 - ▶ the compiler and compiler version,
 - ▶ the compiler options and
 - ▶ the processor derivative used for testing.

The quality statement is provided for each release, platform, and derivative of EB tresos Safety OS. See [\[Q-STATEMENT\]](#).

The EB tresos Safety OS generator is not part of the SEooC. The generated configuration is considered as a template. You need to verify it manually according to the safety manual.

5.2. Definitions

This section defines the processes and terms used in [Section 5.3, “Level 1 requirements”](#) and [Section 5.4, “Level 2 requirements”](#).

5.2.1. Process words

This section defines all *processes*.

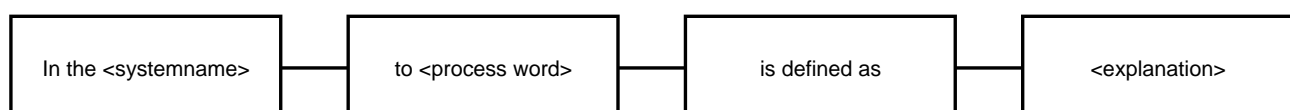


Figure 5.1. Scheme for defining a process

Process	Definition
to ensure	In the EB tresos Safety OS <i>to ensure</i> is defined as taking measures that something is possible only in a defined way.
to program	In the EB tresos Safety OS <i>to program</i> is defined as configuring hardware components during runtime.
to protect	In the EB tresos Safety OS <i>to protect</i> is defined as preventing unauthorized access .
to provide	In the EB tresos Safety OS <i>to provide</i> is defined as making something available.
to support	In the EB tresos Safety OS <i>to support</i> is defined as contributing to making something available.

Table 5.1. Definition of processes

5.2.2. Term definitions

This section defines all *terms*.

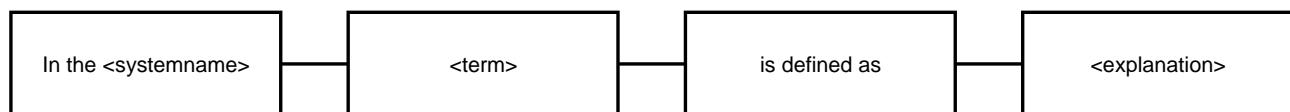


Figure 5.2. Scheme for defining a term

Term	Definition
concurrent execution	In the EB tresos Safety OS <i>concurrent execution</i> is defined as the execution of two or more executable entities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution. Note: This definition is derived from [ISO24765] .
critical section	In the EB tresos Safety OS <i>critical section</i> is defined as a section of an executable entity 's internal logic that is executed mutually exclusively with other executable entities . Note: This definition is derived from [ISO24765] .
deadlock	In the EB tresos Safety OS <i>deadlock</i> is defined as a situation in which two or more executable entities are suspended indefinitely because each executable entity is waiting for a resource acquired by another executable entity. Note: This definition is derived from [ISO24765] .
deadlock-free	In the EB tresos Safety OS <i>deadlock-free</i> is defined as a property of a mutual exclusion protocol that states that, by the use of primitives of that protocol, no deadlocks can occur.

degradation	In the EB tresos Safety OS <i>degradation</i> is defined as deactivating parts of the functionality while other parts are still active.
error handling	In the EB tresos Safety OS <i>error handling</i> is defined as the process of detecting and responding to errors of executable entities , the hardware or from the EB tresos Safety OS itself. Note: This definition is derived from <i>error processing</i> in [ISO24765] .
executable entity	In the EB tresos Safety OS <i>executable entity</i> is defined as software executed as an operating system application.
execution budget	In the EB tresos Safety OS <i>execution budget</i> is defined as the upper limit that is placed on the amount of processor time that can be used by an executable entity.
free from interference	In the EB tresos Safety OS <i>free from interference</i> is defined as <ul style="list-style-type: none"> ▶ having <i>freedom from interference</i> (when consulting [ISO26262_1ST]) and ▶ having <i>non-interference between software elements</i> (when consulting [ISO/IEC 61508:2010]). See also Section 5.2.2.1, “Term: freedom from interference” .
hardware	In the EB tresos Safety OS <i>hardware</i> is defined as the physical equipment used to process, store, or transmit computer programs or data. Note: This definition is copied from [ISO24765] .
inter-process notification	In the EB tresos Safety OS <i>inter-process notification</i> is defined as an atomic shared action resulting in one-way communication from an initiating executable entity to a responding executable entity . Note: This definition is derived from the <i>signal</i> term in [ISO24765] .
isolated subsystem	In the EB tresos Safety OS <i>isolated subsystem</i> is defined as a part of the operating system that is executed as a separate executable entity , thereby retaining spatial freedom from interference to the rest of the operating system.
logical sequence of instructions	In the EB tresos Safety OS <i>logical sequence of instructions</i> is defined as the sequence of instructions given by control flow of the program. Note: This means external interrupts or exceptions are invisible apart from temporal effects.
mechanism for scheduled events	In the EB tresos Safety OS <i>mechanism for scheduled events</i> is defined as a means to start or unblock executable entities based on an abstract notion of time, such as a regular interrupt.
memory region	In the EB tresos Safety OS <i>memory region</i> is defined as a contiguous part of the microcontroller's address space.
mutual exclusion	In the EB tresos Safety OS <i>mutual exclusion</i> is defined as the problem of ensuring that no competing executable entities can be in their critical section at the same time.

operating system	<p>In the EB tresos Safety OS <i>operating system</i> is defined as a collection of software, firmware, and hardware elements that controls the execution of executable entities and provides such services as computer resource allocation and job control in a computer system.</p> <p>Note: This definition is derived from [ISO24765].</p>
protection hardware	<p>In the EB tresos Safety OS <i>protection hardware</i> is defined as a hardware component that detects, prevents, and reports attempted unauthorized access.</p>
reliable execution environment	<p>In the EB tresos Safety OS <i>reliable execution environment</i> is defined as a microcontroller that provides a mechanism that prevents or detects non-systematic hardware faults (data corruption, incorrect execution of instructions).</p> <p>Note: Examples are lockstep mode and ECC memory.</p>
safety-related functionality	<p>In the EB tresos Safety OS <i>safety-related functionality</i> is defined as functionality to which a safety integrity level has been allocated.</p>
severe error	<p>In the EB tresos Safety OS <i>severe error</i> is defined as a non-recoverable error that makes it impossible to continue normal operation (including executing executable entities).</p>
simultaneous	<p>In the EB tresos Safety OS <i>simultaneous</i> is defined as pertaining to two events that occur at the same instant of time.</p> <p>Note: This definition is derived from [ISO24765].</p>
spatial freedom from interference	<p>In the EB tresos Safety OS <i>spatial freedom from interference</i> is defined as</p> <ul style="list-style-type: none"> ▶ freedom from interference for <i>memory</i> (when consulting [ISO26262_1ST]) and ▶ <i>spatial</i> non-interference between software elements (when consulting [ISO/IEC 61508:2010]). <p>See also Section 5.2.2.1, “Term: freedom from interference”.</p>
startup mechanism	<p>In the EB tresos Safety OS <i>startup mechanism</i> is defined as follows: The <i>startup mechanism</i> transfers the EB tresos Safety OS into a state in which it is fully operable and accepts service requests.</p>
time stamp	<p>In the EB tresos Safety OS <i>time stamp</i> is defined as an integer value that is directly proportional to the real time with respect to an arbitrary reference point.</p> <p>Comment: This reference point can be the time of the last reset.</p>
temporal freedom from interference	<p>In the EB tresos Safety OS <i>temporal freedom from interference</i> is defined as</p> <ul style="list-style-type: none"> ▶ freedom from interference for <i>timing and execution</i> (when consulting [ISO26262_1ST]) and ▶ <i>temporal</i> non-interference between software elements (when consulting [ISO/IEC 61508:2010]).

	See also Section 5.2.2.1, “Term: freedom from interference” .
unauthorized executable entity	In the EB tresos Safety OS <i>unauthorized executable entity</i> is defined as an executable entity that does not have the right to perform the given action.
unauthorized access	In the EB tresos Safety OS <i>unauthorized access</i> is defined as the attempt to access a memory location or hardware component without having the appropriate access rights.

Table 5.2. Definition of terms

5.2.2.1. Term: freedom from interference

For your convenience, the following two definitions have been copied verbatim into this document (see the following sections).

5.2.2.1.1. Freedom from interference in ISO 26262

Definition in [\[ISO26262-1_1ST\]](#):

absence of cascading failures between two or more elements that could lead to the violation of a safety requirement.

ISO 26262 states (see [\[ISO26262-6_1ST\]](#) Annex D) that freedom from interference has three aspects:

- ▶ Timing and execution
- ▶ Memory
- ▶ Exchange of information

5.2.2.1.2. Non-interference between software elements in IEC 61508

Definition in [\[ISO/IEC 61508-3:2010\]](#) Annex F.1:

The term "independence of execution" means that elements will not adversely interfere with each other's execution behavior such that a dangerous failure would occur. It is used to distinguish other aspects of independence which may be required between elements, in particular diversity, to meet other requirements of the standard.

Independence of execution should be achieved and demonstrated both in the spatial and temporal domains.

- ▶ *Spatial: the data used by one element shall not be changed by another element. In particular, it shall not be changed by a non-safety related element.*
- ▶ *Temporal: one element shall not cause another element to function incorrectly by taking too high a share of the available processor execution time, or by blocking execution of the other element by locking a shared resource of some kind.*

5.2.3. Requirements

This section defines the scheme for constructing *requirements*. In this document, the terms *should (not)* and *may* are not used in any requirement.

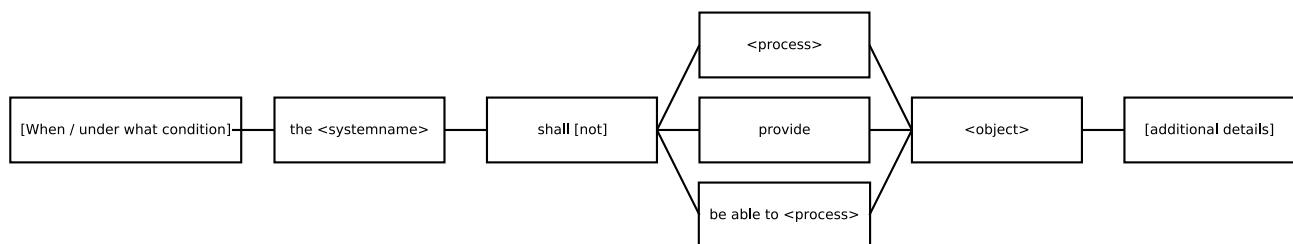


Figure 5.3. Scheme for defining a requirement

5.3. Level 1 requirements

Id:	OperatingSystem
Doctype:	reqspec1
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall provide an operating system .
Safety class:	ASIL-D
Safety rationale:	The operating system shall be developed to ASIL-D processes because the ECU software (including safety-related elements) is not free from interference from the operating system .

Id:	SpatialFreedomFromInterference
Doctype:	reqspec1
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall provide spatial freedom from interference between executable entities .
Safety class:	ASIL-D
Safety rationale:	[ISO26262-6_1ST] 7.4.11 requires a freedom from interference mechanism to be implemented according to the highest ASIL that is used in the element. In general, this is ASIL-D.

Id:	TemporalFreedomFromInterference
Doctype:	reqspec1
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall support temporal freedom from interference between executable entities .
Comment:	An operating system cannot fully provide temporal freedom from interference because some aspects (e.g. incorrect synchronization between software elements) can only be prevented on application level.
Safety class:	ASIL-D
Safety rationale:	[ISO26262-6_1ST] 7.4.11 requires a freedom from interference mechanism to be implemented according to the highest ASIL that is used in the element. In general, this is ASIL-D.

5.4. Level 2 requirements

5.4.1. Operating system requirements

Id:	OperatingSystem.ConcurrentExecution
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall provide the concurrent execution of executable entities .
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	OperatingSystem , Version: 1

Id: OperatingSystem.InterProcessNotification
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide](#) an [inter-process notification](#) mechanism.
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id: OperatingSystem.MutualExclusion
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide](#) a [mutual exclusion](#) mechanism.
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id: OperatingSystem.IsolatedSubsystem
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide](#) execution of [operating system](#) services in an [isolated subsystem](#).
Comment: This enables the provision of AUTOSAR-compliant alarm and schedule table handling as a separate subsystem that need not be developed according to ASIL.
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id: OperatingSystem.ErrorHandling
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide error handling](#).
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id: OperatingSystem.Startup
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide](#) a [startup mechanism](#).
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id: OperatingSystem.Degradation
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide](#) a [degradation](#) mechanism.
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id: OperatingSystem.ProvideCurrentTime
Doctype: reqspec2
Status: APPROVED
Version: 1
Description: The EB tresos Safety OS shall [provide](#) a service to get the current [timestamp](#).
Safety class: ASIL-D
Safety rationale: Inherited from parent requirement.
Provides coverage to: [OperatingSystem](#), Version: 1

Id:	OperatingSystem.ScheduledEvents
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall provide a mechanism for scheduled events
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	OperatingSystem , Version: 1

5.4.2. Freedom from interference requirements

Id:	SpatialFreedomFromInterference.ProtectMemorySections
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall program the protection hardware to protect memory regions and hardware components against unauthorized access .
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	SpatialFreedomFromInterference , Version: 1

Id:	SpatialFreedomFromInterference.SequenceOfInstructions
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall ensure that the logical sequence of instructions of executable entities is free from interference by unauthorized executable entities .
Comment:	Protecting only the data is not sufficient for having a safe system. It is also necessary to protect the logical sequence of instructions . This requirement means that executable entities can be interrupted, but need to be continued at exactly the place where the interruption occurred.
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	SpatialFreedomFromInterference , Version: 1

Id:	SpatialFreedomFromInterference.OsFromOther
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall ensure that its safety related functionality is free from interference by unauthorized executable entities .
Comment:	The non-safety-related part of the EB tresos Safety OS is also considered an unauthorized executable entity .
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	SpatialFreedomFromInterference , Version: 1

Id:	SpatialFreedomFromInterference.FromOtherCores
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	On hardware that is capable of executing two or more executable entities simultaneously, the EB tresos Safety OS shall ensure that its safety related functionality is free from interference by an executable entity that is executing simultaneously.
Comment:	The software executing on a multi-core processor may be under the control of a single multi-core instance of the EB tresos Safety OS or of another independent instance of the EB tresos Safety OS. The software (on another core) could equally be of unknown specification.
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	SpatialFreedomFromInterference , Version: 1

Id:	TemporalFreedomFromInterference.ExecutionBudgetMonitoring
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall provide execution budget monitoring.
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	TemporalFreedomFromInterference , Version: 1

Id:	TemporalFreedomFromInterference.DeadlockFreeMutualExclusion
Doctype:	reqspec2
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS shall provide a deadlock-free mutual exclusion mechanism.
Comment:	This requirement places emphasis on the <i>deadlock-freedom</i> of the mutual exclusion mechanism, whereas the requirement OperatingSystem.MutualExclusion places emphasis on the <i>presence</i> of it. Note that the EB tresos Safety OS may also provide mutual exclusion mechanisms that are <i>not</i> deadlock-free.
Safety class:	ASIL-D
Safety rationale:	Inherited from parent requirement.
Provides coverage to:	TemporalFreedomFromInterference , Version: 1

5.5. Assumptions

This section defines all assumptions on the hardware and surrounding system.

5.5.1. Defining assumptions

Assumptions must be defined using the following scheme:

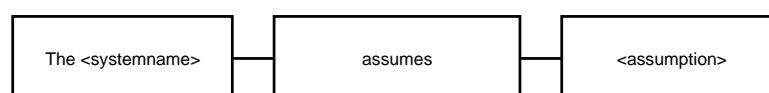


Figure 5.4. Scheme for defining assumptions

NOTE

Assumptions do not have safety integrity levels.



5.5.2. Assumptions on the hardware

To guarantee the fulfillment of the level 1 and 2 requirements, the EB tresos Safety OS assumes the [hardware](#) provides the following features:

Id:	Assumptions.ReliableExecutionEnvironment
Doctype:	assumptions
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS assumes that the hardware provides a reliable execution environment .

Id:	Assumptions.HardwareSupport
Doctype:	assumptions
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS assumes that the hardware provides protection hardware .
Comment:	Protection against interference by other bus masters , including other cores of a multi-core processor, is necessary, if the other bus masters are necessary for the functionality of the system.

Id:	Assumptions.HwSIL
Doctype:	assumptions
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS assumes that the hardware has been developed to the safety integrity level allocated to the operating system .

Id:	Assumptions.PrivilegeLevels
Doctype:	assumptions
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS assumes that the hardware provides at least two different privilege levels.
Comment:	Examples for such privilege levels are privileged mode and non-privileged mode .

5.5.3. Assumptions on the surrounding system

The design of EB tresos Safety OS makes the following assumptions on the surrounding system:

Id:	Assumptions.SevereErrorHandling
Doctype:	assumptions
Status:	APPROVED
Version:	1
Description:	The EB tresos Safety OS assumes that an endless loop with disabled interrupts is a safe way of handling severe errors .
Comment:	If this is not acceptable, the developer of the item must take measures to resolve this, e.g. by using a hardware watchdog.

5.6. Overview

This section provides an overview over the relationship of requirements on level 1 and 2.

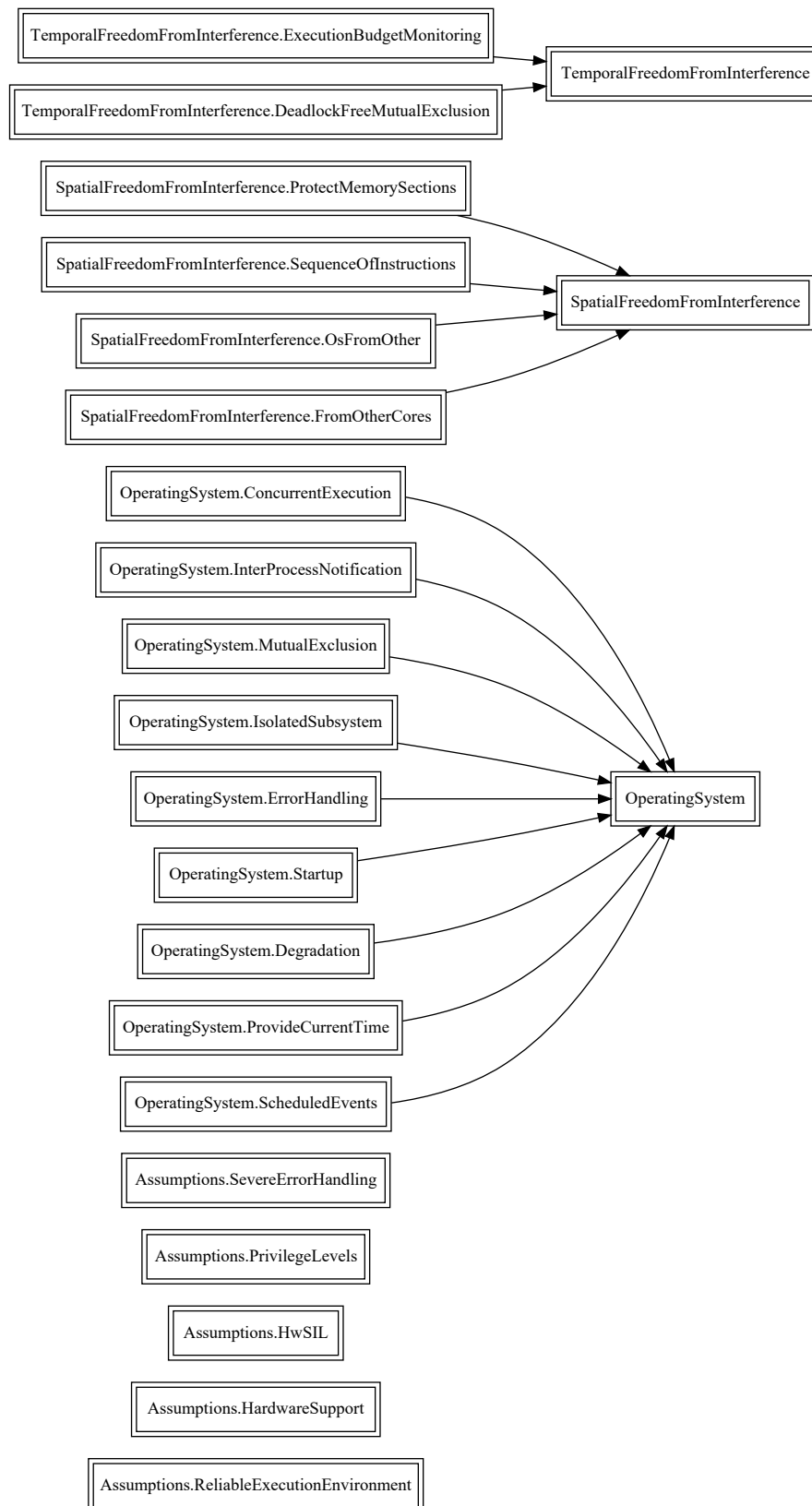


Figure 5.5. Overview of requirements on level 1 and 2

6. Configuration verification criteria

EB tresos Studio, and in particular the EB tresos AutoCore OS generator plugin, are not developed to the standards required for safety-relevant applications. For safe operation, ensure that the files created by EB tresos Studio and compiled into the microkernel specify the intended configuration of tasks, ISRs, resources, etc.

This chapter provides the verification criteria for the generated files `Mk_gen_user.h`, `Mk_gen_config.h`, `Mk_gen_global.c`, and `Mk_gen_addons.h`. You can use these criteria to verify that the generated files accurately represent the OS configuration provided to the OS generator. They do not verify that the OS configuration is correct for any particular application.

It is important to verify that the generated files fulfill your requirements. It is not sufficient to verify that the generated files accurately represent the specification provided as input to the OS generator unless you also verify that the input to the OS generator fulfills your requirements.

These configuration criteria are intended to be applied to the files generated by EB tresos Studio as used in your project. If you modify these files after generation, you must apply the criteria to the modified files. You can generate configurations by other possibilities and you must evaluate their safety separately with reference to the EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

NOTE



Logical and physical cores

The microkernel introduces several ways to identify a processor core.

