MICROSAR

Safety Guide

Version 1.2.2

Authors	Jonas Wolf
Status	Released

Document Information

History

Author	Date	Version	Remarks
Jonas Wolf	2014-01-08	0.1.0	Initial structure
Jonas Wolf	2014-01-27	0.2.0	Ready for review
Jonas Wolf	2014-01-31	0.2.1	First comments included
Jonas Wolf	2014-02-06	0.2.2	Review comments from visml included
Jonas Wolf	2014-02-07	0.2.3	Added additional AUTOSAR components
Jonas Wolf	2014-02-17	0.2.4	Rework after review sessions
Jonas Wolf	2014-03-10	0.3.0	Rework after review with visrn
Jonas Wolf	2014-03-28	1.0.0	Final rework and release
Jonas Wolf	2014-05-27	1.1.0	Improvements from workshop with customer
Jonas Wolf	2014-06-03	1.2.0	Rework after review with visml
Jonas Wolf	2014-10-14	1.2.1	Clarifications on section 2.3
Jonas Wolf	2014-10-20	1.2.2	More information on securing non-volatile data

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_TR_SafetyConceptStatusReport.pdf	V1.2.0
[2]	AUTOSAR	AUTOSAR_SWS_CRCLibrary.pdf	V4.4.0
[3]	AUTOSAR	AUTOSAR_SWS_ECUStateManager.pdf	V4.1.0
[4]	AUTOSAR	AUTOSAR_SWS_E2ELibrary.pdf	V3.1.0
[5]	Vector	MICROSAR RTE Safety Guide SafetyGuide_Rte.pdf	V4.2.0
[6]	Boeing	Single Event Upset at Ground Level Eugene Normand, Boeing Defense & Space Group, Seattle, WA 98124-2499	2004
[7]	ISO	ISO 26262-4:2011 Road vehicles — Functional safety — Product development at the system level	2011
[8]	ISO	ISO 26262-5:2011 Road vehicles — Functional safety — Product development at the hardware level	2011
[9]	ISO	ISO 26262-6:2011 Road vehicles — Functional safety — Product development at the software level	2011
[10]	Koopman	32-Bit Cyclic Redundancy Codes for Internet Applications Philip Koopman, ECE Department & ICES, Carnegie Mellon University, Pittsburgh, PA, USA	2002
[11]	IEEE	IEEE Std 802.3™-2012	2012
[12]	TTTech	Safe Watchdog Manager Safety Manual D-SAFEX-S-70-001	2.3.9
[13]	TTTech	Safe Watchdog Interface Safety Manual D-SAFEX-S-70-005	1.8.3
[14]	TTTech	E2E Protection Wrapper Safety Manual D-MSP-M-70-002	1.2.9

Contents

Illustrations

Tables

1 Introduction

1.1 Purpose

This document provides guidance for designing and implementing software of safety-related systems using Vector's AUTOSAR implementation.

This document does not impose any requirements on your development process when using Vector products. All necessary requirements are given in the respective Safety Manuals.

Its intended audience is system and software engineers and deciders who consider the use of AUTOSAR for their safety-related development.

1.2 Scope

This document focusses on the architecture, design and implementation of software using AUTOSAR V4.1R2 in safety-related systems in the automotive industry.

Only the Vector AUTOSAR implementation MICROSAR is considered.

The design of software without AUTOSAR is not considered.

Knowledge and understanding of functional safety, the ISO 26262 and AUTOSAR is required to understand the contents of this document.

1.3 Definitions

No.	Term	Description
1.51	functional safety	absence of unreasonable risk (1.136) due to hazards (1.57) caused by malfunctioning behavior (1.73) of E/E systems (1.31)
1.73	malfunctioning behavior	failure (1.39) or unintended behavior of an item (1.69) with respect to its design intent
1.39	failure	termination of the ability of an element (1.32), to perform a function as required Note: Incorrect specification is a source of failure
1.57	hazard	potential source of harm (1.56) caused by malfunctioning behavior (1.73) of the item (1.69)
1.42	fault	abnormal condition that can cause an element (1.32) or an item (1.69) to fail
1.102	safe state	operating mode (1.81) of an item (1.69) without an unreasonable level of risk (1.99)
1.92	random hardware failure	failure (1.39) that can occur unpredictably during the lifetime of a hardware element (1.32) and that follows a probability distribution
1.49	freedom from interference	absence of cascading failures (1.13) between two or more elements (1.32) that could lead to the violation of a safety requirement
1.44	fault reaction time	time-span from the detection of a fault (1.42) to reaching the safe state (1.102)

Table 1-1 Definitions from ISO 26262

1.4 Overview

In section 2 considerations and assumptions for the system level are summarized.