- ▶ The logical core ID is typically used by microkernel APIs. It is defined by the configuration and can differ from the physical core ID.
- ▶ The physical core ID is defined by the hardware. Which IDs are used, as well as properties like if they are consecutive, is therefore defined by the hardware.
- ▶ The physical core index (or index of the physical core) is a microkernel internal mapping of the physical core IDs to a zero-based and consecutive index.

Whenever a VC does not specify if core *x* is a logical or physical core, *x* is the index of the physical core.

The microkernel is configured by macros that specify the initial values for constants. There are typically arrays or simple scalar values.

The criteria are listed by file. The statement in each criterion tells you what you must do in order to fulfill the criterion. You must not use the configuration in an application with safety-relevant components if any criterion is not fulfilled, unless you can verify the safety by a separate assessment.

You must ensure that the compiled version of the microkernel does not contain any development aids such as tracing hooks, patches etc.:

[VC.DEVAIDS.1]

Verify that the macro `MK_INCLUDE_CONFIG_PATCH` is not defined in any of your header files, and that it is not provided to the compiler via the command line.

[VC.DEVAIDS.2]

Verify that the macro `MK_USE_TRACE` is not defined in any of your header files, and that it is not provided to the compiler via the command line.

WARNING



Do not use `MK_USE_TRACE` in production software

The macro `MK_USE_TRACE` is only intended for non-production software. For production software the macro `MK_USE_TRACE` must not be used, i.e. it must be undefined.

[VC.DEVAIDS.3]

Verify that the macro `MK_USE_DBG` is not defined in any of your header files, and that it is not provided to the compiler via the command line.

[VC.LOCKINGCALLOUTS.1]

If you define the macro `MK_USE_INT_LOCKING_CALLOUTS` to a non-zero value, verify that the functions `MK_SuspendCallout()` and `MK_ResumeCallout()` provide adequate implementations for the APIs `SuspendAllInterrupts()/DisableAllInterrupts()/SuspendOSInterrupts()` and `ResumeAllInterrupts()/EnableAllInterrupts()/ResumeOSInterrupts()`, respectively.

You are entirely responsible for the functionality and safety of these callout functions. Your application could call these functions from any executable; task, ISR, trusted function, hook function, etc., of any safety integrity level. The locking could be nested, both from within a single executable and between executables. You must implement your functions so that they work correctly and safely in all possible situations.

6.1. Verification criteria for `Mk_gen_global.c`

The file `Mk_gen_global.c` contains the definitions of the stacks. This section lists the verification criteria for the file `Mk_gen_global.c`.

[VC.Mk_gen_global.1]

Verify that each stack declared in the file `Mk_gen_config.h` is defined in `Mk_gen_global.c` with the same size.

[VC.Mk_gen_global.2]

Verify that each stack defined in `Mk_gen_global.c` is marked using a toolchain-specific mechanism to enable the linker script to group the stacks as required. The mechanism is typically a linker section directive or linker section attribute.

6.2. Verification criteria for Mk_gen_config.h

This section lists the verification criteria for the generated file `Mk_gen_config.h`. This file contains the tables of objects managed by the microkernel as well as some global parameters.

This section mentions the *index of a memory partition* in several places. The index of a memory partition is the offset of that memory partition from the first memory partition of the associated core in the memory partition array `MK_CFG_MEMORYPARTITIONCONFIG`. You can calculate the index as the absolute index of the memory partition in that array minus `MK_CFG_Cx_FIRST_MEMORYPARTITION`, where `x` is the core on which the task runs.

This section also mentions values to set the processor status word (PSW) for the usage by an executable in several places. The PSW setting controls the processor mode, interrupt enable status and level, FPU use and other hardware-specific modes of an executable. The PSW value is set by means of the `processorMode`, `interruptLevel`, `interruptsEnabled`, `fpuEnabled` and `hwSpecificPs` parameters of the `MK_ISR_CFG`, `MK_TASKCFG` and `MK_ETASKCFG` macros.

The following criteria apply to the PSW settings for every executable:

[VC.PSW.1.1]

Verify that the `interruptsEnabled` parameter is configured to `MK_THRIRQ_ENABLE` for all executables that shall run with interrupts enabled. Most executables run with interrupts enabled, otherwise important microkernel functionality is not available.

[VC.PSW.1.2]

Verify that the `interruptsEnabled` parameter is configured to `MK_THRIRQ_DISABLE` for all executables that shall run with interrupts disabled.

[VC.PSW.2]

Verify that the `processorMode` parameter selects the least-privileged processor mode for the required functionality of the executable. In most cases the processor mode shall be a non-privileged mode. You must avoid running executables in privileged mode unless it is absolutely necessary. You must fully understand the implications of configuring an executable to run in privileged mode.

[VC.PSW.2.1]

Verify that the `processorMode` parameter is `MK_THRMODE_USER` or `MK_THRMODE_USER1` for all executables except those that need to run in privileged mode.

[VC.PSW.2.2]

Verify that the `processorMode` parameter is `MK_THRMODE_SUPER` for all executables that need to run in privileged mode.

[VC.PSW.3.1]

Verify that the `fpuEnabled` parameter is `MK_THRFPU_DISABLE` for all executables that do not use the floating point unit.

[VC.PSW.3.2]

Verify that the `fpuEnabled` parameter is `MK_THRFPU_ENABLE` for all executables that use the floating point unit.

[VC.PSW.4]

Verify that the `hwSpecificPs` parameter is `MK_THRHWS_DEFAULT` or another suitable value according to the hardware-specific part of this document.

6.2.1. Verification criteria for global settings

[VC.Mk_gen_config.global.1]

Verify that the value of the macro `MK_SCHEDULERPRIO` is equal to the queueing priority (`queuePrio` field of the `MK_TASKCFG()` or `MK_ETASKCFG()` macro of the highest priority task.

[VC.Mk_gen_config.global.3]

Verify that the value of the macro `MK_CAT2LOCKPRIO` is equal to the priority of the highest category 2 ISR. If there are no category 2 ISRs, verify that `MK_CAT2LOCKPRIO` is equal to `MK_SCHEDULERPRIO`.

[VC.Mk_gen_config.global.5]

Verify that the value of the macro `MK_CAT1LOCKPRIO` is the priority of the highest category 1 ISR. If there are no category 1 ISRs, verify that `MK_CAT1LOCKPRIO` is equal to `MK_CAT2LOCKPRIO`.

[VC.Mk_gen_config.global.6]

Verify that the value of the macro `MK_CAT2LOCKLEVEL` represents an interrupt level that blocks all category 2 ISRs, including the QM-OS's hardware timer ISRs.

On TriCore family, this means that there is no category 2 ISR with an interrupt level whose value is numerically higher than the value of the macro `MK_CAT2LOCKLEVEL`.

If there are no category 2 ISRs and no QM-OS hardware timer ISRs, verify that the value of `MK_CAT2LOCKLEVEL` does not cause any interrupts to be blocked.

[VC.Mk_gen_config.global.7]

Verify that the numerical value of the macro `MK_CAT2LOCKLEVEL` is not higher than 255 (the maximum value supported by the hardware).

[VC.Mk_gen_config.global.8]

Verify that the value of the macro `MK_CAT1LOCKLEVEL` represents an interrupt level that blocks all category 1 ISRs.

On TriCore family, this means that there is no category 1 ISR with an interrupt level whose value is numerically higher than the value of the macro `MK_CAT1LOCKLEVEL`.

If there are no category 1 ISRs, verify that the value of `MK_CAT1LOCKLEVEL` is equal to `MK_CAT2LOCKLEVEL`.

[VC.Mk_gen_config.global.9]

Verify that the numerical value of the macro `MK_CAT1LOCKLEVEL` is not higher than 255 (the maximum value supported by the hardware).

[VC.Mk_gen_config.global.10]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_INIT_MEMPART` is the index of an appropriate memory partition for the `main()` executable to run on core `x`.

Note: The function that is called by this executable is specified by `MK_CFG_Cx_INIT_FUNCTION`. The default is `main()`.

[VC.Mk_gen_config.global.11]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_IDLE_MEMPART` is the index of an appropriate memory partition for the idle thread to run on core `x`.

[VC.Mk_gen_config.global.12]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_QMOS_MEMPART` is the index of an appropriate memory partition for the QM-OS executables to run on core `x`.

[VC.Mk_gen_config.global.14]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_ERRORHOOK_MEMPART` is the index of an appropriate memory partition for the `ErrorHook()` executable to run on core `x`. `MK_CFG_Cx_ERRORHOOK_MEMPART` may be `-1` if the `ErrorHook()` is disabled or if the `ErrorHook()` does not need a dynamic partition.

[VC.Mk_gen_config.global.15]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_PROTECTIONHOOK_MEMPART` is the index of an appropriate memory partition for the `ProtectionHook()` executable to run on core `x`. `MK_CFG_Cx_PROTECTIONHOOK_MEMPART` may be `-1` if the `ProtectionHook()` is disabled or if the `ProtectionHook()` does not need a dynamic partition.

[VC.Mk_gen_config.global.16]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_SHUTDOWNHOOK_MEMPART` is the index of an appropriate memory partition for the `ShutdownHook()` executable to run on core `x`. `MK_CFG_Cx_SHUTDOWNHOOK_MEMPART` may be `-1` if the `ShutdownHook()` is disabled or if the `ShutdownHook()` does not need a dynamic partition.

[VC.Mk_gen_config.global.17]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_SHUTDOWN_MEMPART` is the index of an appropriate memory partition for the shutdown-idle executable to run on core `x`. `MK_CFG_Cx_SHUTDOWN_MEMPART` may be `-1` if the shutdown-idle executable does not need a dynamic partition.

[VC.Mk_gen_config.global.20]

[removed]

[VC.Mk_gen_config.global.22]

If the value of the macro `MK_HAS_USERPANICSTOP` is not equal to zero, verify that the value of the macro `MK_USERPANICSTOP(reason)` is the name of the function that you provided for this purpose. This func-

tion is called if a fatal error occurs during start-up when it is not possible to shut down the kernel properly. You can use the parameter `reason` of type `mk_panic_t` to determine the nature of the fatal error.

WARNING **Panic during start-up**



If the user start-up panic stop function is called, the system has not started properly and might be in a fatal dysfunctional state. You *must not rely* upon being able to do *anything useful* in the user start-up panic stop function, like resetting or shutting down the device.

[VC.Mk_gen_config.global.24]

Verify that the value of the macro `MK_CFG_HWMASTERCOREINDEX` is the index of a suitable core on which the low-level initialization of the shared processor hardware is performed. The selected core must be among the cores on which the EB tresos Safety OS is configured to run. It must be a core whose hardware has the highest integrity level of the microcontroller (for example, a lockstep core).

[VC.Mk_gen_config.global.25]

Verify that the value of the macro `MK_CFG_NADDONS` is 0.

6.2.2. Verification criteria for microkernel executables

[VC.Mk_gen_config.MK.1]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_KERNEL_STACK_NELEMENTS` is large enough to allow all microkernel system calls (including those provided by the configured addons) to run on core x without causing a stack overflow.

[VC.Mk_gen_config.MK.2]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_KERNEL_STACK_NELEMENTS` is at least as large as the value `MkKernStack` configured in EB tresos Studio divided by `sizeof(mk_stackelement_t)`.

[VC.Mk_gen_config.MK.3]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_IDLE_FUNCTION` is the address of `MK_Idle`.

[VC.Mk_gen_config.MK.4]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_IDLE_MODE` is `MK_THRMODE_USER`.

[VC.Mk_gen_config.MK.5]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_SHUTDOWN_FUNCTION` is the address of `MK_Idle`.

[VC.Mk_gen_config.MK.6]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_SHUTDOWN_MODE` is `MK_THRMODE_USER`.

[VC.Mk_gen_config.MK.7]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_IDLE_STACK_NELEMENTS` is large enough that the functions specified by `MK_CFG_Cx_IDLE_FUNCTION` and `MK_CFG_Cx_SHUT-DOWN_FUNCTION` can run without causing a stack overflow.

[VC.Mk_gen_config.MK.8]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_IDLE_STACK_NELEMENTS` is at least as large as the value `MkIdleStack` configured in EB tresos Studio divided by `sizeof(mk_stackelement_t)`.

[VC.Mk_gen_config.MK.9]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_INIT_FUNCTION` is the address of the configured init function (typically: `main`) and that its prototype is provided via the `Mk_board.h` file.

[VC.Mk_gen_config.MK.10]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_INIT_MODE` specifies the lowest privilege level that is necessary to permit the configured `MK_CFG_Cx_INIT_FUNCTION` to execute correctly.

[VC.Mk_gen_config.MK.11]

For each configured core x , verify that the value of the macros `MK_CFG_Cx_INIT_QPRIO` and `MK_CFG_Cx_INIT_RPRIO` are equal to the highest priority of all task objects.

6.2.3. Verification criteria for the QM-OS

The QM-OS itself is not covered by this document, but the freedom from interference between the QM-OS and your tasks and ISRs depends on a correct configuration for the QM-OS environment that is managed by the microkernel.

This section specifies the verification criteria for the QM-OS environment.

[VC.Mk_gen_config.QM.1]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_QMOS_MODE` is `MK_THRMODE_USER1`. QM-OS threads must run in a non-privileged mode.

[VC.Mk_gen_config.QM.3]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_QMOS_STACK_NELEMENTS` is large enough for all configured QM-OS services, including the counter update service, to run without causing a stack overflow.

[VC.Mk_gen_config.QM.4]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_QMOS_STACK_NELEMENTS` is at least as large as the value `MkOsStack` configured in EB tresos Studio divided by `sizeof(mk_stackelement_t)`.

[VC.Mk_gen_config.QM.10]

Verify that the value of the macro `MK_CFG_NTIMERS` is 0.

[VC.Mk_gen_config.QM.11]

Verify that the value of the macro `MK_CFG_NCOUNTERS` is equal to the total number of counters that you configured.

[VC.Mk_gen_config.QM.12]

If `MK_CFG_NCOUNTERS` is greater than 0, verify that the value of the macro `MK_CFG_COUNTERPROPERTIES` expands to a comma-separated list of `MK_CFG_NCOUNTERS` invocations of the macro `MK_QMPROP(coreIndex, appId)`.

[VC.Mk_gen_config.QM.13]

If `MK_CFG_NCOUNTERS` is greater than 0, verify that for each invocation of `MK_QMPROP(coreIndex, appId)` in `MK_CFG_COUNTERPROPERTIES` the value of `coreIndex` is greater than or equal to 0 and less than `MK_MAXCORES`.

Note: The correct value for `coreIndex` is the index of the physical processor core to which the corresponding counter is mapped. The corresponding counter for an entry in `MK_CFG_COUNTERPROPERTIES` is the counter with the ID equal to the 0-based index in that list. The correct processor core is the one to which the counter's `OsApplication` is mapped in the configuration.

[VC.Mk_gen_config.QM.14]

Verify that the value of the macro `MK_CFG_NALARMS` is equal to the total number of alarms that you configured.

[VC.Mk_gen_config.QM.15]

If `MK_CFG_NALARMS` is greater than 0, verify that the value of the macro `MK_CFG_ALARMPROPERTIES` expands to a comma-separated list of `MK_CFG_NALARMS` invocations of the macro `MK_QMPROP(coreIndex, appId)`.

[VC.Mk_gen_config.QM.16]

If `MK_CFG_NALARMS` is greater than 0, verify that for each invocation of `MK_QMPROP(coreIndex, appId)` in `MK_CFG_ALARMPROPERTIES` the value of `coreIndex` is greater than or equal to 0 and less than `MK_MAXCORES`.

Note: The correct value for `coreIndex` is the index of the physical processor core to which the corresponding alarm is mapped. The corresponding alarm for an entry in `MK_CFG_ALARMPROPERTIES` is the alarm with the ID equal to the 0-based index in that list. The alarm is mapped to the same processor core as its counter. For more information on the processor core of a counter, see [\[VC.Mk_gen_config.QM.13\]](#).

[VC.Mk_gen_config.QM.17]

Verify that the value of the macro `MK_CFG_NSCHEDULETABLES` is equal to the total number of schedule tables that you configured.

[VC.Mk_gen_config.QM.18]

If `MK_CFG_NSCHEDULETABLES` is greater than 0, verify that the value of the macro `MK_CFG_SCHEDULETABLEPROPERTIES` is a comma-separated list of `MK_CFG_NSCHEDULETABLES` invocations of the macro `MK_QMPROP(coreIndex, appId)`.

[VC.Mk_gen_config.QM.19]

If `MK_CFG_NSCHEDULETABLES` is greater than 0, verify that for each invocation of `MK_QMPROP(coreIndex, appId)` in `MK_CFG_SCHEDULETABLEPROPERTIES` the value of `coreIndex` is greater than or equal to 0 and less than `MK_MAXCORES`.

Note: The correct value for `coreIndex` is the index of the physical processor core to which the corresponding schedule table is mapped. The corresponding schedule table for an entry in `MK_CFG_SCHEDULETABLEPROPERTIES` is the schedule table with the ID equal to the 0-based index in that list. The correct processor core is the one to which the schedule table's `OsApplication` is mapped in the configuration.

6.2.4. Verification criteria for trusted function threads

[VC.Mk_gen_config.TF.3]

For each core `x` on which trusted functions are configured, verify that the value of the macro `MK_CFG_Cx_TF_STACK_NELEMENTS` is large enough to permit any trusted function that is configured to run on core `x` to run without causing a stack overflow.

[VC.Mk_gen_config.TF.4]

For each core `x` on which trusted functions are configured, verify that the value of the macro `MK_CFG_Cx_TF_STACK_NELEMENTS` is at least as large as the largest value of `OsTrustedFunctionStacksize` that you configured in EB tresos Studio, divided by `sizeof(mk_stackelement_t)`.

[VC.Mk_gen_config.TF.10]

Verify that the value of the macro `MK_CFG_NTRUSTEDFUNCTIONS` is equal to the total number of trusted functions that you configured.

[VC.Mk_gen_config.TF.11]

Verify that the value of the macro `MK_CFG_TRUSTEDFUNCTIONLIST` expands to a comma-separated list of `MK_CFG_NTRUSTEDFUNCTIONS` trusted function configurations.

[VC.Mk_gen_config.TF.12]

For each trusted function in the list `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that it expands to an invocation of the macro `MK_TRUSTEDFUNCTIONCFG(name, core, function, mode, fpu, hws, mem-part)`.

[VC.Mk_gen_config.TF.13]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `name` parameter is the name of a trusted function that you configured.

[VC.Mk_gen_config.TF.14]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `core` parameter is the index of the core to which you assigned the trusted function.

[VC.Mk_gen_config.TF.15]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `function` parameter is the C identifier of the function that shall be called. Usually this is `TRUSTED_<name>`.

[VC.Mk_gen_config.TF.16]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `mode` parameter is the least-privileged processor mode that permits the function to execute correctly.

[VC.Mk_gen_config.TF.17]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `fpv` parameter is either `MK_THRFPU_ENABLE` if the trusted function uses floating-point calculations, or `MK_THRFPU_DISABLE` if the trusted function does not use floating-point calculations.

[VC.Mk_gen_config.TF.18]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `hws` parameter is `MK_THRHWS_DEFAULT` or another suitable value according to the hardware-specific part of this document.

[VC.Mk_gen_config.TF.19]

For each invocation of `MK_TRUSTEDFUNCTIONCFG()` in `MK_CFG_TRUSTEDFUNCTIONLIST`, verify that the `mempart` parameter specifies a memory partition that permits the trusted function to access exactly the memory regions (including the trusted function stack) that it is configured to access.

6.2.5. Verification criteria for hook functions

[VC.Mk_gen_config.Hook.ErrorHook.1]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_HAS_ERRORHOOK` is either 1 if the `ErrorHook()` feature is enabled for the core or 0 if the `ErrorHook()` feature is disabled for the core.

[VC.Mk_gen_config.Hook.ErrorHook.1.1]

For each configured core `x` on which the `ErrorHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_ERRORHOOK_MODE` is the least privileged mode that permits your `ErrorHook()` to execute correctly.

[VC.Mk_gen_config.Hook.ErrorHook.1.2]

For each configured core `x` on which the `ErrorHook()` is enabled, verify that the value of the macros `MK_CFG_Cx_ERRORHOOK_QPRIO` and `MK_CFG_Cx_ERRORHOOK_RPRIO` is 2200.

[VC.Mk_gen_config.Hook.ErrorHook.1.3]

For each configured core `x` on which the `ErrorHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_ERRORHOOK_LEVEL` is equal to `MK_CAT1LOCKLEVEL`.

[VC.Mk_gen_config.Hook.ErrorHook.1.4]

For each configured core `x` on which the `ErrorHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_ERRORHOOK_STACK_NELEMENTS` is large enough for your `ErrorHook()` function to run without causing a stack overflow.

[VC.Mk_gen_config.Hook.ErrorHook.1.5]

For each configured core *x* on which the `ErrorHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_ERRORHOOK_STACK_NELEMENTS` is at least as large as the value `MkErrorHookStack` configured in EB tresos Studio divided by `sizeof(mk_stackelement_t)`.

[VC.Mk_gen_config.Hook.ProtectionHook.1]

For each configured core *x*, verify that the value of the macro `MK_CFG_Cx_HAS_PROTECTIONHOOK` is either 1 if the `ProtectionHook()` feature is enabled for the core or 0 if the `ProtectionHook()` feature is disabled for the core.

[VC.Mk_gen_config.Hook.ProtectionHook.1.1]

For each configured core *x* on which the `ProtectionHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_PROTECTIONHOOK_MODE` is the lowest privilege mode that permits your `ProtectionHook()` to execute.

[VC.Mk_gen_config.Hook.ProtectionHook.1.2]

For each configured core *x* on which the `ProtectionHook()` is enabled, verify that the value of the macros `MK_CFG_Cx_PROTECTIONHOOK_QPRIO` and `MK_CFG_Cx_PROTECTIONHOOK_RPRIO` is 2300.

[VC.Mk_gen_config.Hook.ProtectionHook.1.3]

For each configured core *x* on which the `ProtectionHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_PROTECTIONHOOK_LEVEL` is equal to `MK_CAT1LOCKLEVEL`.

[VC.Mk_gen_config.Hook.ProtectionHook.1.4]

For each configured core *x* on which the `ProtectionHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_PROTECTIONHOOK_STACK_NELEMENTS` is large enough for your `ProtectionHook()` function to run without causing a stack overflow.

[VC.Mk_gen_config.Hook.ProtectionHook.1.5]

For each configured core *x* on which the `ProtectionHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_PROTECTIONHOOK_STACK_NELEMENTS` is at least as large as the value `MkProtectionHookStack` configured in EB tresos Studio divided by `sizeof(mk_stackelement_t)`.

[VC.Mk_gen_config.Hook.ShutdownHook.1]

For each configured core *x*, verify that the value of the macro `MK_CFG_Cx_HAS_SHUTDOWNHOOK` is either 1 if the `ShutdownHook()` feature is enabled for the core or 0 if the `ShutdownHook()` feature is disabled for the core.

[VC.Mk_gen_config.Hook.ShutdownHook.1.1]

For each configured core *x* on which the `ShutdownHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_SHUTDOWNHOOK_MODE` is a suitable value for the `ShutdownHook()` thread.

[VC.Mk_gen_config.Hook.ShutdownHook.1.2]

For each configured core *x* on which the `ShutdownHook()` is enabled, verify that the values of the macros `MK_CFG_Cx_SHUTDOWNHOOK_QPRIO` and `MK_CFG_Cx_SHUTDOWNHOOK_RPRIO` are both 2100.

[VC.Mk_gen_config.Hook.ShutdownHook.1.3]

For each configured core x on which the `ShutdownHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_SHUTDOWNHOOK_LEVEL` is equal to `MK_CATALOCKLEVEL`.

[VC.Mk_gen_config.Hook.ShutdownHook.1.4]

For each configured core x on which the `ShutdownHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_SHUTDOWNHOOK_STACK_NELEMENTS` is large enough for your `ShutdownHook()` function to run without causing a stack overflow.

[VC.Mk_gen_config.Hook.ShutdownHook.1.5]

For each configured core x on which the `ShutdownHook()` is enabled, verify that the value of the macro `MK_CFG_Cx_SHUTDOWNHOOK_STACK_NELEMENTS` is at least as large as the value `MkShutdownHookStack` configured in EB tresos Studio divided by `sizeof(mk_stackelement_t)`.

6.2.6. Verification criteria for interrupts and ISRs

[VC.Mk_gen_config.IRQ.1]

Verify that the value of the macro `MK_CFG_NIRQS` is the total number of interrupt sources that are in use. The number includes the [internal interrupts](#) used by the microkernel, as well as your ISRs and the QM-OS interrupts. Internal interrupts are described in [Section 6.2.7, “Verification criteria for internal interrupts”](#).

[VC.Mk_gen_config.IRQ.2]

Verify that the value of the macro `MK_CFG_IRQLIST` expands to a comma-separated list of `MK_CFG_NIRQS` invocations of the macro `MK_IRQCFG(ctrlreg, level, coreIndex, flags)`.

[VC.Mk_gen_config.IRQ.2.1]

For each invocation of `MK_IRQCFG()` in `MK_CFG_IRQLIST`, verify that the value of the `ctrlreg` parameter is the address of the control register in the interrupt controller that contains the enable and level settings for the interrupt source.

[VC.Mk_gen_config.IRQ.2.2]

For each invocation of `MK_IRQCFG()` in `MK_CFG_IRQLIST`, verify that the value of the `level` parameter in each invocation of `MK_IRQCFG()` is the hardware-dependent interrupt level at which this request operates.

[VC.Mk_gen_config.IRQ.2.3.1]

For each invocation of `MK_IRQCFG()` in `MK_CFG_IRQLIST`, verify that the value of the `coreIndex` parameter contains the index of the physical core on which the interrupt shall be serviced.

[VC.Mk_gen_config.IRQ.2.3]

For each invocation of `MK_IRQCFG()` in `MK_CFG_IRQLIST`, verify that the value of the `flags` parameter contains `MK_IRQ_ENABLE` if the interrupt source is configured to be enabled at start-up. If the interrupt source is configured to remain disabled at start-up, `MK_IRQ_ENABLE` shall not be present.

Note: All internal interrupts and QM-OS interrupts shall be enabled at start-up.

[VC.Mk_gen_config.IRQ.4]

For each invocation of `MK_IRQCFG()` in `MK_CFG_IRQLIST`, verify that there is a `MK_SOFTVECTOR_nnnn` macro with the correct vector number `nnnn`. For TriCore processors, the vector number is 1 less than the `level` parameter in the invocation of `MK_IRQCFG()`.

[VC.Mk_gen_config.ISR.1]

For each ISR object in your configuration, verify that the corresponding macro `MK_SOFTVECTOR_nnnn` is defined with the value `MK_VECTOR_ISR(i)`, where `i` is the index of the ISR in the ISR array defined by `MK_CFG_ISRLIST`.

[VC.Mk_gen_config.ISR.4]

Verify that the value of the macro `MK_CFG_ISRLIST` expands to a comma-separated list of `MK_CFG_NISRS` invocations of either the macro

`MK_ISRCFG(irqref, threadref, regref, dynamic, name, coreIndex, stacklimit, initialSP, ISRFunction, processorMode, interruptLevel, interruptsEnabled, fpuEnabled, hwSpecificPs, queuePrio, runPrio, partIdx, isrId, execBudget, lockBudgets, aId)`

or the macro

`MK_QMOSISRCFG(irqref, threadref, regref, dynamic, name, coreIndex, stacklimit, initialSP, ISRFunction, processorMode, interruptLevel, interruptsEnabled, fpuEnabled, hwSpecificPs, queuePrio, runPrio, partIdx, isrId, execBudget, lockBudgets, aId, counter).`

[VC.Mk_gen_config.ISR.4.1]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `irqref` parameter is a reference (pointer) to the corresponding IRQ configuration. The value is normally `&MK_irqCfgTable[j]`, where `j` is the index of the corresponding IRQ in `MK_CFG_IRQLIST`.

[VC.Mk_gen_config.ISR.4.2]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `threadref` parameter is a reference (pointer) to the thread structure that is used to control the ISR thread. The value shall be `MK_cx_isrThreads[n]`, where `x` is the physical core index of the ISR and `n` is an integer in the range 0 to `MK_CFG_Cx_NISRTHREADS-1`.

[VC.Mk_gen_config.ISR.4.2.1]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `threadref` parameter is not shared by any task or by any other ISR that has a different `runPrio`.

[VC.Mk_gen_config.ISR.4.3]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `regref` parameter is a reference (pointer) to a register store for the ISR thread. The value shall be `MK_cx_isrRegisters[n]`, where `x` is the physical core index of the ISR and `n` is an integer in the range 0 to `MK_CFG_Cx_NISRREGISTERS-1`.

[VC.Mk_gen_config.ISR.4.4]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `regref` parameter is not shared by any other ISR that has a different `queuePrio`.

[VC.Mk_gen_config.ISR.4.4.1]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `dynamic` parameter is a reference to `MK_cx_isrDynamic[n]`, where `x` is the core on which the ISR shall execute and `n` is an integer in the range 0 to `MK_CFG_Cx_NISRS`. See also the `coreIndex` parameter.

[VC.Mk_gen_config.ISR.4.4.2]

Verify that no two ISRs share the same `dynamic` object.

[VC.Mk_gen_config.ISR.4.5]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `name` parameter is the name of the ISR in the form of a literal C character string.

[VC.Mk_gen_config.ISR.4.5.1]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `coreIndex` parameter is the index of the physical processor core on which the ISR shall execute.

[VC.Mk_gen_config.ISR.4.6]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `stacklimit` parameter is the lowest value that the stack pointer is permitted to take during the life-time of the ISR.

[VC.Mk_gen_config.ISR.4.7]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `initialSP` parameter is the correct initial value of the stack pointer for the thread. On some hardware this could be outside the stack. On others there must be a defined amount of accessible memory above the `initialSP`. The value must be correctly aligned according to the requirements of the hardware.

For hardware-specific verification criteria, see [Section 6.2.15.1, “Verification criteria for stacks on TriCore family processors”](#).

[VC.Mk_gen_config.ISR.4.8]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `ISRFunction` parameter is the address of the interrupt service routine's function. For generated ISRs the function name is `OS_ISR_<name>`. For ISRs that handle the hardware timers in the QM-OS, the function name is the name of the generated OS function in `Os_gen.c`.

[VC.Mk_gen_config.ISR.4.11]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `processorMode` parameter is the least privileged processor mode needed by the ISR.

[VC.Mk_gen_config.ISR.4.12]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `interruptLevel` parameter is not lower than the interrupt level defined for the corresponding interrupt request referenced by `irqref`.

[VC.Mk_gen_config.ISR.4.32]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, if the ISR is non-preemptive, verify that the value of the `interruptLevel` parameter is equal to the highest `interruptLevel` among ISRs of the same category.

[VC.Mk_gen_config.ISR.4.25]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `interruptsEnabled` parameter is `MK_THRIRQ_ENABLE`, so that the ISR threads run with interrupts enabled.

[VC.Mk_gen_config.ISR.4.26]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `fpuEnabled` parameter is either `MK_THRFPU_ENABLE` or `MK_THRFPU_DISABLE`.

[VC.Mk_gen_config.ISR.4.27]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `hwSpecificPs` parameter is either `MK_THRHS_DEFAULT` or another suitable value according to the hardware-specific part of this document.

[VC.Mk_gen_config.ISR.4.13]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the `queuePrio` parameter is higher than the `runPrio` parameter of any other ISR whose `interruptLevel` is lower than this ISR.

[VC.Mk_gen_config.ISR.4.14]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `queuePrio` parameter is higher than `MK_SCHEDULERPRIO`.

[VC.Mk_gen_config.ISR.4.16]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `runPrio` parameter is not lower than the `queuePrio` parameter.

[VC.Mk_gen_config.ISR.4.33]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, if the ISR is non-preemptive, verify that the value of the `runPrio` parameter is equal to the highest `queuePrio` among the ISRs of the same category.

[VC.Mk_gen_config.ISR.4.17]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `partIdx` parameter is either `-1` or the index of an appropriate memory partition for this ISR in the `MK_CFG_MEMORYPARTITIONCONFIG` array.

[VC.Mk_gen_config.ISR.4.20]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `isrId` parameter is the index of this ISR. It should be defined by means of the ISR's identifier macro from `Mk_gen_user.h`.

[VC.Mk_gen_config.ISR.4.21]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `execBudget` parameter is:

- ▶ `MK_EXECBUDGET_INFINITE`, if the ISR is configured to run with no execution time limit.

- ▶ `MK_ExecutionNsToTicks(N)`, if the ISR is configured to run with an execution time limit of `N` nanoseconds.

[VC.Mk_gen_config.ISR.4.23]

In each invocation of `MK_ISRCFG` and `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `lockBudgets` parameter is `MK_NULL`. This parameter is not used.

[VC.Mk_gen_config.ISR.4.24]

In each invocation of `MK_ISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `aId` parameter is the index of the OS-Application to which the ISR belongs. `aId` should be defined by means of the OS-Application's identifier macro from `Mk_gen_user.h`.

[VC.Mk_gen_config.ISR.4.34]

In each invocation of `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `aId` parameter is `MK_APPL_NONE`.

[VC.Mk_gen_config.ISR.4.35]

In each invocation of `MK_QMOSISRCFG` in `MK_CFG_ISRLIST`, verify that the value of the `counter` is the index of the respective QM-OS counter.

[VC.Mk_gen_config.ISR.5]

Verify, that each occurrence `MK_SOFTVECTOR_nnnn` in your configuration, where `nnnn` is the vector number, is defined to one of the following values: `MK_VECTOR_ISR()`, `MK_VECTOR_DEMUX()`, `MK_VECTOR_XCORE()`, `MK_VECTOR_QMOSISR()` or `MK_VECTOR_INTERNAL()`.

6.2.7. Verification criteria for internal interrupts

[VC.Mk_gen_config.InternalISR.2]

For each [inter-core](#) interrupt, verify that the macro `MK_SOFTVECTOR_nnn` is defined as `MK_VECTOR_XCORE(0)`, where `nnn` is the vector number of the interrupt. The mapping of inter-core interrupts is described in [Section 6.2.15.4, “Verification criteria for TriCore family processors with multiple cores”](#).

6.2.8. Verification criteria for tasks

[VC.Mk_gen_config.Task.1]

Verify that the value of the macro `MK_CFG_NTASKS` is the total number of tasks, both BASIC and EXTENDED, that you configured.

[VC.Mk_gen_config.Task.1.1]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_NTASKS` is the total number of tasks, both EXTENDED and BASIC, that you configured to execute on core `x`.

[VC.Mk_gen_config.Task.2]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_NETASKS` is the number of EXTENDED tasks that you configured to execute on core `x`.

[VC.Mk_gen_config.Task.3]

Verify that the value of the macro `MK_CFG_TASKLIST` expands to a comma-separated list of invocations of one of the two macros:

- ▶ `MK_TASKCFG(dynamic, threadref, regref, name, coreIndex, stacklimit, initialSP, Taskfunction, processorMode, interruptLevel, interruptsEnabled, fpuEnabled, hwSpecificPs, queuePrio, runPrio, maxActivations, partIdx, taskId, execBudget, lockBudgets, appId)`
- ▶ `MK_ETASKCFG(dynamic, threadref, regref, name, coreIndex, stacklimit, initialSP, Taskfunction, processorMode, interruptLevel, interruptsEnabled, fpuEnabled, hwSpecificPs, queuePrio, runPrio, eventRef, partIdx, taskId, execBudget, lockBudgets, appId)`

[VC.Mk_gen_config.Task.4.1]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `dynamic` parameter is a reference to `MK_cx_taskDynamic[n]`, where `x` is the core on which the task shall execute. See also the `coreIndex` parameter.

[VC.Mk_gen_config.Task.4.1.1]

Verify that no two tasks share the same `dynamic` object.

[VC.Mk_gen_config.Task.4.2]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `threadref` parameter is a reference (pointer) to the thread that is used to control the execution of the task. The value shall be `MK_cx_taskThreads[i]`, where `x` is the physical core index of the task and `i` is an integer in the range 0 to `MK_CFG_Cx_NTASKTHREADS-1`.

[VC.Mk_gen_config.Task.4.3]

Verify that all tasks that share a common thread instance (`threadref`) have the same `queuePrio` value.

[VC.Mk_gen_config.Task.4.3.1]

Verify that all tasks that share a common thread instance (`threadref`) have the same `coreIndex` value.

[VC.Mk_gen_config.Task.4.4]

Verify that all tasks with the same `coreIndex` and `queuePrio` share a common thread instance (`threadref`).

[VC.Mk_gen_config.Task.4.6]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `regref` parameter is a reference (pointer) to a register store for the task. The value shall be `MK_cx_taskRegisters[n]`, where `x` is the physical core index of the task and `n` is an integer in the range 0 to `MK_CFG_Cx_NTASKREGISTERS-1`.

[VC.Mk_gen_config.Task.4.7]

In each invocation of `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `regref` parameter does not appear in any other invocation of either `MK_TASKCFG()` or `MK_ETASKCFG()`.

[VC.Mk_gen_config.Task.4.8]

Verify that all tasks that share a common register store (`regref`) have the same `queuePrio`.

[VC.Mk_gen_config.Task.4.8.1]

[removed]

[VC.Mk_gen_config.Task.4.9]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the `name` parameter is the name of the task in the form of a literal C character string.

[VC.Mk_gen_config.Task.4.9.1]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `coreIndex` parameter is the index of the physical processor core on which you configured the task to execute.

[VC.Mk_gen_config.Task.4.10]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `stacklimit` parameter is the lowest value that the stack pointer can take during the life-time of the task.

[VC.Mk_gen_config.Task.4.11]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `initialSP` parameter is the correct initial value of the stack pointer for the task and is correctly aligned according to the requirements specified by the hardware supplier.

The exact value in relation to the stack array is hardware-dependent. Some processors need some accessible memory above the stack pointer, whereas others can use an initial value that is outside the stack array. For hardware-specific verification criteria, refer to [Section 6.2.15.1, "Verification criteria for stacks on Tri-Core family processors"](#).

[VC.Mk_gen_config.Task.4.12]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `Taskfunction` parameter is the name of the task's function. The function name is `OS_TASK_XXX` where `XXX` is the configured name of the task.

[VC.Mk_gen_config.Task.4.13]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `processorMode` parameter is the least privileged processor mode that the task requires to perform its functionality.

[VC.Mk_gen_config.Task.4.29]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `interruptLevel` parameter is the correct interrupt level that corresponds with the `runPrio` value. The correct value is `MK_HWENABLEALLLEVEL` unless the task uses an internal resource that is also used by an ISR.

[VC.Mk_gen_config.Task.4.30]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `interruptsEnabled` parameter is `MK_THIRQ_ENABLE`.

[VC.Mk_gen_config.Task.4.31]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `fpuEnabled` parameter is:

- ▶ `MK_THRFPU_ENABLE` if the task uses the hardware floating-point unit.
- ▶ `MK_THRFPU_DISABLE` if the task does not use the hardware floating-point unit.

[VC.Mk_gen_config.Task.4.32]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `hwSpecificPs` parameter is `MK_THRHWS_DEFAULT` or another suitable value according to the hardware-specific part of this document.

[VC.Mk_gen_config.Task.4.14]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `queuePrio` parameter is strictly greater than zero.

[VC.Mk_gen_config.Task.4.15]

For each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `queuePrio` parameter is

- ▶ the same as the `queuePrio` value of all tasks with the same `coreIndex` and with the same configured priority.
- ▶ higher than the `queuePrio` value of all tasks with the same `coreIndex` and with a lower configured priority.
- ▶ lower than the `queuePrio` value of all tasks with the same `coreIndex` and with a higher configured priority.

[VC.Mk_gen_config.Task.4.16]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `queuePrio` parameter is lower than or equal to `MK_SCHEDULERPRIO`.

[VC.Mk_gen_config.Task.4.17]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `runPrio` parameter is equal to the value of the `queuePrio` parameter unless the task is non-preemptable or uses an internal resource.

[VC.Mk_gen_config.Task.4.18]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST` that configures a non-preemptable task, verify that the value of the `runPrio` parameter is at least the `queuePrio` of the highest-priority task with the same `coreIndex`.

[VC.Mk_gen_config.Task.4.19]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST` that configures a task that uses an internal resource, verify that the value of the `runPrio` parameter is at least as high as the largest `queuePrio` value from the set of tasks and ISRs that share an internal resource with this task.

OSEK/VDX and AUTOSAR limit the number of internal resources that a task can use to one, and do not permit ISRs to use internal resources. The microkernel does not have these limitations.

[VC.Mk_gen_config.Task.4.20]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `partIdx` parameter is either `-1` or the index of an appropriate memory partition for this task in the `MK_CFG_MEMORYPARTITIONCONFIG` array.

[VC.Mk_gen_config.Task.4.23]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `taskId` parameter is the index of this task. It should be defined using the task's identifier macro from `Mk_gen_user.h`.

[VC.Mk_gen_config.Task.4.24]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `execBudget` parameter is:

- ▶ `MK_EXECBUDGET_INFINITE`, if the task is configured to run with no execution time limit.
- ▶ `MK_ExecutionNsToTicks(N)`, if the task is configured to run with an execution time limit of `N` nanoseconds.

[VC.Mk_gen_config.Task.4.26]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `lockBudgets` parameter is `MK_NULL`. This parameter is not used.

[VC.Mk_gen_config.Task.4.27]

In each invocation of `MK_TASKCFG` and `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `appId` parameter is the index of the OS-Application to which the task belongs. It should be defined using the OS-Application's identifier macro from `Mk_gen_user.h`.

[VC.Mk_gen_config.Task.4a.28]

In each invocation of `MK_TASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `maxActivations` parameter is at least 1.

[VC.Mk_gen_config.Task.4a.29]

In each invocation of `MK_TASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `maxActivations` parameter is equal to the `OsTaskMaxActivations` parameter that you configured for the task in EB tresos Studio.

[VC.Mk_gen_config.Task.4b.30]

In each invocation of `MK_ETASKCFG` in `MK_CFG_TASKLIST`, verify that the value of the `eventRef` parameter is the address of the event status structure for the task. The value shall be `&MK_cx_eventStatus[n]` where `x` is the physical core index of the task and `n` is in the range 0 to `MK_CFG_Cx_NETASKS`.

[VC.Mk_gen_config.Task.4b.32]

Verify that no two invocations of the `MK_ETASKCFG()` in `MK_CFG_TASKLIST` have the same value for the `eventRef` parameter.

6.2.9. Verification criteria for threads and job queues

[VC.Mk_gen_config.JobQueue.1]

For each configured core, verify that the value of the macro `MK_CFG_Cx_NTASKTHREADS` is the total number of threads that you need in order to run the tasks that are configured to run on core `x`. Its value shall be 1 greater than the largest `n` among the tasks whose `threadref` configuration is `&MK_cx_taskThreads[n]`.

[VC.Mk_gen_config.JobQueue.2]

For each configured core, verify that the value of the macro `MK_CFG_Cx_NISRTHREADS` is the total number of threads that you need in order to run the ISRs that are configured to run on core `x`. Its value shall be 1 greater than the largest `n` among the ISRs whose `threadref` configuration is `&MK_cx_isrThreads[n]`.

[VC.Mk_gen_config.JobQueue.3]

For each configured core, verify that the value of the macro `MK_CFG_Cx_NTASKREGISTERS` is the total number of register stores that you need to run the tasks that are configured to run on core `x`. Its value shall be 1 greater than the largest `n` among the tasks whose `regref` configuration is `&MK_cx_taskRegisters[n]`.

[VC.Mk_gen_config.JobQueue.4]

For each configured core, verify that the value of the macro `MK_CFG_Cx_NISRREGISTERS` is the total number of register stores that you need to run the ISRs that are configured to run on core `x`. Its value shall be 1 greater than the largest `n` among the ISRs whose `regref` configuration is `&MK_cx_isrRegisters[n]`.

[VC.Mk_gen_config.JobQueue.5]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_NJOBQUEUES` is 2 greater than the number of task threads on core `x` that are associated with one or both of the following:

- ▶ Two or more tasks.
- ▶ A task whose `maxActivations` is greater than 1.

[VC.Mk_gen_config.JobQueue.6]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_JOBQUEUECONFIG` expands to a comma-separated list of `MK_CFG_Cx_NJOBQUEUES` invocations of the macro `MK_JOBQUEUECFG(threadref, len, size)`.