In section 3 recommendations and best-practices for safety mechanisms are given based on the definitions and assumptions of section 2.

In section 4 example use-cases for software architectures using AUTOSAR for different kinds of systems are given.

In section 5 additional recommendations for the use of Vector's AUTOSAR stack are given.

In section 6 procedural requirements for implementing functional safety using Vector's safety solution are stated.

In section 7 some of the assumptions made by different components of Vector's safety solution are resolved.

2 Functional safety on system level

Some functions provided by E/E systems in cars bear an inherent risk of harm to the driver, passengers and pedestrians. Examples for such systems are an active front steering or breaking system.

To have a low level of residual risk for harm to persons and damage to cars, the ISO 26262 was defined to address the development of such safety-related E/E systems.

The ISO 26262 defines *functional safety* as the *absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems*.

Malfunctioning behavior is assumed when a *failure or unintended behavior of an item with respect to its design intent* occurs.

Thus, a functionally safe system has to mitigate the risk associated with a failure and prevent unintended behavior.

2.1 Prevention of unintended behavior

Prevention of unintended behavior requires a sufficiently detailed and precise specification to identify unintended behavior as such. A sufficiently detailed and precise specification can be assured by reviews required by the development process.

Additionally, unintended behavior can accidentally be implemented into the system by e.g. programming faults. All faults designed or implemented into the system are systematic faults. Systematic faults are prevented by the development process through, e.g. coding standards, reviews and static analysis tools. As these verification methods are limited, testing is also applied.

Testing can only show the presence of faults, but never their absence. Thus, a certain test depth is required by the ISO 26262. For example a software unit may be required to be tested based on the requirements so that branch coverage is achieved. The assumption is that if the complete system functionality is tested and no faults occurred, the probability that the complete system has residual faults is very low.

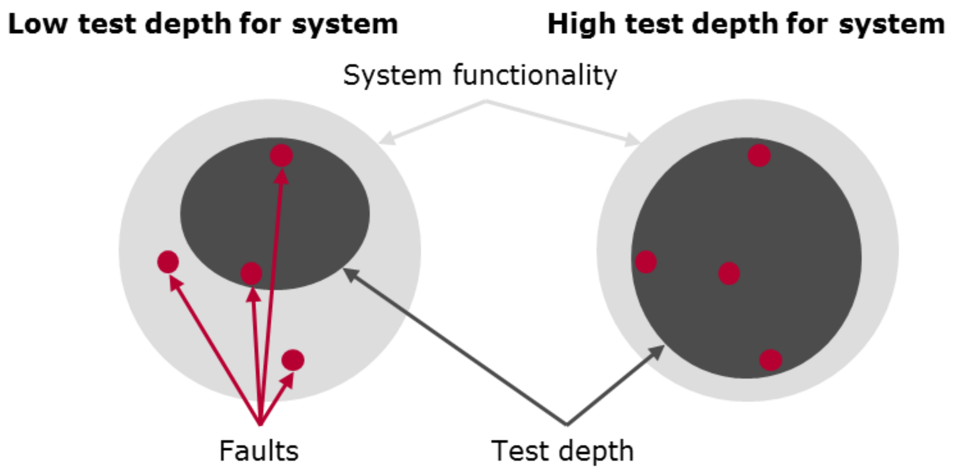

Figure 2-1 Relation between test depth, system functionality and residual faults

To facilitate testing the system needs to provide interfaces for stimulation and tracing of outputs and internal states. To achieve a very high test depth, the system needs to have a reasonable complexity.

On system level, complexity is most dominantly influenced by:

> the number of distinct functions of the system,

> the number of states and transitions of the system, and

> the number of elements performing computations in parallel.

Thus, the system should be as simple as possible to keep it testable.

Additionally, non-deterministic behavior also increases complexity of the system verification.

Thus, the system should be as deterministic as possible to ease verification.

The combination of development process measures and testing prevents unintended behavior of the system.

2.2 Mitigation of risk associated with a failure

The risk associated with a failure can be mitigated by reducing the probability of occurrence of the failure or the severity of the failure. The *failure* of the system is a consequence of a *fault*. *Faults* considered in this section are *random hardware faults*.