[VC.Mk_gen_config.JobQueue.6.1]

For each invocation of `MK_JOBQUEUECFG()` in each `MK_CFG_Cx_JOBQUEUECONFIG`, verify that the value of the `threadref` parameter in each invocation of `MK_JOBQUEUECFG()` is a reference (pointer) to one of the following:

- ▶ `MK_cx_aux1Thread`.
- ▶ `MK_cx_aux2Thread`.
- ▶ an element of the array `MK_cx_taskThreads[]` that requires a job queue.

For the condition for requiring a job queue, see `VC.Mk_gen_config.JobQueue.5`.

[VC.Mk_gen_config.JobQueue.6.2]

In each `MK_CFG_Cx_JOBQUEUECONFIG`, verify that no two invocations of the `MK_JOBQUEUECFG()` macro have the same value for the `threadref` parameter.

[VC.Mk_gen_config.JobQueue.6.4]

For each invocation of `MK_JOBQUEUECFG()` in each `MK_CFG_Cx_JOBQUEUECONFIG` whose `threadref` parameter refers to a task thread, verify that the `len` parameter in the invocation is 1 fewer than the aggregate number of task activations in the tasks that refer to the same `threadref`. To calculate the aggregate, add together the `maxActivations` of all tasks (`MK_TASKCFG()` and `MK_ETASKCFG()`) that refer to the thread. In the case of `MK_ETASKCFG()`, take `maxActivations` to be 1.

[VC.Mk_gen_config.JobQueue.6.4.1]

For each invocation of `MK_JOBQUEUECFG()` in each `MK_CFG_Cx_JOBQUEUECONFIG` whose `threadref` parameter refers to either `MK_cx_aux1Thread` or `MK_cx_aux1Thread`, verify that the `len` parameter in the invocation is a strictly positive number.

[VC.Mk_gen_config.JobQueue.6.7]

For each invocation of `MK_JOBQUEUECFG()` in each `MK_CFG_Cx_JOBQUEUECONFIG` whose `threadref` parameter refers to a task thread, verify that the `size` parameter has a numerical value of 1.

[VC.Mk_gen_config.JobQueue.6.7.1]

For each invocation of `MK_JOBQUEUECFG()` in each `MK_CFG_Cx_JOBQUEUECONFIG` whose `threadref` parameter refers to a `MK_cx_aux1Thread`, verify that the `size` parameter is `MK_QMJ_SIZE`.

[VC.Mk_gen_config.JobQueue.6.7.2]

For each invocation of `MK_JOBQUEUECFG()` in each `MK_CFG_Cx_JOBQUEUECONFIG` whose `threadref` parameter refers to a `MK_cx_aux2Thread`, verify that the `size` parameter is `MK_TFJ_SIZE`.

6.2.10. Verification criteria for OS-Applications

[VC.Mk_gen_config.OSApplications.1]

Verify that the macro `MK_CFG_NAPPLICATIONS` is the total number of OS-Applications in the system.

Note: the value of `MK_CFG_NAPPLICATIONS` might include one or more OS-Applications with name of the form `OS_SYSTEM_<n>`. For the purposes of the remaining verification criteria relating to OS-Applications you can ignore these system applications.

[VC.Mk_gen_config.OSApplications.1.2]

Verify that the macro `MK_CFG_APPLIST` expands to a comma-separated list of `MK_CFG_NAPPLICATIONS` invocations of the macro `MK_APPCONFIG(appId, appDynamic, coreIndex, restartTask, globalPerm, accessingApps)`.

[VC.Mk_gen_config.OSApplications.1.2.1]

For each invocation of `MK_APPCONFIG()` in `MK_CFG_APPLIST`, verify that the value of the `appId` parameter is the index of this invocation of `MK_APPCONFIG()` in the list.

[VC.Mk_gen_config.OSApplications.1.3]

For each invocation of `MK_APPCONFIG()` in `MK_CFG_APPLIST`, verify that the value of the `appDynamic` parameter is the address of a unique element in the `MK_cx_appDynamic[]` array where `x` is the core on which the OS-Application is configured to run. The index of the element shall be in the range 0 to `MK_CFG_Cx_NAPPLICATIONS-1`.

[VC.Mk_gen_config.OSApplications.1.4]

For each invocation of `MK_APPCONFIG()` in `MK_CFG_APPLIST`, verify that the value of the `coreIndex` parameter is the index of the physical core on which the OS-Application is configured to run.

[VC.Mk_gen_config.OSApplications.2]

For each invocation of `MK_APPCONFIG()` in `MK_CFG_APPLIST`, verify that the value of the `restartTask` parameter is the task ID of the configured restart task for the OS-Application. If the OS-Application has no restart task, the value of the `restartTask` shall be `MK_INVALID_TASK`.

[VC.Mk_gen_config.OSApplications.6]

For each invocation of `MK_APPCONFIG()` in `MK_CFG_APPLIST`, verify that the value of the `globalPerm` parameter contains zero or more of the following macros:

- ▶ `MK_PERMIT_SHUTDOWNOS` if your configuration permits this application to call `ShutdownOS()`
- ▶ `MK_PERMIT_SHUTDOWNALLCORES` if your configuration permits this application to call `ShutdowncWAllCoresOS()`

The permissions shall be combined by means of a bitwise-OR operator. If you granted no permissions, the value shall be zero.

[VC.Mk_gen_config.OSApplications.7]

For each invocation of `MK_APPCONFIG()` in `MK_CFG_APPLIST`, verify that the value of the `accessingApps` parameter evaluates to a bitwise-OR of the access bits of all OS-Applications that are permitted to terminate this application. The access bit of an application whose `appId` is `AA` is calculated as $(1 \ll AA)$. Note: an OS-Application should always have permission to terminate itself, so the `accessingApps` parameter should never be zero.

[VC.Mk_gen_config.OSApplications.5]

For each configured physical core `x`, verify that the macro `MK_CFG_Cx_NAPPLICATIONS` is the number of OS-Applications in `MK_CFG_APPLIST` whose `coreIndex` is equal to `x`.

6.2.11. Verification criteria for locks

[VC.Mk_gen_config.Locks.1]

Verify that the macro `MK_CFG_LOCKLIST` is a comma-separated list of `MK_CFG_NLOCKS` invocations of the macro `MK_LOCKCFG(prio, level, nesting, spinlock, coreIndex, lockIndex)`.

[VC.Mk_gen_config.Locks.2]

Verify that the macro `MK_CFG_NGLOBALLOCKS` is `lockIndex + 1` of the lock with the highest `lockIndex` that is present on all cores. Locks that are present on all cores are the following:

- ▶ RESCAT1.
- ▶ RESCAT2.
- ▶ RES_SCHEDULER.
- ▶ All locks whose `spinlock` parameter is not `MK_NULL`.

[VC.Mk_gen_config.Locks.4]

If you configured `RES_SCHEDULER`, verify that `MK_CFG_LOCKLIST` contains an entry for `RES_SCHEDULER` for each configured physical core `x` with the following values:

- ▶ `prio = MK_SCHEDULERPRIO`
- ▶ `level = MK_HWENABLEALLLEVEL`
- ▶ `nesting = 1`
- ▶ `spinlock = MK_NULL`
- ▶ `coreIndex = x`
- ▶ `lockIndex = RES_SCHEDULER` (as defined in `Mk_gen_user.h`).

[VC.Mk_gen_config.Locks.5]

Verify that `MK_CFG_LOCKLIST` contains an entry for `MK_RESCAT2` for each configured physical core `x` with the following values:

- ▶ `prio = MK_CAT2LOCKPRIO`
- ▶ `level = MK_CAT2LOCKLEVEL`
- ▶ `nesting = 0xffff`
- ▶ `spinlock = MK_NULL`
- ▶ `coreIndex = x`
- ▶ `lockIndex = MK_RESCAT2` (as defined in `Mk_gen_user.h`).

[VC.Mk_gen_config.Locks.6]

Verify that `MK_CFG_LOCKLIST` contains an entry for `MK_RESCAT1` for each configured physical core `x` with the following values:

- ▶ `prio = MK_CAT1LOCKPRIO`
- ▶ `level = MK_CAT1LOCKLEVEL`
- ▶ `nesting = 0xffff`
- ▶ `spinlock = MK_NULL`
- ▶ `coreIndex = x`
- ▶ `lockIndex = MK_RESCAT1` (as defined in `Mk_gen_user.h`).

[VC.Mk_gen_config.Locks.7]

For each `OsSpinlock` that you configured, verify that `MK_CFG_LOCKLIST` contains an entry for it.

[VC.Mk_gen_config.Locks.7.1]

For each `OsSpinlock` that you configured, verify that the value of its `prio` is one of the following, depending on the locking method you chose for the spinlock:

- ▶ `MK_CAT1LOCKPRIO.`
- ▶ `MK_CAT2LOCKPRIO.`
- ▶ `MK_SCHEDULERPRIO.`
- ▶ `-1`

[VC.Mk_gen_config.Locks.7.2]

For each `OsSpinlock` that you configured, verify that the value of its `level` is one of the following, depending on the locking method you chose for the spinlock:

- ▶ `MK_CAT1LOCKLEVEL.`
- ▶ `MK_CAT2LOCKLEVEL.`
- ▶ `MK_HWENABLEALLLEVEL.`

[VC.Mk_gen_config.Locks.7.3]

For each `OsSpinlock` that you configured, verify that the value of its `nesting` is 1.

[VC.Mk_gen_config.Locks.7.4]

For each `OsSpinlock` that you configured, verify that the value of the `spinlock` parameter is defined as `MK_HwGetSpinlockCfg(i)` for some `i` strictly in the range `[0..MK_CFG_NSPINLOCKS-1]`.

[VC.Mk_gen_config.Locks.7.6]

For each `OsSpinlock` that you configured, verify that the value of its `lockIndex` parameter is strictly less than `MK_CFG_NGLOBALLOCKS`.

[VC.Mk_gen_config.Locks.8]

For each `OsSpinlock` that you configured, verify that all of its representations in `MK_CFG_LOCKLIST` have the same value for their `lockIndex` parameters.

Verify that there is a representation for each physical core that you have configured with `coreIndex` set to the corresponding physical core index.

[VC.Mk_gen_config.Locks.9]

For each STANDARD or LINKED `OsResource` that you configured, verify that `MK_CFG_LOCKLIST` contains an entry for it.

Internal resources do not appear in the lock list.

[VC.Mk_gen_config.Locks.9.1]

For each STANDARD or LINKED `OsResource` that you configured, verify that the value of its `prio` is the `queuePrio` of the highest-priority task or ISR that uses the resource or any other resource to which it is linked.

[VC.Mk_gen_config.Locks.9.2]

For each STANDARD or LINKED `OsResource` that you configured that is not used by an ISR or linked to another resource that is used by an ISR, verify that its `level` parameter is `MK_HWENABLEALLLEVEL`.

[VC.Mk_gen_config.Locks.9.3]

For each STANDARD or LINKED `OsResource` that you configured that is used by one or more ISRs or linked to another resource that is used by ISRs, verify that the numerical value of its `level` is sufficiently high to block any ISR that uses the resource or another resource to which it is linked.

[VC.Mk_gen_config.Locks.9.4]

For each STANDARD or LINKED `OsResource` that you configured, verify that the value of its `nesting` is 1.

[VC.Mk_gen_config.Locks.9.5]

For each STANDARD or LINKED `OsResource` that you configured, verify that the value of its `spinlock` is `MK_NULL`.

[VC.Mk_gen_config.Locks.9.6]

For each STANDARD or LINKED `OsResource` that you configured, verify that the value of its `coreIndex` is the index of the physical core to which it is bound.

[VC.Mk_gen_config.Locks.9.7]

For each STANDARD or LINKED `OsResource` that you configured, verify that the value of its `lockIndex` parameter is not less than `MK_CFG_NGLOBALLOCKS` and is strictly less than `MK_CFG_Cx_NLOCKS`, where `x` is the lock's `coreIndex`.

[VC.Mk_gen_config.Locks.9.8]

For each STANDARD or LINKED `OsResource` that you configured, verify that it does not have the same `lockIndex` as any other entry in `MK_CFG_LOCKLIST` with the same value for `coreIndex`.

6.2.12. Verification criteria for memory partitions

[VC.Mk_gen_config.MemoryPartition.1]

Verify that the value of the macro `MK_CFG_NMEMORYPARTITIONS` is equal to the sum of the `MK_CFG_Cx_NMEMORYPARTITIONS` over all configured cores.

If `MK_CFG_NMEMORYPARTITIONS` is less than zero, the memory protection unit remains disabled. While this behavior can be useful for debugging, the resulting system does not provide spatial freedom from interference and is therefore unlikely to fulfill any safety requirements, especially when services with integrity category 2 or 3 (QM-OS services) are used.

[VC.Mk_gen_config.MemoryPartition.1.1]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_NMEMORYPARTITIONS` not less than `MK_CFG_Cx_NSTATICPARTITIONS`.

In most configurations it is significantly greater.

[VC.Mk_gen_config.MemoryPartition.1.2]

For each configured core `x`, verify that the value of the macro `MK_CFG_Cx_NSTATICPARTITIONS` is strictly greater than zero.

[VC.Mk_gen_config.MemoryPartition.1.3]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_FIRST_MEMORYPARTITION` is the index of the first memory partition of core x in the `MK_CFG_MEMORYPARTITIONCONFIG` list.

[VC.Mk_gen_config.MemoryPartition.1.4]

For each configured core x , verify that the value of the sum `MK_CFG_Cx_FIRST_MEMORYPARTITION + MK_CFG_Cx_NMEMORYPARTITIONS` is less than or equal to `MK_CFG_NMEMORYPARTITIONS`.

[VC.Mk_gen_config.MemoryPartition.1.5]

For each configured core x , verify that the value of the sum `MK_CFG_Cx_FIRST_MEMORYPARTITION + MK_CFG_Cx_NMEMORYPARTITIONS` is less than or equal to the `MK_CFG_Cy_FIRST_MEMORYPARTITION` of any other core y whose `MK_CFG_Cy_FIRST_MEMORYPARTITION` is greater than or equal to `MK_CFG_Cx_FIRST_MEMORYPARTITION`.

[VC.Mk_gen_config.MemoryPartition.2]

Verify that the value of the macro `MK_CFG_NMEMORYPARTITIONS` is the number of entries in the `MK_CFG_MEMORYPARTITIONCONFIG` list.

[VC.Mk_gen_config.MemoryPartition.3]

Verify that the value of the macro `MK_CFG_MEMORYPARTITIONCONFIG` is a comma-separated list of invocations of the macro `MK_MEMORYPARTITION(start, num)`.

[VC.Mk_gen_config.MemoryPartition.3.1]

For each invocation of `MK_MEMORYPARTITION()` in `MK_CFG_MEMORYPARTITIONCONFIG`, verify that the value of the `start` parameter is the index of the first region map in the `MK_CFG_MEMORYREGIONMAPCONFIG` list that is part of the partition.

[VC.Mk_gen_config.MemoryPartition.3.2]

For each invocation of `MK_MEMORYPARTITION()` in `MK_CFG_MEMORYPARTITIONCONFIG`, verify that the value of the `num` parameter is the number of memory regions in the partition.

[VC.Mk_gen_config.MemoryPartition.3.3]

For each invocation of `MK_MEMORYPARTITION()` in `MK_CFG_MEMORYPARTITIONCONFIG`, verify that the sum of the values of the `start` and `num` parameters is less than or equal to the value of the `MK_CFG_NMEMORYREGIONMAPS` macro.

[VC.Mk_gen_config.MemoryPartition.4]

Verify that the value of the macro `MK_CFG_NMEMORYREGIONMAPS` is the number of entries in the `MK_CFG_MEMORYREGIONMAPCONFIG` list.

[VC.Mk_gen_config.MemoryPartition.5]

Verify that the value of the macro `MK_CFG_MEMORYREGIONMAPCONFIG` is a comma-separated list of invocations of the macro `MK_MEMORYREGIONMAP(partition, region, permissions, hwselection)`.

[VC.Mk_gen_config.MemoryPartition.5.0]

In each invocation of `MK_MEMORYREGIONMAP()` in `MK_CFG_MEMORYREGIONMAPCONFIG`, verify that the value of the `partition` parameter is the index of the memory partition to which this region mapping belongs (as defined in the partition's entry in `MK_CFG_MEMORYPARTITIONCONFIG`).

The parameter is not used by the microkernel, but is emitted by the OS generator for you to use as a verification aid.

[VC.Mk_gen_config.MemoryPartition.5.1]

In each invocation of `MK_MEMORYREGIONMAP()` in `MK_CFG_MEMORYREGIONMAPCONFIG`, verify that the value of the `region` parameter is the index of a memory region in the `MK_CFG_MEMORYREGIONCONFIG()` list that belongs to the partition given in the `partition` parameter in the invocation. `region` shall therefore be in the range `0..MK_CFG_NMEMORYREGIONS-1`.

[VC.Mk_gen_config.MemoryPartition.5.2]

In each invocation of `MK_MEMORYREGIONMAP()` in `MK_CFG_MEMORYREGIONMAPCONFIG`, verify that the value of the `permissions` parameter correctly represents the access rights that you granted over the `region` to any executable object that uses this `partition`.

[VC.Mk_gen_config.MemoryPartition.5.3]

In each invocation of `MK_MEMORYREGIONMAP()` in `MK_CFG_MEMORYREGIONMAPCONFIG`, verify that the value of the `hwselection` parameter is `MK_UNUSED` unless the region is selected using a hardware-specific selection criterion (such as might be used for fast partitions).

[VC.Mk_gen_config.MemoryPartition.6]

Verify that the value of the macro `MK_CFG_NMEMORYREGIONS` is the number of entries in the `MK_CFG_MEMORYREGIONCONFIG` list.

[VC.Mk_gen_config.MemoryPartition.7]

Verify that the value of the macro `MK_CFG_MEMORYREGIONCONFIG` is a comma-separated list of invocations of one of the three macros `macros MK_MR_INIT(name, icore, hwextra), MK_MR_NOINIT(name, hwextra) or MK_MR_STACK(name, icore, hwextra)`.

[VC.Mk_gen_config.MemoryPartition.7.A]

In each invocation of `MK_MR_INIT(name, icore, hwextra), MK_MR_NOINIT(name, hwextra) or MK_MR_STACK(name, icore, hwextra)` in `MK_CFG_MEMORYREGIONCONFIG`, verify that the value of the `name` parameter is the name of either a memory region that you configured or a memory region that EB tresos Safety OS needs for its correct operation.

The memory regions needed by EB tresos Safety OS are described in the User's Guide in Table 7.2, and in chapter 9 in the architecture specific memory region table.

[VC.Mk_gen_config.MemoryPartition.7.B]

In each invocation of `MK_MR_INIT(name, icore, hwextra), MK_MR_NOINIT(name, hwextra) or MK_MR_STACK(name, icore, hwextra)` in `MK_CFG_MEMORYREGIONCONFIG`, verify that the value of the `hwextra` parameter is `MK_UNUSED` or correctly represents the hardware-specific configuration for this memory region.

[VC.Mk_gen_config.MemoryPartition.7.C]

In each invocation of `MK_MR_STACK(name, icore, hwextra)` in `MK_CFG_MEMORYREGIONCONFIG`, verify that the value of the `icore` parameter is the index of the core on which this stack region is used.

[VC.Mk_gen_config.MemoryPartition.7.D]

In each invocation of `MK_MR_INIT(name, icore, hwextra)` in `MK_CFG_MEMORYREGIONCONFIG`, verify that the value of the `icore` parameter is either `-1` or the index of the core on which this memory region shall be initialized.

[VC.Mk_gen_config.MemoryPartition.7.E]

In each invocation of `MK_MR_INIT(name, icore, hwextra)` in `MK_CFG_MEMORYREGIONCONFIG` whose value for the `icore` is `-1`, verify that the memory region represents memory that is strictly local to each core. “Strictly local” means that each core has private physically independent memory at the same address that needs to be initialized separately by the core.

[VC.Mk_gen_config.MemoryPartition.7.F]

In each invocation of `MK_MR_NOINIT(name, hwextra)` in `MK_CFG_MEMORYREGIONCONFIG` verify that the memory region represents memory that is one of the following:

- ▶ A stack.
- ▶ A memory-mapped peripheral region.
- ▶ A global region whose purpose is to provide read-only access to other configured regions.
- ▶ A memory region that is initialized by other overlapping regions.
- ▶ A memory region that is initialized by some other means.

In the latter two cases, it is your responsibility to ensure that the region is initialized to satisfy your requirements before using it.

[VC.Mk_gen_config.MemoryPartition.9]

Verify that the configuration of the static partitions does not permit executables to have unrestricted access to regions that contain safety-relevant data.

The regions mapped into the static memory partitions are permanently present in the memory protection hardware. You must configure the access rights and selection criteria for these regions to ensure that the hardware only grants access according to your requirements. In particular, write access to the microkernel's data regions shall be restricted to the microkernel.

See also [Section 6.2.15.3, “Verification criteria for memory regions on TriCore family processors”](#).

[VC.Mk_gen_config.MemoryPartition.11]

Verify that no memory region that is writable by a thread overlaps a region that contains any of the following:

- ▶ Code belonging to the microkernel.
- ▶ Data belonging to the microkernel.
- ▶ Peripherals reserved for exclusive use by the microkernel.

[VC.Mk_gen_config.MemoryPartition.13]

For each configured core *x*, verify that the value of the macro `MK_CFG_Cx_MPUREGIONCACHESIZE` is equal to the sum of the values *num* of all invocations of `MK_MEMORYPARTITION(start, num)` for memory partitions which are assigned to core *x*.

6.2.13. Verification criteria for simple schedule tables

The configuration of the SST must satisfy the verification criteria listed in this section. This applies whether you configured the SST manually or by using EB tresos Studio.

[VC.Mk_gen_config.SST.1]

Verify that the value of the macro `MK_CFG_NSSTS` is the number of configured counters.

[VC.Mk_gen_config.SST.2]

Verify that the macro `MK_CFG_SSTCOUNTERTABLE` is defined as a comma-separated list of invocations of `MK_SSTCOUNTERCONFIG(ctr, mod, act, nact, tkr, rel, core)`. The number of invocations shall be equal to the value of `MK_CFG_NSSTS`.

[VC.Mk_gen_config.SST.3]

Verify that in each invocation of `MK_SSTCOUNTERCONFIG`, the first parameter *ctr* is the address of an element in the `MK_cx_sstCounters[]` array and that every such element is referenced exactly once by all `MK_SSTCOUNTERCONFIG` invocations. The *x* in the array name must be equal to the *core* argument.

[VC.Mk_gen_config.SST.4]

Verify that in each invocation of `MK_SSTCOUNTERCONFIG`, the second parameter *mod* is an integer between 2 and $1073741824 = 2^{30}$.

[VC.Mk_gen_config.SST.5]

Verify that in each invocation of `MK_SSTCOUNTERCONFIG`, the third parameter *act* is the address of an array or part of an array of expiry actions constructed using the macro `MK_SSTACTIONCONFIG(ct, task, evt)`. The array or partial array shall contain *nact* elements (see VC.Mk_gen_config.SST.6).

[VC.Mk_gen_config.SST.6]

Verify that in each invocation of `MK_SSTCOUNTERCONFIG`, the fourth parameter *nact* specifies the number of elements in the array or partial array based at *act*.

[VC.Mk_gen_config.SST.7]

Verify that in the first invocation of the macro `MK_SSTACTIONCONFIG` in the array or partial array referenced by each invocation of `MK_SSTCOUNTERCONFIG`, the value of the first parameter *ct* is an integer between 0 and *mod* inclusive.

[VC.Mk_gen_config.SST.8]

Verify that in each subsequent invocation of the macro `MK_SSTACTIONCONFIG` in the array or partial array referenced by each invocation of `MK_SSTCOUNTERCONFIG`, the value of the first parameter *ct* is an integer between the *ct* of the prior invocation and *mod* inclusive.

[VC.Mk_gen_config.SST.9]

Verify that in each invocation of the macro `MK_SSTACTIONCONFIG` the second parameter `task` specifies the id of the task that you configured to be activated or receive events at the time of the expiry point.

[VC.Mk_gen_config.SST.10]

Verify that in each invocation of the macro `MK_SSTACTIONCONFIG` the third parameter `evt` specifies a bitwise-OR combination of the events that you configured to be sent to the task (see VC.Mk_gen_config.SST.9), or 0 if the task shall be activated.

[VC.Mk_gen_config.SST.11]

Verify that in each invocation of `MK_SSTCOUNTERCONFIG`, the fifth parameter `tkr` is either -1 or the id of an available hardware timer which is not used for any other purpose.

[VC.Mk_gen_config.SST.12]

REMOVED

[VC.Mk_gen_config.SST.13]

Verify that in each invocation of `MK_SSTCOUNTERCONFIG` for which `tkr` is not -1, the sixth parameter `rel` specifies (after macro expansion) a number of cycles of the hardware timer's input clock that results in the interrupt frequency that your application requires. If `tkr` is -1, the parameter `rel` is not used at all.

[VC.Mk_gen_config.SST.14]

Verify that the value of `rel` in VC.Mk_gen_config.SST.13 causes the hardware timer to generate an interrupt frequency that the processor can handle. For this take into account the execution time of your tasks that are awakened by the SST and the duration of the critical sections in your application that result in a delay in servicing the interrupt. Note that if you violate this, interrupts or task activations may get lost, or the `ProtectionHook()` may be called on some hardware.

[VC.Mk_gen_config.SST.15]

Verify that the value of `rel` in VC.Mk_gen_config.SST.13 results in an interval that can be measured by the hardware timer. If the value is too large, it is truncated, and thus the final value might be smaller than you expect.

[VC.Mk_gen_config.SST.16]

Verify that the value of the seventh parameter `core` is the id of the core to which the SST counter is assigned to. See VC.Mk_gen_config.SST.3.

6.2.14. Verification criteria for advanced logical core identifiers

[VC.Mk_gen_config.ALCI.1]

Verify that for each logical core `x` from 0 to `MK_MAXCORES-1`, the macro `MK_CFG_Cx_ALCI_LOG_TO_PHY` is defined with a valid number greater than or equal to 0 and less than `MK_MAXCORES`.

[VC.Mk_gen_config.ALCI.2]

Verify that for each physical core x from 0 to $MK_MAXCORES-1$, the macro $MK_CFG_Cx_ALCI_PHY_TO_LOG$ is defined with a valid number greater than or equal to 0 and less than $MK_MAXCORES$.

[VC.Mk_gen_config.ALCI.3]

Verify that for each logical core x from 0 to $MK_MAXCORES-1$, the macros $MK_CFG_Cx_ALCI_LOG_TO_PHY$ are unique.

[VC.Mk_gen_config.ALCI.4]

Verify that for each physical core x from 0 to $MK_MAXCORES-1$, the macros $MK_CFG_Cx_ALCI_PHY_TO_LOG$ are unique.

[VC.Mk_gen_config.ALCI.5]

Verify that for each logical core x from 0 to $MK_MAXCORES-1$ and the macro $MK_CFG_Cx_ALCI_LOG_TO_PHY$ defined as y , the macro $MK_CFG_Cy_ALCI_PHY_TO_LOG$ is defined as x .

[VC.Mk_gen_config.ALCI.6]

Verify that for each physical core x from 0 to $MK_MAXCORES-1$ and the macro $MK_CFG_Cx_ALCI_PHY_TO_LOG$ defined as y , the macro $MK_CFG_Cy_ALCI_LOG_TO_PHY$ is defined as x .

[VC.Mk_gen_config.ALCI.7]

Verify that for each logical core x from 0 to $MK_MAXCORES-1$, the value of the macro $MK_CFG_Cx_ALCI_LOG_TO_PHY$ is the index of the intended physical core.

[VC.Mk_gen_config.ALCI.8]

Verify that for each physical core x from 0 to $MK_MAXCORES-1$, the value of the macro $MK_CFG_Cx_ALCI_PHY_TO_LOG$ is the index of the intended logical core.

6.2.15. Additional configuration criteria for TriCore family processors

6.2.15.1. Verification criteria for stacks on TriCore family processors

[VC.TRICORE.Stack.1]

Verify that the initial value of the stack pointer for every executable object (the value of `initialSP` in the invocation of `MK_TASKCFG()` and `MK_ISRCFG()`) is aligned on an 8-byte boundary.

[VC.TRICORE.Stack.2]

Verify that the initial value of the stack pointer for every executable is equal to or lower than the address of the first stack element above the end of the stack. (The size of a stack element is 4 bytes.)

6.2.15.2. Verification criteria for interrupts on TriCore family processors

[VC.TRICORE.Interrupts.1]

For each invocation of `MK_ISRCFG()`, `MK_QMOSISRCFG()` and `MK_TASKCFG()`, verify that the `interruptLevel` parameter does not exceed

- ▶ 255 (on a single-core configuration)
- ▶ 254 (if you configured the microkernel to run on two or more cores).

[VC.TRICORE.Interrupts.1a]

For each invocation of `MK_LOCKCFG()`, verify that the `level` parameter does not exceed

- ▶ 255 (on a single-core configuration)
- ▶ 254 (if you configured the microkernel to run on two or more cores).

[VC.TRICORE.Interrupts.2]

With the exception of the cross-core interrupts on a multi-core system, verify that there are no two invocations of `MK_IRQCFG()` in `MK_CFG_IRQLIST` with the same value for their `level` parameter.

[VC.TRICORE.Interrupts.3]

Verify that the value of the `level` parameter in each invocation of `MK_IRQCFG()` in `MK_CFG_IRQLIST` is in the range 1 to 255.

[VC.TRICORE.Interrupts.5]

Deviation of rule [\[VC.Mk_gen_config.ISR.4.12\]](#).

Verify that the value of the `interruptLevel` parameter in each invocation of `MK_ISRCFG()` is greater or equal as the interrupt level defined for the corresponding interrupt request referenced by `irqref`.

6.2.15.3. Verification criteria for memory regions on TriCore family processors

[VC.TRICORE.MemoryPartition.3]

For each memory region, verify that both its start address (`MK_RSA_*`) and its limit address (`MK_RLA_*`) are aligned on 8-byte boundaries.

[VC.TRICORE.MemoryPartition.4b]

For each configured core, verify that the sum of the number of regions in all the static memory partitions plus the number of regions in the largest dynamic memory partition does not exceed 16.

[VC.TRICORE.MemoryPartition.5]

In the memory access permissions for each memory region, verify that the access rights contains exactly one of `MK_MPERM_U_RX`, `MK_MPERM_U_RW` or `MK_MPERM_U_R`. The following table denotes the meaning of the individual constants:

Constant	Rights
<code>MK_MPERM_U_RX</code>	read and execute allowed

MK_MPERM_U_RW	read and write allowed
MK_MPERM_U_R	read allowed

[VC.TRICORE.MemoryPartition.8]

Verify that there are no memory partitions except the microkernel's own partition that grant access to the memory-mapped address space of peripherals that are reserved for exclusive use by the microkernel. The peripherals which are reserved for exclusive use by the microkernel are specified in [Section 7.13, “Supported TriCore family processors”](#).

[VC.TRICORE.MemoryPartition.9]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_NSTATICPARTITIONS` is the number of static memory partitions for core x , i.e., two plus the number of fast partitions for the core.

[VC.TRICORE.MemoryPartition.11b]

For each configured core x , verify that the value of the macro `MK_CFG_Cx_DYNREGIONS_MAX` is the maximum number of dynamic memory regions used by a dynamic memory partition that is among the partitions used by core x .

[VC.TRICORE.MemoryPartition.11c]

Verify that the value of the macro `MK_CFG_DYNREGIONS_MAX` is equal to the largest of all the `MK_CFG_Cx_DYNREGIONS_MAX` values.

6.2.15.4. Verification criteria for TriCore family processors with multiple cores

Some TriCore processors have multiple processing cores. For these processors additional verification criteria apply. These are listed in this section.

[VC.TRICORE.General.2]

Verify that the value of the macro `MK_TRICORE_TIMESTAMPSTMBASE` defines the STM module that you chose for the time stamp service. The value shall be one of `MK_STM0_BASE`, `MK_STM1_BASE` or `MK_STM2_BASE`.

[VC.TRICORE.XcoreInterrupt.1]

Verify that the software vector entry for the inter-core interrupt is `MK_SOFTVECTOR_0254`. See also `VC.Mk_gen_config.InternalISR.2` in [Section 6.2.7, “Verification criteria for internal interrupts”](#).

6.2.15.5. Verification criteria for simple schedule table on TriCore family processors

[VC.TRICORE.SST.1]

For each invocation of `MK_SSTCOUNTERCONFIG`, verify that the parameter `tkr` is set to one the following values: -1 or $2 * x + y$, where x identifies the STM and y the comparator in it. The following table shows the ticker ids (`tkr`) for three STMs.

STM	Comparator	Ticker Id
STM0	CMP0	0
STM0	CMP1	1
STM1	CMP0	2
STM1	CMP1	3
STM2	CMP0	4
STM2	CMP1	5

A comparator shall only be used to drive one SST counter at a time.

[VC.TRICORE.SST.2]

For each invocation of `MK_SSTCOUNTERCONFIG` where the parameter `tkr` is unequal to -1, verify that there is a corresponding `MK_SOFTVECTOR_nnnn` macro. `MK_SOFTVECTOR_nnnn` shall be defined to `MK_VECTOR_INTERNAL(MK_SstTickerInterruptHandler, counterId)` where `counterId` has the value `MK_CFG_NSCHEDULETABLES + x` and `x` is the index in `MK_CFG_SSTCOUNTERTABLE` of the SST identified by `tkr`.

6.3. Verification criteria for `Mk_gen_user.h`

This section lists the verification criteria for the generated file `Mk_gen_user.h`. This file contains only the identifiers for configured objects. The identifiers must correspond with the positions of the objects in the micro-kernel configuration arrays in `Mk_gen_config.h`.

[VC.Mk_gen_user.1]

For each configured ISR (including ISRs for hardware counter objects), verify that the numerical value of the ISR's identifier is identical to the index of the ISR's entry in the array initialized by the macro `MK_CFG_ISRLIST(Mk_gen_config.h)`.

[VC.Mk_gen_user.2]

For each configured resource (excluding internal resources and `RES_SCHEDULER`), verify that its ID is defined as `MK_MakeLockId(c, i)`, where `c` is the core to which the resource is allocated and `i` is the `lockIndex` of the corresponding entry in `MK_CFG_LOCKLIST(Mk_gen_config.h)`.

[VC.Mk_gen_user.2.1]

For each configured spinlock, verify that its ID expands to the same value as the `lockIndex` of the corresponding entry in `MK_CFG_LOCKLIST(Mk_gen_config.h)`. If `MK_MakeLockId(c, i)` is used, the `c` parameter shall be 0.

[VC.Mk_gen_user.2.2]

If you enabled `RES_SCHEDULER` (in EB tresos Studio `OsUseResScheduler=TRUE`), verify that the macro `RES_SCHEDULER` expands to the same value as the `lockIndex` of the corresponding entry in `MK_CFG_LOCKLIST(Mk_gen_config.h)`. If `MK_MakeLockId(c, i)` is used, the `c` parameter shall be 0.

[VC.Mk_gen_user.2.3]

Verify that the macro `MK_RESCAT1` expands to the same value as the `lockIndex` of the corresponding entry in `MK_CFG_LOCKLIST (Mk_gen_config.h)`. If `MK_MakeLockId(c, i)` is used, the `c` parameter shall be 0.

[VC.Mk_gen_user.2.4]

Verify that the macro `MK_RESCAT2` expands to the same value as the `lockIndex` of the corresponding entry in `MK_CFG_LOCKLIST (Mk_gen_config.h)`. If `MK_MakeLockId(c, i)` is used, the `c` parameter shall be 0.

[VC.Mk_gen_user.3]

For each configured task, verify that the numerical value of the task's identifier is identical to the index of the task's entry in the array initialized by the macro `MK_CFG_TASKLIST (Mk_gen_config.h)`. Indexes are zero-based.

[VC.Mk_gen_user.4]

For each configured OS-Application, verify that the numerical value of the OS-Application's identifier is equal to the index of the OS-Application's entry in the array initialized by `MK_CFG_APPLIST ((Mk_gen_config.h)`.

[VC.Mk_gen_user.5]

For each configured trusted function, verify that the numerical value of the trusted function's identifier is identical to the index of the trusted function's entry in the array initialized by `MK_CFG_TRUSTEDFUNCTIONLIST (Mk_gen_config.h)`.

6.4. Verification criteria for `Mk_gen_addons.h`

The file `Mk_gen_addons.h` contains the configuration of the installed microkernel add-ons. This section lists the verification criteria for the file `Mk_gen_addons.h`.

There are no verification criteria for derivatives in the TriCore family for the file `Mk_gen_addons.h`.

6.5. Verification criteria for `Mk_board.h`

The file `Mk_board.h` contains the definitions of time conversion macros for the time stamp timer and the execution timer. This section lists the verification criteria for the file `Mk_board.h`.

[VC.Mk_board.1]

Verify that the hardware timer from which the time stamp services are derived is configured to advance at the correct rate.

[VC.Mk_board.2]

Verify that the hardware timer from which the execution budget monitor is derived is configured to advance at the correct rate.

[VC.Mk_board.3]

Verify that the macro `MK_StampNsToTicks` evaluates to the correct number of ticks for all possible input values. The word *correct* here includes an acceptable tolerance caused by the constraint of being computable at compile time using 32-bit unsigned integer arithmetic.

[VC.Mk_board.4]

Verify that the macro `MK_StampTicksToNs` evaluates to the correct number of nanoseconds for all input values that are in range. The word "correct" here includes an acceptable tolerance caused by the constraint of being computable at compile time using 32-bit unsigned integer arithmetic.

[VC.Mk_board.5]

Verify that the macro `MK_StampTicksToNs` evaluates to `0xffffffff` for all out-of-range input values.

[VC.Mk_board.6]

Verify that the macro `MK_StampNsToTicksF` evaluates to the correct number of ticks for all possible input values.

[VC.Mk_board.7]

Verify that the macro `MK_StampTicksToNsF` evaluates to the correct number of nanoseconds for all input values that are in range.

[VC.Mk_board.8]

Verify that the macro `MK_StampTicksToNsF` evaluates to `0xffffffff` for all out-of-range input values.

[VC.Mk_board.9]

Verify that the macro `MK_ExecNsToTicks` evaluates to the correct number of ticks for all possible input values. The word *correct* here includes an acceptable tolerance caused by the constraint of being computable at compile time using at least 32-bit unsigned integer arithmetic.

NOTE



Saturation in time conversion

In [VC.Mk_board.4], [VC.Mk_board.5], [VC.Mk_board.7] and [VC.Mk_board.8]:

- ▶ The term *in range* refers to the range of input values for which a 32-bit unsigned output value can be calculated without overflow.
- ▶ The term *out-of-range* refers to the range of input values which cause an overflow of the output value.

For out-of-range values, the calculation shall go into saturation by returning the maximum value of 32-bit unsigned integer types.

TIP



Use EB's time conversion macros

If the above macros are defined in terms of the appropriate macros in `Mk_timeconversion.h`, the above criteria [VC.Mk_board.3] through [VC.Mk_board.9] are satisfied. You can request a statement of the tolerance used in the accuracy assessment from EB.

[VC.Mk_board.10]

Verify that the macro `MK_TIMESTAMPCLOCKFACTOR100U` is defined as an unsigned integer with such a value that the formula `ticks/MK_TIMESTAMPCLOCKFACTOR100U` evaluates to the correct number in units of 100 microseconds. This corresponds with the given number of ticks of the time stamp timer. If the macro `MK_HAS_TIMESTAMPCLOCKFACTOR100U` is defined to be zero, this criterion can be ignored.

[VC.Mk_board.11]

Verify that the macro `MK_TIMESTAMPCLOCKFACTOR10U` is defined as an unsigned integer with such a value that the formula `ticks/MK_TIMESTAMPCLOCKFACTOR10U` evaluates to the correct number in units of 10 microseconds. This corresponds with the given number of ticks of the time stamp timer. If the macro `MK_HAS_TIMESTAMPCLOCKFACTOR10U` is defined to be zero, this criterion can be ignored.

[VC.Mk_board.12]

Verify that the macro `MK_TIMESTAMPCLOCKFACTOR1U` is defined as an unsigned integer with such a value that the formula `ticks/MK_TIMESTAMPCLOCKFACTOR1U` evaluates to the correct number in units of microseconds. This corresponds with the given number of ticks of the time stamp timer. If the macro `MK_HAS_TIMESTAMPCLOCKFACTOR1U` is defined to be zero, this criterion can be ignored.

[VC.Mk_board.13]

For each configured core `x`, verify that the macro `MK_CFG_Cx_MSG_INVALIDHANDLER` is either not defined or is defined to be one of the following values:

- ▶ `MK_CountInvalidXcoreMessage`
- ▶ `MK_IgnoreInvalidXcoreMessage`
- ▶ `MK_PanicInvalidXcoreMessage`

7. TriCore family supplement

This section describes safety considerations that are specific to the TriCore processor family.

The processors in this family that are supported by EB tresos Safety OS are described in [Section 7.13, “Supported TriCore family processors”](#).

7.1. Using TriCore processors safely

Ensure that you meet the following prerequisites before you use EB tresos Safety OS with a TriCore processor:

- ▶ Read and fully understand the respective safety manual from the processor vendor (AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#) or AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)). Fulfill all requirements that are stated in the safety manual from the processor vendor.

CAUTION



AURIX assumptions of use

EB tresos Safety OS by itself does not fulfill any assumptions of use stated in the AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#) or AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#). Instead, the EB tresos Safety OS provides an abstraction so you can conveniently use hardware mechanisms like privilege separation and memory protection according to your application's needs.

- ▶ Read and fully understand the reference manuals for the respective microcontroller:
 - ▶ AURIX TC21x/TC22x/TC23x Family User's Manual [\[TC23X_UM\]](#)
 - ▶ AURIX TC27x B-Step User's Manual [\[TC27X_UM_B\]](#)
 - ▶ AURIX TC27x C-Step User's Manual [\[TC27X_UM_C\]](#)
 - ▶ AURIX TC29x B-Step User's Manual [\[TC29X_UM_B\]](#)
 - ▶ AURIX TC33x/TC32x User's Manual Appendix [\[TC33X_TC32X_APPX_UM_14\]](#)
 - ▶ AURIX TC36x User's Manual Appendix [\[TC36X_UM_APPX_15\]](#) (incl. [\[TC3XX_UM_1_16\]](#) and [\[TC3XX_UM_2_16\]](#))
 - ▶ AURIX TC37x Target Specification [\[TC37X_TS_V251\]](#)
 - ▶ AURIX TC38x Target Specification [\[TC38X_TS_V230\]](#)
 - ▶ AURIX TC39x-B Target Specification [\[TC39X_TS_V251\]](#)
- ▶ Read and fully understand the reference manuals for the respective processor core (TriCore TC1.6P & TC1.6E User Manual (Volume 1) [\[TC_ARCH_161_VOL1\]](#) or TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)).
- ▶ Meet the start-up conditions for EB tresos Safety OS, that are described in [Section 7.12, “Start-up preconditions”](#), before control is transferred to the entry point of the microkernel (`MK_Entry2()`).

- Reinitialize safety-related hardware or verify that the initialization has been performed correctly.

WARNING**Execution of unsafe software before start-up**

It is possible to permit unsafe software to execute before you transfer control to the microkernel. If you do so, do not place any safety allocation on initialization that was done before or while the unsafe software ran. Always verify the initialization or re-initialize after you start the microkernel.

You can perform this verification in one of the two [callout functions](#) that the microkernel calls shortly after your start-up code transfers control to EB tresos Safety OS.

Software that runs under the microkernel is permitted to use all 16 data registers (D0 to D15) and the 12 address registers A2 to A7 and A10 to A15 provided by the TriCore processor core without restrictions. The microkernel saves all these registers when it switches threads, thus it provides a virtual subset of the TriCore register set for each thread.

The global address registers (A0, A1, A8 and A9) are not available for general-purpose use. The microkernel uses A9 as the base pointer for its core-specific state variables. The compiler might use A0 and A1 as base pointers for near data. In any case, these registers are expected to be initialized once at start-up and remain constant thereafter. To protect these registers against inadvertent modification, the microkernel clears the [GW](#) in the [PSW](#) after initialization of these registers, and starts all executables by default with [GW](#) set to zero.

On microcontrollers that have lockstep cores, the microkernel is expected to be configured to execute on these cores ([[SM_AURIX_20](#)], [[AURIX_SM15_1](#)]; [[TC3XX_SM109](#)], section 3.2.4) in order to satisfy the assumption [Assumptions.ReliableExecutionEnvironment](#).

If you configure the microkernel to run on a core without lockstep ensure that you enable the MPU readback and verification feature on each core without lockstep ([[SM_AURIX_CPU_MPU_INIT_1](#)], [[AURIX_SM15_1](#)] or [ESM\[SW\]:CPU:*_MPU_CHECK](#), [[TC3XX_SM109](#)]).

The microkernel is designed to be free from fault propagation across cores, provided that the bus-level memory protection hardware is configured to protect the microkernel's variables on each core from modification by other cores. Provided that this protection is properly implemented, you can configure the microkernel to run on any combination of the available cores. However, the cores without lockstep do not provide the required reliable execution environment, so you must ensure that you allocate the applications with safety-relevant functionality to lockstep cores.

In all cases, you must configure the bus-level memory protection hardware to protect the memory used by the microkernel-based application from interference by other bus masters, including other cores. You must configure the bus-level memory protection in the most restrictive way possible, so that the data that you designate as *shared* is the only data that can be modified by more than one core. With the exception of the shared spin-lock variables, the data used by the microkernel is separated into core-specific regions. Without affecting the functionality of the microkernel, the bus-level memory protection can be configured to deny access if a core attempts to modify the microkernel data that belongs to a different core.

Most TriCore processors are equipped with both data and instruction caches. These caches do not provide hardware coherency. This means that a change of data by another bus master may not be visible to the processor core and can lead to undefined behaviour. It is therefore recommended to use the cache only for the ROM segments.

The microkernel implements neither interrupt rate monitoring nor deadline monitoring. These are beyond the scope of the microkernel and — if needed — shall be implemented at application level. The microkernel does, however, check that the values it writes to the SRC registers can be read back from the hardware successfully.

The microkernel initializes the [BTV](#) as soon as possible after control is transferred to `MK_Entry2()`, thereby taking control of the exception handling. If, however, a hardware exception occurs before this point and the microkernel nevertheless continues start-up, the integrity of the microkernel may be adversely affected. The microkernel therefore assumes that an exception in this early phase of start-up will not execute or return to any microkernel code.

During start-up (after the initialization of BTV), the microkernel assumes every processor trap to be a critical exception and shuts down the system if an exception occurs. Therefore you must ensure that external non-maskable interrupts cannot occur.

AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#) specifies some safety mechanisms (e.g., [SM_AURIX_CPU_1:2], [SM_AURIX_CPU_2:1]) that require you to create traps and check that the processor generates the expected trap. It is strongly recommended that you perform this kind of check before the start of the microkernel. If you, however, choose to perform this test after the microkernel has started, the microkernel will call the `ProtectionHook()` for every generated trap. In the `ProtectionHook()` you can inspect the variable `MK_GetProtectionInfo()->type` to verify that the correct trap has been reported by the processor, and then return `MK_PRO_TERMINATE` to terminate the thread that caused the trap.

While the microkernel is executing, it assumes every processor trap to be a critical exception and shuts down the system if this happens. Therefore you must ensure that external non-maskable interrupts cannot occur.

The hardware resources available for the configuration in EB tresos Studio may refer to the umbrella device of the respective series. Consult the hardware manual for your device for further information. For example, the TC38xQ derivative is the umbrella device for the TC38x series.

7.2. Safety of OS callout functions specific to TriCore

The following table contains the safety status of callout functions that are specific to the TriCore implementation. See also [Appendix C, “Safety of OS callout functions”](#).

Callout function	Calling method	Remarks
MK_ProtectRamFromExternal()	Jump-and-link	Memory protection is not in effect. Additional restrictions defined in Section 7.2.1, “MK_ProtectRamFromExternal” apply.

Table 7.1. Safety status of microkernel callout functions for TriCore

7.2.1. MK_ProtectRamFromExternal

The callout function `MK_ProtectRamFromExternal()` is invoked very early when the microkernel starts. Its intention is to prevent other bus masters from modifying RAM used and managed by the microkernel. For more information, see [Section 7.5, “Protection from other bus masters”](#) and [\[VC.TRICORE.Startup.3\]](#). You need to implement this function in assembly language. The implementation shall observe the following restrictions:

[VC.TRICORE.ProtectRamFromExternal.1]

Verify that your implementation of the callout function `MK_ProtectRamFromExternal()` does not change the values in the processor registers D12, D13, D14 and D15.

[VC.TRICORE.ProtectRamFromExternal.2]

Verify that your implementation of the callout function `MK_ProtectRamFromExternal()` does not use the stack.

[VC.TRICORE.ProtectRamFromExternal.3]

Verify that your implementation of the callout function `MK_ProtectRamFromExternal()` does not use any CSAs.

[VC.TRICORE.ProtectRamFromExternal.4]

Verify that your implementation of the callout function `MK_ProtectRamFromExternal()` does not use or rely on any data stored in RAM.

[VC.TRICORE.ProtectRamFromExternal.5]

Verify that your implementation of the callout function `MK_ProtectRamFromExternal()` does not exit by means of `RET` instruction. Instead, it shall jump to the address that was in register A11 on entry to the function.

7.3. Core special function registers (CSFRs)

This section describes the CSFRs that are relevant for the microkernel. Unless noted otherwise, the registers listed here are available on all supported derivatives. You can find full descriptions of these registers in the respective core and derivative manuals.

PCX/PCXI

Previous Context Information Register: This register references the last CSA used by the current thread. It is stored by the microkernel on every exception, interrupt or system call as it represents the context of the interrupted thread. No thread shall directly write to the PCXI register, i.e. by using the `mtcr` instruction.

PSW

Processor status word: Various settings are contained in this register:

USB

User Status Bits: These bits contain processor flags such as CARRY and OVERFLOW.

S

Safety Task Identifier: Selects whether the CPU core uses its *normal* or its *safety* bus master ID when accessing the data bus.

PRS

Protection Register Set: Selects the active memory protection register set.

IO

Access Privilege Level Control: Selects the processor mode (User-0, User-1, Supervisor).

IS

Interrupt Stack Control: Tells the processor whether a switch to the shared global stack is necessary on the next trap or interrupt. This flag is set to zero by the microkernel whenever a user thread is dispatched. It is set to one whenever a trap or interrupt occurs. It is used by the microkernel to detect if an exception has happened while executing the microkernel

GW

Global Address Register Write Permission: Controls if writing to the registers A0, A1, A8 and A9 is allowed. This is set to zero (i.e., writing is not allowed) at all times after the microkernel is initially started.

CDE

Call Depth Count Enable: Enables/disables the call depth counting feature.

CDC

Call Depth Counter: The current value of the call depth counter. For more details, see the core manual. No thread shall directly write to the PSW register, i.e. by using the `mtcr` instruction.

PC

Program Counter: Next instruction to be executed by the processor core.

SYSCON

System Configuration Register: Various settings are contained in this register:

U1_IOS

User-1 Peripheral access as supervisor: Defines whether threads in User-1 mode can access peripherals as if they were running in supervisor mode. This field is not changed by the microkernel and can be set as you wish. It is recommended not to change this value after the microkernel has successfully started. A good place to do so would be the `MK_InitHardwareAfterData()` callout.

U1_IED

User-1 Instruction execution disable: Defines whether threads in User-1 mode can enable and disable interrupts. This field is not changed by the microkernel and can be set as you wish in the `MK_InitHardwareAfterData()` callout. It is recommended not to change this value after the microkernel has successfully started.

TS

TS: Defines whether memory accesses by the microkernel are performed using the *normal* or the *safety* bus master ID of the core, when the microkernel has been entered via a trap. This field is not changed by the microkernel and can be set as you wish. It is recommended not to change this value after the microkernel has successfully started. A good place to do so is the `MK_InitHardwareAfterData()` callout. This should be set identical to `SYSCON.IS`. (Note: Infineon does not give a name for this flag.)

IS

IS: Defines whether memory accesses by the microkernel are performed using the *normal* or the *safety* bus master ID of the core, when the microkernel has been entered via an interrupt. This field is not changed by the microkernel and can be set as you wish. It is recommended not to change this value after the microkernel has successfully started. A good place to do so is the `MK_InitHardwareAfterData()` callout. This should be set identical to `SYSCON.TS`. (Note: Infineon does not give a name for this flag.)

TPROTEN

Temporal Protection Enable: Enables the TPS. Must be zero when the microkernel is started. For more information, see [Section 7.12, "Start-up preconditions"](#). Reserved for exclusive use by the microkernel.

PROTEN

Memory Protection Enable: Enables the memory protection. Must be zero when the microkernel is started. For more information, see [Section 7.12, "Start-up preconditions"](#). Reserved for exclusive use by the microkernel.

FCDSF

Free Context List Depleted Sticky Flag: This is not used by the microkernel.

CORE_ID

Core Identification Register: Contains a unique identifier for each core in a multi-core MCU. The microkernel uses this register to verify that it is executing on a core that you configured for use by the microkernel and to select the correct set of state variables for managing the core on which it is executing.

BIV

Base Interrupt Vector Table Pointer: Reserved for exclusive use by the microkernel.

BTV

Base Trap Vector Table Pointer: Reserved for exclusive use by the microkernel.

ISP

Interrupt Stack Pointer Register: Reserved for exclusive use by the microkernel.

ICR

ICU Interrupt Control Register: The ICR contains the following fields:

PIPN

Pending Interrupt Priority Number: Not used by the microkernel.

IE

Global Interrupt Enable Bit: You can change the value of this bit in supervisor and (optionally) user-1 mode threads by using dedicated assembler instructions.

CCPN

Current CPU Priority Number: Only interrupts with a priority higher than the CCPN can interrupt the CPU. This field is managed by the CPU and the microkernel.

No thread shall directly write to the ICR register, i.e. by using the `mtcr` instruction.

FCX

Free CSA List Head Pointer Register: This register references the next unused CSA. No thread shall directly write to the FCX register, i.e. by using the `mtcr` instruction.

LCX

Free CSA List Limit Pointer: Reserved for exclusive use by the microkernel.

COMPAT

Compatibility Control Register: Controls various derivative-specific compatibility settings of the processor. Reserved for exclusive use by the microkernel.

DPR_n*L/DPR_n*U

Data Protection Range Lower/Upper Bound Registers: These registers define the memory protection regions for data accesses. Reserved for exclusive use by the microkernel.

CPR_n*L/CPR_n*U

Code Protection Range Lower/Upper Bound Registers: These registers define the memory protection regions for instruction fetches. Reserved for exclusive use by the microkernel.

CPXE_n

Code Protection Execute Enable Set Configuration Registers: These registers assign instruction access rights to protection register sets. Reserved for exclusive use by the microkernel.

DPRE_n

Data Protection Read Enable Set Configuration Registers: These registers assign data read access rights to protection register sets. Reserved for exclusive use by the microkernel.

DPWE_n

Data Protection Write Enable Set Configuration Registers: These registers assign data write access rights to protection register sets. Reserved for exclusive use by the microkernel.

TPS_CON

Temporal Protection System Control Register: Reserved for exclusive use by the microkernel.

TPS_TIMER_n

TPS Timer Registers: TPS decrementers. Reserved for exclusive use by the microkernel.

DSTR

DMI Synchronous Trap Flag Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

DATR

DMI Asynchronous Trap Flag Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

DEADD

Data Error Address Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

DIEAR

Data Integrity Error Address Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

DIETR

Data Integrity Error Trap Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

PSTR

Program Memory Interface Synchronous Trap Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

PIEAR

Program Integrity Error Address Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

PIETR

Program Integrity Error Trap Register: This register contains information generated by the processor on certain traps. Its value is stored by the microkernel if one of those traps occurs, so that it can be inspected by user software. For more details, see EB tresos Safety OS user's guide [\[SAFEOSUSRDOC\]](#).

7.4. Memory protection unit

The microkernel uses the [MPU](#) as part of its protection mechanism to provide fine-grained control over memory accesses. The MPU is critical to the correct operation of the microkernel, so the microkernel reserves the MPU for its own exclusive use. Non-privileged mode threads are prevented from accessing the MPU by the hardware.

The microkernel is not able to prevent privileged mode threads from accessing and modifying the current MPU settings, so you must develop and verify all privileged mode threads to the highest safety standards. Incorrect modification of *any* register in the MPU could affect the microkernel's protection mechanism.

On AURIX processors, it is possible that the microkernel is executed on processor cores without a lockstep core. In this case AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#) and AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#) require the software to read back and verify the MPU configuration after the MPU is programmed (safety mechanism [SM_AURIX_CPU_MPU_INIT_1] or ESM[SW]:CPU:DATA_MPU_CHECK). The microkernel implements this check; however it is disabled by default. To activate this check, you need to set the macro `MK_CFG_Cx_MPU_REGISTER_CHECK` to 1 for the cores on which the check is needed.

In the MPU, memory regions have granularity and alignment of 8 bytes. If a memory region has an unaligned lower or upper boundary, the MPU truncates the respective boundary address to be 8-byte-aligned. Therefore the memory regions shall be aligned on 8-byte boundaries by the linker. In the TC1.6.2 MPU, code regions have granularity and alignment of 32 bytes. Therefore, code regions shall be aligned on 32-byte boundaries by the linker on TC1.6.2 derivatives.

The MPU only checks the starting address of a memory access. This means that multi-word accesses can access memory beyond the upper boundary defined in the MPU. There are instructions to read and write whole CSAs (i.e., 64 bytes) from/to memory at once. These instructions can read/write up to 56 bytes beyond a memory region. You shall ensure that the linker provides enough padding space between memory regions so that no data of other memory regions can be written.

[VC.TRICORE.MPU.1]

Verify that the lower boundary of each memory region is aligned to a 64-byte boundary.

[VC.TRICORE.MPU.2]

Verify that the upper boundary of each memory region is aligned to an 8-byte boundary.

[VC.TRICORE.MPU.3]

Verify that there is a gap of at least 8 bytes between every pair of adjacent memory regions.

[VC.TRICORE.MPU.4]

Verify that the upper boundary of each code region is aligned to a 32-byte boundary on TC1.6.2 derivatives.

7.5. Protection from other bus masters

An important aspect to ensure freedom from interference is to prevent unintentional accesses from other bus masters. The TriCore MPU as described in [Section 7.4, “Memory protection unit”](#) is logically connected between the processor core and the memory bus of the MCU. Therefore it cannot be used to control memory accesses triggered by other bus masters. It is therefore up to you to prevent such unintentional accesses. For more information, see [\[VC.TRICORE.Startup.3\]](#)). Depending on the derivative, the hardware provides different mechanisms that you can apply. These are presented in the following subsections.

All the currently supported derivatives of the AURIX line of processors (TC22xL, TC23xL, TC27xT and TC29xT) and AURIX2G processors (TC33x, TC36x, TC37x, TC38x, TC39x) have bus MPUs and peripheral access protection.

Each memory module on the system bus has its own memory MPU. These memory MPUs provide eight memory regions with a granularity of 32 or 64 bytes that can be used to restrict write accesses to the respective memory module. Read accesses are always allowed on TC2XX. For more details see [\[TC23X_UM\]](#), Section 6.12.6, [\[TC27X_UM_C\]](#), Sections 5.12.6 and 11.3 and [\[TC29X_UM_B\]](#), Sections 5.12.6 and 11.3. On TC3XX read accesses can be restricted, too. For more details see [\[TC3XX_TS_1_201\]](#) section 11.3.3.4.2 "SRAM Protection".

7.6. Interrupt router

The interrupt router (IR) of the AURIX processors is the central component that arbitrates the peripheral's interrupt requests and dispatches them to the respective processor cores. The main part of the IR comprises the service request control registers (SRCs) of the individual interrupt sources. These registers are used to configure settings like interrupt priority and the core to which this interrupt is signalled.

The SRCs are subject to register access protection. However, this protection affects all the SRCs as a whole. It is not possible to protect individual SRCs from specific bus masters. This has a serious implication regarding freedom from interference. Since each interrupt source is effectively assigned to a specific core, only that core needs to access the SRC register for that interrupt source (e.g., to set the priority). But since the register access protection affects the IR as a whole, all cores need to be given access to all SRC registers. As a consequence, every core with access to the IR is able to interfere with each other core's interrupt configuration. This could happen as a result of either a systematic fault in the software executing on any core or a sporadic fault of a non-lockstep core.

On TC3XX processors you can protect SRC bits 31 down to 16 depending on the target which handles them. However, there is no fine-grained protection for SRC bits 15 down to 0, which contain priority, target and locking of the interrupt. See [\[TC3XX_TS_1_201\]](#) section 17.4.1.3 "Protection of the SRC registers".

If such a fault occurs and the IR configuration is modified as a consequence, the effect on the system is one of the following:

- ▶ ISRs are not executed when the corresponding interrupt occurs.
- ▶ ISRs are executed, even though the corresponding interrupt did not occur.
- ▶ The microkernel fails to execute cross-core service requests.
- ▶ The microkernel shuts down with the panic code `MK_panic_UnexpectedInterrupt`.

You can detect the first three of these errors by applying flow control monitoring and rate monitoring on your application.

[VC.TRICORE.MK_CFG_OVERRIDE_INT_IR_CONFIG_DRIVER.1]

Verify that macro `MK_CFG_OVERRIDE_INT_IR_CONFIG_DRIVER` is either undefined or set to a value equal to 0.

7.7. Code placement

There are no special requirements for the placement of program code in ROM on TriCore. There are restrictions for placement of the kernel entry points, i.e. the interrupt and exception tables, but these are addressed in the microkernel code and need no special action on your part.

7.8. Linking requirements

Besides the restrictions for memory regions described in [Section 7.4, “Memory protection unit”](#), there are some symbols that have additional requirements you need to consider when you create your linker script:

Linking requirements for CSAs

[VC.TRICORE.Placement.3]

For each configured core x , verify that the symbol `MK_RSA_MK_Csa_Cx` is placed at an address that is divisible by 64.

[VC.TRICORE.Placement.4]

For each configured core x , verify that the symbol `MK_RLA_MK_Csa_Cx` is placed at an address that is divisible by 64 and is at least 1152 bytes above `MK_RSA_MK_Csa_Cx`.

[VC.TRICORE.Placement.6]

If you place the CSA region at the core's private address for its local memory, verify that the physical memory that you reserved for CSAs cannot be used for any other purpose through a mapping to a different address.

The CSAs allocated for each core are shared between all tasks, ISRs and other threads that are executed on that core. If you do not allocate sufficient CSAs for your application, the processor might trigger a `FCX=LCX` trap (class 3, TIN 1), which causes the microkernel to shut down. The occurrence of the trap may happen very infrequently, when some rare combination of task, ISR and other thread nesting happens. It is therefore important to ensure that you have sufficient CSAs for the worst case nesting, or that the shutdown resulting from the `FCX=LCX` can always be handled safely.

[VC.TRICORE.Placement.7]

Verify that the number of CSAs that you allocated is sufficient for the worst case of task, interrupt and other thread nesting that is possible in your application.

NOTE**Call-depth counting**

To prevent uncontrolled allocation of CSAs, e.g. by unbounded recursion, it is recommended that you use the call-depth counting mechanism of the processor to limit the maximum call-depth within tasks and ISRs.

Compiler specific linking requirements

The Wind River® Diab Compiler offers a stack smashing protection feature which can be activated via the compiler flags `-Xstack-protection`, `-Xstack-protection-strong`, or `-Xstack-protection-all`. If this feature is active it causes additional requirements you need to consider when you create your linker script:

[VC.TRICORE.Placement.Diab.1]

Verify that the symbol `__stack_chk_guard` is placed in a read-only memory region if the Diab compiler is used and stack smashing protection is active.

Note: After any core entered `MK_Entry2()`, the content of this symbol must not be changed by anybody.

7.9. Verification measures for hardware

The microkernel relies on hardware features in order to detect software faults. If the hardware has developed a failure that prevents a software fault from being detected, a subsequent software fault may place the system in an unsafe state.

To defend against this, the hardware safety manual requires verification measures to be performed.

- ▶ Verify that the CPU trap system is working correctly. To perform the test, generate a trap of each trap class that shall be tested and check that the correct trap handler gets invoked with the correct trap identification number. For more information, see AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#) Section 5.2.2.

Do not attempt to test the SYSCALL trap (class 6) or the `PCXI=0` trap (class 3, TIN 5) in this way. The microkernel uses these traps for normal operation and they do not result in an activation of `Protection-Hook()`.

Memory protection traps (most of class 1) are covered by MPU verification.

- ▶ Verify that the MPU is working correctly. The following test cases shall be considered:
 - ▶ Read accesses to a protected memory region are blocked and a MPR trap is generated.
 - ▶ Write accesses to a protected memory region are blocked and a MPW trap is generated.
 - ▶ Read accesses to a memory region which are not covered by any Data Protection Range are blocked and a MPR trap is generated.

- ▶ Write accesses to a memory region which are not covered by any Data Protection Range are blocked and a MPW trap is generated.
- ▶ Read accesses to an unprotected memory region are not blocked and no trap is generated.
- ▶ Write accesses to an unprotected memory region are not blocked and no trap is generated.
- ▶ Program fetches from a protected memory region are blocked and a MPX trap is generated.
- ▶ Program fetches from an unprotected memory region are not blocked and no trap is generated.

For more information, see AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#) Section 5.2.19.

All traps are handled by the microkernel and result in your `ProtectionHook()` being called. Your implementation of `ProtectionHook()` must verify that the correct trap has been signalled by the hardware. The protection information structure (obtained by calling `MK_GetProtectionInfo()`) contains all the information you need to verify that the trap was caused by a test function and was signalled correctly.

On a successful test, `ProtectionHook()` should return `MK_PRO_TERMINATE` to terminate the test thread that caused the exception.

7.10. Hardware timers

7.10.1. The System Timer (STM)

The microkernel uses the free-running counter of one of the System Timer Modules (STM) to provide the `MK_ReadTime()` service and execution budget monitoring. The STM provides a 64-bit free running timer on all supported derivatives. On derivatives with more than one STM, you can select which STM the microkernel shall use.

The microkernel does not enable the STM by itself, because only the system integrator is able to select an appropriate frequency.

[VC.TRICORE.STM.1]

If you use the `MK_ReadTime()` service and/or execution budget monitoring, verify that the STM you chose is enabled, at the latest in the `MK_InitHardwareAfterData()` callout.

If the microkernel uses the STM, you must ensure that no thread disables or resets the STM. Therefore, it is recommended that no thread is granted write permissions to the STM's registers.

[VC.TRICORE.STM.2]

If you use the `MK_ReadTime()` service and/or execution budget monitoring, you must implement the safety mechanisms defined in the processor's safety manual. For AURIX processors these are `[SM_`

AURIX_STM_1] and [SM_AURIX_STM_3:1] as defined in [\[AURIX_SM15_1\]](#). For AURIX2G processors this is ESM[SW]:STM:MONITOR as defined in [\[TC3XX_SM109\]](#).

7.10.2. The Temporal Protection System (TPS)

The microkernel uses the free running counter of a System Timer Module (STM) as the basis for the execution budget monitoring, i.e. all the calculations of the budget are internally calculated in ticks of the STM. However, to interrupt a thread that has used all its execution budget, the Temporal Protection System (TPS) of the processor core is used. The TPS doesn't always use the same clock frequency as the STM. It is therefore necessary to convert time durations between STM ticks and TPS ticks.

[VC.TRICORE.MK_ExecutionTicksToTpsTicks]

Verify that the macro named `MK_ExecutionTicksToTpsTicks(ticks)` in the header file `Mk_board.h` takes a number of STM ticks as parameter and calculates the number of TPS ticks that represent the same amount of real time. This calculation shall be saturated, i.e., if the result is bigger than can be represented by a 32-bit unsigned integer, the macro shall return 4294967295.

7.11. TriCore-specific processor status

As described in [Section 7.3, "Core special function registers \(CSFRs\)"](#), the TriCore PSW contains some fields which you can configure according to your application's needs. This is done using the `hwSpecificPs` parameter of the respective thread construction macros (see [Section 6.2, "Verification criteria for Mk_gen_config.h"](#)). The following elements can be configured using the `hwSpecificPs` parameter:

- ▶ The protection register set (PRS) used for the thread.
- ▶ The safety task identifier, if used for the thread
- ▶ The call-depth counter configuration for the thread.
- ▶ The FPU rounding mode used in the thread.

The actual value for the `hwSpecificPs` parameter is constructed as a bitwise OR of multiple individual terms defining the choices for the elements stated above.

[VC.TRICORE.HwPs.1]

Verify that there is one term selecting the correct PRS for the thread. This is `MK_PSW_PRS_1` for regular threads and either `MK_PSW_PRS_2` or `MK_PSW_PRS_3` for threads using a fast partition.

[VC.TRICORE.HwPs.2]

Verify that the term `MK_PSW_PRS_0` is not present.

[VC.TRICORE.HwPs.3]

Verify that the term `MK_THRHWS_SAFEID` is present if and only if the thread shall use the "safe" busmaster id for memory accesses.

[VC.TRICORE.HwPs.4]

Verify that the term `MK_PSW_CDC_DIS` is present, if the thread shall execute without call-depth limitations.

[VC.TRICORE.HwPs.5]

Verify that the term `MK_PSW_CDE` is present, if the thread shall execute with the call-depth counting enabled. In that case, also verify that a numerical term in the range from 1 to 64 is present. The value of this term is calculated as $(64-x)$, where x is the intended call-depth limit.

[VC.TRICORE.HwPs.6]

Verify that exactly one of the following four terms is present: `MK_PSW_RM_ROUNDNEAREST`, `MK_PSW_RM_ROUNDUP`, `MK_PSW_RM_ROUNDDOWN` or `MK_PSW_RM_ROUNDZERO`.

7.12. Start-up preconditions

The following list contains start-up preconditions for the microkernel. Each precondition in the list specifies by which phase of the start-up you must fulfill the precondition and any additional restrictions that may apply if you use a callout function.

[VC.TRICORE.Startup.1]

Verify that your start-up code disables the memory protection system before calling `MK_Entry2()`. The `PROTEN` field in the `SYSCON` register controls memory protection. The default value after reset is zero (disabled).

[VC.TRICORE.Startup.2]

Verify that your start-up code disables the Temporal Protection System (TPS) before calling `MK_Entry2()`. The `TPROTEN` field in the `SYSCON` register controls temporal protection. The default value after reset is zero (disabled).

[VC.TRICORE.Startup.3]

For each configured core x , verify that other bus masters (including other cores) cannot modify any memory-mapped registers exclusively managed by the microkernel on core x .

For each configured core x , verify that other bus masters (including other cores) cannot modify any memory exclusively managed by the microkernel on core x .

For each configured core x , verify that other bus masters (excluding other cores that are controlled by the microkernel) cannot modify any shared memory managed by the microkernel.

You must do this at the latest in the callout function `MK_ProtectRamFromExternal()`.

Example: Spinlocks are shared memory that may only be modified by cores controlled by the microkernel. Other bus masters may not access them.

[VC.TRICORE.Startup.4]

Before you call `MK_Entry2()`, verify that all interrupt sources that are configured for use by the microkernel are disabled.

An interrupt source is disabled when the `SRE` field in its `SRC` is zero. This is the default state after reset.

[VC.TRICORE.Startup.4.1]

Before you call `MK_Entry2()`, verify that all interrupt sources that are *not* configured for use by the microkernel are either disabled or *not* programmed to interrupt a core on which you have configured the microkernel.

The TOS field in an interrupt's SRC determines which core is interrupted.

[VC.TRICORE.Startup.5]

Verify that the processor executing in supervisor mode when your start-up code calls `MK_Entry2()`.

[VC.TRICORE.Startup.6]

Verify that the CPU and STM clocks in the CCU module are configured to use a stable clock source and that the frequencies match the definition of the following conversion macros in the safety-related board header `Mk_board.h`:

- ▶ `MK_ExecutionNsToTicks()`
- ▶ `MK_TimestampNsToTicks()`
- ▶ `MK_TimestampTicksToNs()`
- ▶ `MK_TimestampNsToTicksF()`
- ▶ `MK_TimestampTicksToNsF()`

You must fulfill this precondition at the latest in the callout function `MK_InitHardwareAfterData()`.

Failure to satisfy the above preconditions might result in a failure of the system some time later. There should be no safety implications, provided that you configured the microkernel correctly and that timing protection measures are in place.

The microkernel refuses to start unless control is transferred to `MK_Entry2()`.

7.13. Supported TriCore family processors

The following list describes the microcontrollers in the TriCore family that EB tresos Safety OS supports.

TC22xL. The TC22xL microcontroller contains a TC1.6E processor core which is accompanied by a lock-step core. The TC22xL has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with 8 instruction protection areas and 16 data protection areas.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with 3 channels (1 is used).

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC22xL microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.2, “TC22xL peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0000000 to 0xF00000FF	Used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel

Table 7.2. TC22xL peripherals

TC23xL. The TC23xL microcontroller contains a TC1.6E processor core which is accompanied by a lock-step core. The TC23xL has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with 8 instruction protection areas and 16 data protection areas.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with 3 channels (1 is used).

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC23xL microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.3, “TC23xL peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0000000 to 0xF00000FF	Used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can

Name	Address range(s)	Remarks
		be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel

Table 7.3. TC23xL peripherals

TC27xT. The TC27xT microcontroller contains three processor cores in total: a TC1.6E processor core which is accompanied by a lockstep core, a TC1.6P processor core which is accompanied by a lockstep core, and a TC1.6P processor core without a lockstep core. The TC27xT has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with eight instruction protection areas and 16 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC27xT microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.4, “TC27xT peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0000000 to 0xF00000FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0000100 to 0xF00001FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM2	0xF0000200 to 0xF00002FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.

Name	Address range(s)	Remarks
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, “Protection from other bus masters” .
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, “Protection from other bus masters” .
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.

Table 7.4. TC27xT peripherals

TC29xT. The TC29xT microcontroller contains three processor cores in total: a TC1.6P processor core which is accompanied by a lockstep core and two TC1.6P processor cores without lockstep cores. The TC29xT has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with eight instruction protection areas and 16 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC29xT microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.5, “TC29xT peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0000000 to 0xF00000FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0000100 to 0xF00001FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM2	0xF0000200 to 0xF00002FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, “Protection from other bus masters” .
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, “Protection from other bus masters” .

Name	Address range(s)	Remarks
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.

Table 7.5. TC29xT peripherals

TC33xL. The TC33xL microcontroller contains a TC1.6.2 processor core which is accompanied by a lock-step core. The TC33xL has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC33xL microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.6, “TC33xL peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel.

Table 7.6. TC33xL peripherals

TC36xD. The TC36xD microcontroller contains two processor cores in total. Both of them are TC1.6.2 and lockstep cores, so they are accompanied by a checker core. The TC36xD has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC36xD microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.7, “TC36xD peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0001100 to 0xF00011FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, “Protection from other bus masters” .

Name	Address range(s)	Remarks
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.

Table 7.7. TC36xD peripherals

TC37xT. The TC37xT microcontroller contains three processor cores in total. All of them are TC1.6.2 cores. Two of them are lockstep cores, so they are accompanied by a checker core. The TC37xT has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC37xT microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.8, “TC37xT peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0001100 to 0xF00011FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM2	0xF0001200 to 0xF00012FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.

Name	Address range(s)	Remarks
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, “Protection from other bus masters” .
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, “Protection from other bus masters” .
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.

Table 7.8. TC37xT peripherals

TC38xT. The TC38xT microcontroller contains three processor cores in total. All of them are TC1.6.2 cores. Two of them are lockstep cores, so they are accompanied by a checker core. The TC38xT has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC38xT microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.9, “TC38xT peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0001100 to 0xF00011FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.

Name	Address range(s)	Remarks
STM2	0xF0001200 to 0xF00012FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM3	0xF0001300 to 0xF00013FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, "Protection from other bus masters" .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, "Protection from other bus masters" .
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, "Protection from other bus masters" .
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.

Table 7.9. TC38xT peripherals

TC38xQ. The TC38xQ microcontroller contains four processor cores in total. All of them are TC1.6.2 cores. Two of them are lockstep cores, so they are accompanied by a checker core. The TC38xQ has the following hardware characteristics that are relevant for the operation of the microkernel:

- Support for system register protection, execution protection, data protection and peripheral protection.

- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC38xQ microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.10, “TC38xQ peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0001100 to 0xF00011FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM2	0xF0001200 to 0xF00012FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM3	0xF0001300 to 0xF00013FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, “Protection from other bus masters” .

Name	Address range(s)	Remarks
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, “Protection from other bus masters” .
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.
CPU3 SFR	0xF8860000 to 0xF886FFFF	Configuration requirements defined by the microkernel if running on core 3. See Section 7.5, “Protection from other bus masters” .
CPU3 CSFR	0xF8870000 to 0xF887FFFF	Reserved for exclusive use by the microkernel if running on core 3.

Table 7.10. TC38xQ peripherals

TC39xX. The TC39xX microcontroller contains six processor cores in total. All of them are TC1.6.2 cores. Four of them are lockstep cores, so they are accompanied by a checker core. The TC39xX has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC39xX microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.11, “TC39xX peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0001100 to 0xF00011FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM2	0xF0001200 to 0xF00012FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.

Name	Address range(s)	Remarks
STM3	0xF0001300 to 0xF00013FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM4	0xF0001400 to 0xF00014FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM5	0xF0001500 to 0xF00015FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, “Protection from other bus masters” .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, “Protection from other bus masters” .
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, “Protection from other bus masters” .
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.

Name	Address range(s)	Remarks
CPU3 SFR	0xF8860000 to 0xF886FFFF	Configuration requirements defined by the microkernel if running on core 3. See Section 7.5, “Protection from other bus masters” .
CPU3 CSFR	0xF8870000 to 0xF887FFFF	Reserved for exclusive use by the microkernel if running on core 3.
CPU4 SFR	0xF8880000 to 0xF888FFFF	Configuration requirements defined by the microkernel if running on core 4. See Section 7.5, “Protection from other bus masters” .
CPU4 CSFR	0xF8890000 to 0xF889FFFF	Reserved for exclusive use by the microkernel if running on core 4.
CPU5 SFR	0xF88C0000 to 0xF88CFFFF	Configuration requirements defined by the microkernel if running on core 5. See Section 7.5, “Protection from other bus masters” .
CPU5 CSFR	0xF88D0000 to 0xF88DFFFF	Reserved for exclusive use by the microkernel if running on core 5.

Table 7.11. TC39xX peripherals

TC39xQ. The TC39xQ microcontroller contains four processor cores in total. All of them are TC1.6.2 cores. TC39xQP: All of them are lockstep cores. TC39xQA: Three of them are lockstep cores. Lockstep cores are accompanied by a checker core. The TC39xQ has the following hardware characteristics that are relevant for the operation of the microkernel:

- ▶ Support for system register protection, execution protection, data protection and peripheral protection.
- ▶ Memory protection unit (MPU) with ten instruction protection areas and 18 data protection areas per core.
- ▶ Interrupt controller with 255 interrupt levels.
- ▶ Temporal Protection System (TPS) with three channels per core. One channel is used by the microkernel.

The microkernel requires special handling or exclusive access to some of the peripheral modules of the TC39xQ microcontroller. These peripheral modules, along with their address ranges are listed in [Table 7.12, “TC39xQ peripherals”](#). All other modules are not used by the microkernel.

Name	Address range(s)	Remarks
STM0	0xF0001000 to 0xF00010FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM1	0xF0001100 to 0xF00011FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.

Name	Address range(s)	Remarks
STM2	0xF0001200 to 0xF00012FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
STM3	0xF0001300 to 0xF00013FF	Can be used by the microkernel for the time stamp and execution budget. You can use the comparators for application purposes.
SCU	0xF0036000 to 0xF00363FF	Partly used by the microkernel (ENDINIT). Example code is shipped in the delivery that can be called at start-up to set the clock frequencies.
IR	0xF0037000 to 0xF0039FFF	The registers that are configured for use by the microkernel are reserved for exclusive use by the microkernel. The registers that are not reserved are available for other purposes but must not be programmed to trigger interrupts on cores that are controlled by the microkernel.
CPU0 SFR	0xF8800000 to 0xF880FFFF	Configuration requirements defined by the microkernel if running on core 0. See Section 7.5, "Protection from other bus masters" .
CPU0 CSFR	0xF8810000 to 0xF881FFFF	Reserved for exclusive use by the microkernel if running on core 0.
CPU1 SFR	0xF8820000 to 0xF882FFFF	Configuration requirements defined by the microkernel if running on core 1. See Section 7.5, "Protection from other bus masters" .
CPU1 CSFR	0xF8830000 to 0xF883FFFF	Reserved for exclusive use by the microkernel if running on core 1.
CPU2 SFR	0xF8840000 to 0xF884FFFF	Configuration requirements defined by the microkernel if running on core 2. See Section 7.5, "Protection from other bus masters" .
CPU2 CSFR	0xF8850000 to 0xF885FFFF	Reserved for exclusive use by the microkernel if running on core 2.
CPU3 SFR	0xF8860000 to 0xF886FFFF	Configuration requirements defined by the microkernel if running on core 3. See Section 7.5, "Protection from other bus masters" .

Name	Address range(s)	Remarks
CPU3 CSFR	0xF8870000 to 0xF887FFFF	Reserved for exclusive use by the microkernel if running on core 3.

Table 7.12. TC39xQ peripherals

7.14. Safety-related documentation used during development

The following documents are used as a basis for implementing the TriCore-specific port of the microkernel. Keep in mind that you must always consult the newest version of these documents when you implement, validate or verify your system.

7.14.1. TC22xL

- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 1) [\[TC_ARCH_161_VOL1\]](#)
- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 2) [\[TC_ARCH_161_VOL2\]](#)
- ▶ AURIX TC21x/TC22x/TC23x Family User's Manual [\[TC23X_UM\]](#)
- ▶ AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#)

7.14.2. TC23xL

- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 1) [\[TC_ARCH_161_VOL1\]](#)
- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 2) [\[TC_ARCH_161_VOL2\]](#)
- ▶ AURIX TC21x/TC22x/TC23x Family User's Manual [\[TC23X_UM\]](#)
- ▶ TC233 / TC234 / TC237 A-Step Target Data Sheet [\[TC23X_DS\]](#)
- ▶ AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#)

7.14.3. TC27xT

- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 1) [\[TC_ARCH_161_VOL1\]](#)
- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 2) [\[TC_ARCH_161_VOL2\]](#)
- ▶ AURIX TC27x C-Step User's Manual [\[TC27X_UM_C\]](#)

- ▶ AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#)

7.14.4. TC29xT

- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 1) [\[TC_ARCH_161_VOL1\]](#)
- ▶ TriCore TC1.6P & TC1.6E User Manual (Volume 2) [\[TC_ARCH_161_VOL2\]](#)
- ▶ AURIX TC29x B-Step User's Manual [\[TC29X_UM_B\]](#)
- ▶ AURIX Safety Manual - AP32224 [\[AURIX_SM15_1\]](#)

7.14.5. TC33xL

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1_11\]](#)
- ▶ AURIX TC3xx User's Manual (part 1) [\[TC3XX_UM_1_14\]](#)
- ▶ AURIX TC3xx User's Manual (part 2) [\[TC3XX_UM_2_14\]](#)
- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM106\]](#)
- ▶ TC33x Target Data Sheet / Target Specification [\[TC33X_DS_06\]](#)
- ▶ TC33x/TC32x Data Sheet Addendum [\[TC33X_TC32X_DS_AD_13\]](#)
- ▶ AURIX TC33x/TC32x User's Manual Appendix [\[TC33X_TC32X_APPX_UM_14\]](#)

7.14.6. TC36xD

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)
- ▶ AURIX TC3xx Target Specification [\[TC3XX_TS_1_210\]](#)
- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)
- ▶ TC36x AA-Step Target Data Sheet / Target Specification [\[TC36X_DS_10_AA\]](#)
- ▶ TC36x Data Sheet Addendum [\[TC36X_DS_AD_13\]](#)
- ▶ AURIX TC36x User's Manual Appendix [\[TC36X_UM_APPX_15\]](#)

7.14.7. TC37xT

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)
- ▶ AURIX TC3xx Target Specification [\[TC3XX_TS_1_210\]](#)

- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)
- ▶ AURIX TC37x Target Specification [\[TC37X_TS_V251\]](#)
- ▶ AURIX TC37xED Target Specification [\[TC37XED_TS_V251\]](#)
- ▶ TC37x Target Data Sheet / Target Specification [\[TC37X_DS_10_AA\]](#)
- ▶ TC37x Data Sheet Addendum [\[TC37X_DS_AD_13\]](#)

7.14.8. TC38xT

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)
- ▶ AURIX TC3xx Target Specification [\[TC3XX_TS_1_210\]](#)
- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)
- ▶ AURIX TC38x Target Specification [\[TC38X_TS_V251\]](#)
- ▶ TC38x Target Data Sheet / Target Specification [\[TC38X_DS_10\]](#)

7.14.9. TC38xQ

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)
- ▶ AURIX TC3xx Target Specification [\[TC3XX_TS_1_210\]](#)
- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)
- ▶ AURIX TC38x Target Specification [\[TC38X_TS_V251\]](#)
- ▶ TC38x Target Data Sheet / Target Specification [\[TC38X_DS_10\]](#)

7.14.10. TC39xX

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)
- ▶ AURIX TC3xx Target Specification [\[TC3XX_TS_1_210\]](#)
- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)
- ▶ AURIX TC39x-B Target Specification [\[TC39X_TS_V251\]](#)
- ▶ TC39x Target Data Sheet / Target Specification [\[TC39X_DS_06\]](#)

7.14.11. TC39xQ

- ▶ TriCore TC 1.6.2 core architecture manual (Volume 1) [\[TC_ARCH_162_VOL1\]](#)

- ▶ AURIX TC3xx Target Specification [\[TC3XX_TS_1_210\]](#)
- ▶ AURIX TC3xx Safety Manual [\[TC3XX_SM109\]](#)
- ▶ AURIX TC39x-B Target Specification [\[TC39X_TS_V251\]](#)
- ▶ TC39xB Target Data Sheet / Target Specification [\[TC39XB_DS_11\]](#)

Appendix A. Reserved identifiers

The identifiers listed in the following table are defined by the AUTOSAR OS specification. You must not use them as names for any user-defined object.

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

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

▶ GetEvent	▶ MemorySizeType	▶ RES_SCHEDULER
▶ GetISRID	▶ MemoryStartAddressType	▶ RESTART
▶ GetNumberOfActivatedCores	▶ ObjectAccessType	▶ RUNNING
▶ GetResource	▶ ObjectTypeType	▶ SCHEDULETABLE_NEXT
▶ GetScheduleTableStatus	▶ OSServiceIdType	▶ SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS
▶ GetSpinlock	▶ PhysicalTimeType	▶ SCHEDULETABLE_RUNNING
▶ GetTaskID	▶ ProtectionReturnType	▶ SCHED- ULETABLE_STOPPED
▶ GetTaskState	▶ ResourceType	▶ SCHEDULETABLE_WAITING
▶ IncrementCounter	▶ RestartType	▶ SUSPENDED
▶ NextScheduleTable	▶ ScheduleTableStatusRefType	▶ WAITING
▶ OSErrorGetServiceId	▶ ScheduleTableStatusType	▶ ALARMCALLBACK
▶ ReleaseResource	▶ ScheduleTableType	▶ ISR
▶ ReleaseSpinlock	▶ SpinlockIdType	▶ TASK
▶ ResumeAllInterrupts	▶ StatusType ^a	▶ OSMEMORY_IS_EXE- CUTABLE
▶ ResumeOSInterrupts	▶ TaskRefType	▶ OSMEMORY_IS_READABLE
▶ Schedule	▶ TaskStateRefType	▶ OSMEMORY_IS_STACKS- PACE
▶ SetAbsAlarm	▶ TaskStateType	▶ OSMEMORY_IS_WRITE- ABLE
▶ SetEvent	▶ TaskType	
▶ SetRelAlarm	▶ TickRefType	
▶ SetScheduleTableAsync	▶ TickType	
▶ ShutdownAllCores	▶ TrustedFunctionIndexType	
▶ ShutdownOS	▶ TrustedFunctionParameter- RefType	
▶ StartCore	▶ TryToGetSpinlockType	
▶ StartNonAutosarCore		
▶ StartOS	▶ DeclareAlarm	
	▶ DeclareEvent	
	▶ DeclareResource	
	▶ DeclareTask	

^aYou can define the `E_OK` and `StatusType` by an AUTOSAR module other than the OS if you adhere to the requirements of the OSEK/VDX binding specification [\[OSEKOS142BIND\]](#).

^bAll AUTOSAR and OSEK/VDX error code identifiers.

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

GetStackInfo	SimTimerAdvance	InitCpuLoad
--------------	-----------------	-------------

Appendix A. Reserved identifiers

GetCpuLoad	ISR1
------------	------

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

Appendix B. Safety of OS services

NOTE

Integrity category

The meanings of the categories used in the *Integrity category* column in the following table are defined in [Section 4.6.3, “EB tresos Safety OS services”](#)

Service name	Integrity category	Remarks
ActivateTask	1	
CallTrustedFunction	1	The integrity category refers to calling the trusted function and passing the parameter. No statement is made about the user-defined functionality of the trusted function that is called.
AllowAccess	1	
CancelAlarm	2	
ChainTask	1	
CheckIsrMemoryAccess		Not implemented
CheckObjectAccess		Not implemented
CheckObjectOwnership		Not implemented
CheckTaskMemoryAccess		Not implemented
ClearEvent	1	
ControlIdle		Not implemented
DisableAllInterrupts	1	
MK_DisableInterruptSource	1	
EnableAllInterrupts	1	
MK_EnableInterruptSource	1	
GetActiveApplicationMode	3	
GetAlarmBase	3	
GetAlarm	2	
GetApplicationID	1	
GetCoreID	1	
GetCounterValue	2	
GetCurrentApplicationID	1	

Service name	Integrity category	Remarks
GetElapsedCounterValue	3	Implemented as a QM function in the microkernel; calls GetCounterValue but uses other OS functionality too.
GetEvent	1	
GetISRID	1	
GetResource	1	
GetScheduleTableStatus	3	
GetSpinlock	1	
GetTaskID	1	
GetTaskState	1	
IncrementCounter	2	
NextScheduleTable	2	
OSErrorGetServiceId	1	
OSError_<svc>_<param>	1	
ReleaseResource	1	
ReleaseSpinlock	1	
ResumeAllInterrupts	1	
ResumeOSInterrupts	1	
Schedule	1	
SetAbsAlarm	2	
SetEvent	1	
SetRelAlarm	2	
SetScheduleTableAsync	2	
ShutdownAllCores	1	
ShutdownOS	1	
StartCore	1	
StartNonAutosarCore		Not implemented.
StartOS	1/2	Do not call more than once. This API handles both the microkernel and QM-OS start.
StartScheduleTableAbs	2	
StartScheduleTableRel	2	

Service name	Integrity category	Remarks
StartScheduleTableSynchron	2	
StopScheduleTable	2	
SuspendAllInterrupts	1	
SuspendOSInterrupts	1	
SyncScheduleTable	2	
TerminateApplication	1/2	The microkernel terminates the tasks and ISRs. Termination of alarms etc. is delegated to the QM-OS.
TerminateTask	1	
TryToGetSpinlock	1	
WaitEvent	1	
MK_AcquireLock	1	
MK_ReleaseLock	1	
MK_WaitGetClearEvent	1	
MK_GetClearEvent	1	
GetCpuLoad		Not implemented; not part of the AUTOSAR specification
GetStackInfo		Not implemented; not part of the AUTOSAR specification
InitCpuLoad		Not implemented; not part of the AUTOSAR specification
SimTimerAdvance		Not implemented; not part of the AUTOSAR specification
MK_ConditionalGetResource	1	
MK_ReadTime	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers”
MK_DiffTime	1	
MK_ElapsedTime	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers”
MK_DiffTime32	1	
MK_ElapsedTime32	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers”

Service name	Integrity category	Remarks
MK_MulDiv	1	You must ensure that the multiplication and division factors are in the correct range (1..65535) because no range check is performed.
MK_ElapsedMicroseconds	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers” . See also Section B.1, “Safety of short-duration time services” .
MK_ElapsedTime1u	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers” . See also Section B.1, “Safety of short-duration time services” .
MK_ElapsedTime10u	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers” . See also Section B.1, “Safety of short-duration time services” .
MK_ElapsedTime100u	1	The integrity of the time obtained depends on the hardware. See Section 7.10, “Hardware timers” . See also Section B.1, “Safety of short-duration time services” .
MK_ErrorGetParameter	1	
MK_ErrorGetServiceId	1	
MK_IsScheduleNecessary	1	Do not call from the idle-thread.
MK_ReportError	1	
MK_ScheduleIfNecessary	1	
MK_AsyncActivateTask	1	
MK_AsyncSetEvent	1	
MK_AsyncCancelAlarm	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncIncrementCounter	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncNextScheduleTable	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncSetAbsAlarm	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .

Service name	Integrity category	Remarks
MK_AsyncSetRelAlarm	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncSetScheduleTableAsync	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncStartScheduleTableAbs	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncStartScheduleTableRel	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncStartScheduleTableSyn- chron	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncStopScheduleTable	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_AsyncSyncScheduleTable	2	Request might get dropped; see Section B.2, “Safety of asynchronous QM-OS services” .
MK_NsToTicks_<freq>	1	
MK_NsToTicksF_<freq>	1	
MK_TicksToNs_<freq>	1	
MK_TicksToNsF_<freq>	1	
MK_TimestampNsToTicks	1	Provided that the software integrator implemented the service correctly and verified the implementation according to Section 6.5, “Verification criteria for Mk_board.h” .
MK_TimestampNsToTicksF	1	Provided that the software integrator implemented the service correctly and verified the implementation according to Section 6.5, “Verification criteria for Mk_board.h” .
MK_TimestampTicksToNs	1	Provided that the software integrator implemented the service correctly and verified the implementation according to Section 6.5, “Verification criteria for Mk_board.h” .
MK_TimestampTicksToNsF	1	Provided that the software integrator implemented the service correctly and verified the implementation according to Section 6.5, “Verification criteria for Mk_board.h” .

Table B.1. Safety status of OS services in the microkernel

B.1. Safety of short-duration time services

The short-duration time services `MK_ElapsedMicroseconds`, `MK_ElapsedTime1u`, `MK_ElapsedTime10u` and `MK_ElapsedTime100u` perform their calculations internally and externally using 32-bit unsigned arithmetic. This means that an overflow in the calculation cannot be detected. The overflow value for `MK_ElapsedTime1u` is 2^{32} microseconds, which is just over one hour. For `MK_ElapsedTime10u` and `MK_ElapsedTime100u` the times are correspondingly longer (almost 12 hours and just below 5 days respectively). The overflow value for `MK_ElapsedMicroseconds` depends on the `factor` parameter. A real elapsed time that is greater than the overflow value is truncated. The software integrator must ensure by other means that this cannot happen.

Variables used as the “previous time” location for calls to the short duration services must not be re-used with a different scaling unless they are re-initialized first. To initialize a “previous time” variable to a particular scaling, call the corresponding service and discard the “elapsed time” result.

B.2. Safety of asynchronous QM-OS services

It is possible for the application to call asynchronous QM-OS services on another core more frequently than the target core can handle them. If this happens, the target core has no means to inform the caller of the error, and activating `ErrorHook()` would only compound the problem.

An overload of this nature is handled on the target core by not performing the request and instead calling the function `MK_ReportDroppedXcoreRequest()`. This function increments a counter called `nDroppedJobs`, which is part of the microkernel core control variables. Apart from the lack of service, this is the only indication that asynchronous QM-OS requests have been dropped.

Appendix C. Safety of OS callout functions

Callout function	Calling method	Remarks
ErrorHook	Thread	
ProtectionHook	Thread	
StartupHook	Direct call by QM-OS	
ShutdownHook	Thread	
MK_InterruptLockingHook	Direct call	
ErrorHook_<App>	n.a.	Not implemented; application-specific: one function per OS-application
StartupHook_<App>	n.a.	Not implemented; application-specific: one function per OS-application
ShutdownHook_<App>	n.a.	Not implemented; application-specific: one function per OS-application
PreTaskHook	n.a.	Not implemented
PostTaskHook	n.a.	Not implemented
PreISRHook	n.a.	Not implemented; EB vendor-specific; intended only as a development aid
PostISRHook	n.a.	Not implemented; EB vendor-specific; intended only as a development aid
Alarm Callback	Direct call by QM-OS	See Section 4.6.4.3, “Alarm callback functions” .
MK_ProtectRamFromExternal	Direct call	Hardware-specific; refer to Section 7.2.1, “MK_ProtectRamFromExternal”
MK_InitHardwareBeforeData	Direct call	Memory protection is not in effect. See also Section 4.6.5, “Starting the microkernel” .
MK_InitHardwareAfterData	Direct call	Memory protection is not in effect. See also Section 4.6.5, “Starting the microkernel” .
MK_StartupPanic	Direct call	Memory protection is not in effect until <code>MK_HwEnableMemoryProtection()</code> in <code>MK_InitMemoryProtection()</code> has re-

Callout function	Calling method	Remarks
		turned. Afterwards, the memory protection settings of the microkernel are in effect.

Table C.1. Safety status of AUTOSAR-OS callout functions

C.1. Implementation notes

The `ProtectionHook()` is called whenever a protection violation occurs. The return value of `MK_PRO_CONTINUE` returns to the thread that contains the protection violation. Unless the reason for the protection violation is addressed prior to returning to the causing thread, the protection violation is triggered directly again. Consequently, the `ProtectionHook()` is invoked again, which may result in an undesired loop.

If the `ProtectionHook()` thread itself provokes a violation fault, the microkernel shuts the system down as a consequence.

The `ShutdownHook()` is invoked whenever the microkernel shuts down the system. Take care to not provoke protection violations within the shutdown hook. If a protection violation occurs and the `ProtectionHook()` returns `PRO_SHUTDOWN`, which is also the default action, the microkernel must shut down. The `ShutdownHook()` is restarted in this situation. Consequently, a possible loop that invokes the `ShutdownHook()` and the `ProtectionHook()` alternately may occur. It is recommended to guard against such a loop. One possible way is to check if the `ShutdownHook()` is invoked more than once.

Appendix D. 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/branches/MicroOs2.0_TRICORE_inspected_2020cw48/doc/public/safety_manual.

Filename	Revision
../../../../project/fragments/cross_references/Cross_References.xml	41812
../../../../project/fragments/glossary/Glossary.xml	39071
../fragments/entities/SafetyOS.ent.m4	36173
../fragments/top_level_requirements/Assumptions.xml	38199
../fragments/top_level_requirements/Definitions.xml	39071
../fragments/top_level_requirements/Level1.xml	9970
../fragments/top_level_requirements/Level2.xml	37392
../fragments/top_level_requirements/Overview.xml	9938
../fragments/top_level_requirements/SEooC.xml	29657
8.2_Safety_OS_safety_manual.xml	24546
ApiSafety.xml	41488
Assumed_Requirements.xml	18403
CalloutSafety.xml	27938
ConfigCriteria_TRICORE.xml	41562
ConfigCriteria.xml	41488
Derivatives_TRICORE.xml	41467
Description.xml	41488
DocumentInformation.xml	37392
History.xml	41812
images/FreedomFromInterference.svg	24085
Recommendations.xml	22872
ReservedWords.xml	24067
SafetyMan_InterruptLevelsAscending.ent	36650
SafetyMan_TickType32bit.ent	38853
SafetyMan_TRICORE.ent	41488

Filename	Revision
SafeUse.xml	41488
Supplement_TRICORE.xml	41550

Glossary

bus master	A <i>bus master</i> is a hardware unit that is capable of reading from or writing to the memory or peripheral hardware in a microcontroller.
callout function	A callout function is a function that the microkernel calls, but you must implement it. You must ensure that functional safety is guaranteed by its implementation.
core	A core is a single processing unit of a microcontroller, containing registers, arithmetic/logic unit and other processing hardware. A multi-core microcontroller has two or more cores that can execute software simultaneously.
error-correction code	Error correction codes (ECC) are redundant information stored along with the data they protect. The combination of both enables hardware to detect and eventually correct transient errors.
emergency stop state	The emergency stop state consists of an endless loop with disabled interrupts. The microkernel enters the emergency stop state if an inconsistent internal state is detected. Also if the protection hook causes a further protection fault, and the subsequent attempt to shut down in an orderly manner also fails.
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.
inter-core	On a multi- core microcontroller, an inter-core operation is an operation that involves two (or more) of the cores. For example: inter-core communication, inter-core mutual exclusion.
internal interrupt	<i>Internal</i> interrupts are interrupts that are handled by the microkernel itself. They are used for microkernel internal purposes. Inter-core interrupts are an example of internal interrupts.
interrupt	The interrupt request signals the CPU to stop the program execution at the current point and to continue the execution at the address which is associated with the specific interrupt request.
interrupt service routine	An interrupt service routine (ISR) is an OS-object . An ISR is the code that is executed to handle an interrupt request. ISRs can be global or belong to an OS-Application .

lockstep	Lockstep means that hardware components are replicated and perform the identical function for each clock cycle. The output of the hardware components is compared. If the output diverges, the error is detected and appropriate hardware-specific actions are performed.
memory protection unit	A memory protection unit (MPU) is a programmable hardware mechanism that restricts accesses to memory or memory-mapped hardware unless the access is explicitly permitted by the programming.
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 generator	The OS generator creates C-source code from the OS configuration in EB tresos Studio.
OS-object	An OS-object is a data structure managed by the OS. This includes the runnable code of tasks , ISRs , 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 .
processor status word	The processor status word (PSW) contains the flags of the CPU. Flags are typically a combination of results from arithmetic/logic operations (e.g. carry, overflow), CPU state (e.g. interrupts enable) and CPU mode (e.g. non-privileged or privileged mode).
QM-OS	EB tresos Safety OS is divided into two parts. The QM-OS is the part without ASIL allocation according to ISO26262.
restricted function	A function which does not use initialized data nor assumes cleared variables.
safe state	A safe state is an operating mode of an item without an unreasonable level of risk.
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.

Bibliography

- [ASCOS_USER-GUIDE]** Elektrobit Automotive GmbH: *EB tresos AutoCore OS documentation*
- [AURIX_SM15_1]** Infineon: *AURIX Safety Manual - AP32224*, V1.5.1, March 2018
- [ISO/IEC 61508:2010]** *Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010
- [ISO/IEC 61508-3:2010]** *Functional safety of electrical/electronic/programmable electronic safety-related systems: Part 3: Software requirements*, 2010
- [ISO24765]** *ISO/IEC/IEEE 24765 First edition: Systems and software engineering - Vocabulary*, 2010
- [ISO26262_1ST]** *INTERNATIONAL STANDARD ISO 26262: Road vehicles - Functional safety*, 2011
- [ISO26262-1_1ST]** *INTERNATIONAL STANDARD ISO 26262-1: Road vehicles - Functional safety - Part 1: Vocabulary*, 2011
- [ISO26262-10_1ST]** *INTERNATIONAL STANDARD ISO 26262-10: Road vehicles - Functional safety - Part 10: Guideline on ISO 26262*, 2012
- [ISO26262-3_1ST]** *INTERNATIONAL STANDARD ISO 26262-3: Road vehicles - Functional safety - Part 3: Concept phase*, 2011
- [ISO26262-6_1ST]** *INTERNATIONAL STANDARD ISO 26262-6: Road vehicles - Functional safety - Part 6: Product development at the software level*, 2011

- [KNOWN-PROBLEMS]** *EB tresos AutoCore known problems, EB_tresos_Autocore_known_issues-SAFETY-OS_<version>_COMMON_SRC.pdf*
- [OSEKOS142BIND]** *OSEK/VDX:OSEK Binding Specification, Version 1.4.2*
<http://portal.osek-vdx.org/files/pdf/specs/binding142.pdf>
- [Q-STATEMENT]** *EB tresos AutoCore Quality Statement Safety OS, EB_tresos_Autocore-quality_statement-SafetyOs_<version>_<derivative>.pdf*
- [RELNOTES]** *EB tresos Safety OS release notes*
- [SAFEOSUSRDOC]** *EB tresos Safety OS user's guide*
- [SM2INSP1]** *Inspection Report 1 of the Microkernel version 2 Safety Manual*
- [SM2INSP10]** *Inspection Report 10 of the Microkernel version 2 Safety Manual*
- [SM2INSP11]** *Inspection Report 11 of the Microkernel version 2 Safety Manual*
- [SM2INSP12]** *Inspection Report 12 of the Microkernel version 2 Safety Manual*
- [SM2INSP13]** *Inspection Report 13 of the Microkernel version 2 Safety Manual*
- [SM2INSP14]** *Inspection Report 14 of the Microkernel version 2 Safety Manual*
- [SM2INSP15]** *Inspection Report 15 of the Microkernel version 2 Safety Manual*
- [SM2INSP16]** *Inspection Report 16 of the Microkernel version 2 Safety Manual*

- [SM2INSP17]** *Inspection Report 17 of the Microkernel version 2 Safety Manual*

- [SM2INSP19]** *Inspection Report 19 of the Microkernel version 2 Safety Manual*

- [SM2INSP2]** *Inspection Report 2 of the Microkernel version 2 Safety Manual*

- [SM2INSP20]** *Inspection Report 20 of the Microkernel version 2 Safety Manual*

- [SM2INSP21]** *Inspection Report 21 of the Microkernel version 2 Safety Manual*

- [SM2INSP22]** *Inspection Report 22 of the Microkernel version 2 Safety Manual*

- [SM2INSP3]** *Inspection Report 3 of the Microkernel version 2 Safety Manual*

- [SM2INSP4]** *Inspection Report 4 of the Microkernel version 2 Safety Manual*

- [SM2INSP5]** *Inspection Report 5 of the Microkernel version 2 Safety Manual*

- [SM2INSP6]** *Inspection Report 6 of the Microkernel version 2 Safety Manual*

- [SM2INSP7]** *Inspection Report 7 of the Microkernel version 2 Safety Manual*

- [SM2INSP8]** *Inspection Report 8 of the Microkernel version 2 Safety Manual*

- [SM2INSP9]** *Inspection Report 9 of the Microkernel version 2 Safety Manual*

- [TC_ARCH_161_-VOL1]** Infineon: *TriCore TC1.6P & TC1.6E User Manual (Volume 1)*, V1.0 D10, February 2012
- [TC_ARCH_161_-VOL2]** Infineon: *TriCore TC1.6P & TC1.6E User Manual (Volume 2)*, V1.0 D15, July 2013
- [TC_ARCH_162_-VOL1]** Infineon: *TriCore TC 1.6.2 core architecture manual (Volume 1)*, V1.0, January 2017
- [TC_ARCH_162_-VOL1_11]** Infineon: *TriCore TC 1.6.2 core architecture manual (Volume 1)*, V1.1, August 2017
- [TC23X_DS]** Infineon: *TC233 / TC234 / TC237 A-Step Target Data Sheet*, V1.1, June 2015
- [TC23X_UM]** Infineon: *AURIX TC21x/TC22x/TC23x Family User's Manual*, V1.1, December 2014
- [TC27X_UM_B]** Infineon: *AURIX TC27x B-Step User's Manual*, V1.4.1, February 2014
- [TC27X_UM_C]** Infineon: *AURIX TC27x C-Step User's Manual*, V2.2, December 2014
- [TC29X_UM_B]** Infineon: *AURIX TC29x B-Step User's Manual*, V1.3, December 2014
- [TC33X_DS_06]** Infineon: *TC33x Target Data Sheet / Target Specification*, V0.6, June 2019
- [TC33X_TC32X_-APPX_UM_14]** Infineon: *AURIX TC33x/TC32x User's Manual Appendix*, V1.4, December 2019
- [TC33X_TC32X_-DS_AD_13]** Infineon: *TC33x/TC32x Data Sheet Addendum*, V1.3, March 2020
- [TC36X_DS_10_-AA]** Infineon: *TC36x AA-Step Target Data Sheet / Target Specification*, V1.0, April 2020

- [TC36X_DS_AD_-
13] Infineon: *TC36x Data Sheet Addendum*, V1.3, January 2020
- [TC36X_UM_AP-
PX_15] Infineon: *AURIX TC36x User's Manual Appendix*, V1.5, April 2020
- [TC37X_DS_10_-
AA] Infineon: *TC37x Target Data Sheet / Target Specification*, V1.0, January 2020
- [TC37X_DS_AD_-
13] Infineon: *TC37x Data Sheet Addendum*, V1.3, January 2020
- [TC37X_TS_V251] Infineon: *AURIX TC37x Target Specification*, V2.5.1, April 2018
- [TC37XED_TS_-
V251] Infineon: *AURIX TC37xED Target Specification*, V2.5.1, April 2018
- [TC38X_DS_10] Infineon: *TC38x Target Data Sheet / Target Specification*, V1.0, October 2018
- [TC38X_TS_V230] Infineon: *AURIX TC38x Target Specification*, V2.3.0, September 2017
- [TC38X_TS_V251] Infineon: *AURIX TC38x Target Specification*, V2.5.1, April 2018
- [TC39X_DS_06] Infineon: *TC39x Target Data Sheet / Target Specification*, V0.6, April 2016
- [TC39X_TS_V251] Infineon: *AURIX TC39x-B Target Specification*, V2.5.1, April 2018
- [TC39XB_DS_11] Infineon: *TC39xB Target Data Sheet / Target Specification*, V1.1, September 2019
- [TC3XX_SM106] Infineon: *AURIX TC3xx Safety Manual*, V1.06, E11 2019
- [TC3XX_SM109] Infineon: *AURIX TC3xx Safety Manual*, V1.09, E04 2020

[TC3XX_TS_1_201] Infineon: *AURIX TC3xx Target Specification*, V2.0.1, July 2016

[TC3XX_TS_1_210] Infineon: *AURIX TC3xx Target Specification*, V2.1.0, February 2017

[TC3XX_UM_1_14] Infineon: *AURIX TC3xx User's Manual (part 1)*, V1.4.0, December 2019

[TC3XX_UM_1_16] Infineon: *AURIX TC3xx User's Manual (part 1)*, V1.6.0, August 2020

[TC3XX_UM_2_14] Infineon: *AURIX TC3xx User's Manual (part 2)*, V1.4.0, December 2019

[TC3XX_UM_2_16] Infineon: *AURIX TC3xx User's Manual (part 2)*, V1.6.0, August 2020