Additional *technical safety requirements* have to be introduced to mitigate the severity of a *fault*. The system needs to be able to **detect** *random hardware faults* and then to **act** on them to achieve a *safe state*.

Random hardware faults can result either from *permanent faults* or *transient faults*. There are different methods to detect these two types of faults. Depending on the ASIL, there are different requirements on the probabilities of *random hardware faults* (e.g. for the probabilistic metric for random hardware faults PMHF).

The probabilities, i.e. failure rate, for *permanent faults* can be estimated using standards, e.g. IEC TR 62380, or is provided by the manufacturer of the *hardware components*. The probabilities of some *transient faults*, especially Single Event Upsets (SEUs) cannot be estimated that easily. Several scientific papers (e.g. [6]) suggest that the probability of a transient fault can be in the order of magnitude of $10^{-8 \text{ to } -7}$ 1/h. We assume that the Single Event Upset may directly lead to a violation of a safety goal and is, thus, a single-point fault. The probability of a single-point fault has to be less than 10^{-7} 1/h for ASIL B or 10^{-8} 1/h for ASIL C respectively.

Thus, we recommend the consideration of transient faults starting from ASIL B. For ASIL C systems transient faults must be handled appropriately.

There are different methods to detect a *fault* in a *hardware component*. Among them are:

> Use of error detection codes, e.g. a CRC

> Comparison of redundant data

> Use of a time limit supervisor, e.g. a watchdog

There are also different methods for a system to act upon a fault. Among them are:

> Reset of the component where the fault occurred
> Depowering of the component where the fault occurred
> Exclusion of inputs that are known to be incorrect

Best practices for some of these mechanisms in relation to AUTOSAR are presented in section 3.

2.3 Fail-safe and fail-operational systems

Most systems in the automotive today have a fail-safe behavior, i.e. in case of a *random hardware fault* the system tries to achieve a *safe state*. This is the implicitly assumed mechanism by ISO 26262 to mitigate *random hardware faults*.

Mitigation of faults for a system, thus, only means to detect the fault and enter a safe state.

For example, in case of an active front steering system, the actuator of the system is disabled (= *safe state*) when a *failure* is detected.

Thus, only fail-safe systems are considered in this guide.

There are other systems where the correct operation is required even in the presence of one or more *faults*. These systems are called *fail-operational*.

For example, a steer-by-wire system may be required to operate correctly even if, e.g. the microcontroller is broken.

Since support by the ISO 26262 of fail-operational systems through definitions of terms, methods and requirements is poor, we only consider fail-safe systems in this guide.

3 Recommendations on safety mechanisms

A logical prerequisite for a functionally safe system is the integrity of all comprising elements.

For this document we define integrity as intended behavior with respect to its design intent.

3.1 Integrity of the microcontroller

Since this document focuses on software, the executing microcontroller needs to be integer.

Usually, there are the following techniques to achieve integrity of a microcontroller:

- Operation in lock-step mode
- Self-tests of the microcontroller components (incl. plausibility tests)
- Monitoring the temperature of the microcontroller

The following sections provide details about these techniques.

3.1.1 Operation in lock-step mode

Microcontrollers with lock-step mode assure the integrity of calculations and register contents with respect to transient random hardware failures (e.g. bit flips). Details on what exactly is protected depend on the selected hardware architecture.

Enabling lock-step depends on the selected hardware architecture. We recommend **enabling lock-step as early as possible in the boot process** (e.g. in start-up code).

An additional self-test of the lock-step mechanism has to be implemented, depending on the required *diagnostic coverage* and the selected hardware architecture.

A fault in the lock-step mechanism is a *latent fault*.

3.1.2 Monitoring the temperature of the microcontrollers

High temperature strongly increases the probability of transient random hardware faults. Thus, keeping the microcontroller within its operating limits is necessary for its integrity.

Some microcontrollers support measuring and monitoring the die temperature.

Monitoring the die temperature is a measure to increase the robustness of the system. Usually, the monitoring can be implemented in analogy to self-tests.

3.1.3 Self-test of the microcontroller components

Additional self-tests of individual microcontroller components may have to be implemented, depending on the required diagnostic coverage of the application. Self-tests can be used to increase diagnostic coverage for single-point faults (integrity at start-up) and latent faults (integrity in continuous operation).

Thus, the requirements on self-tests are defined by the required ASIL.

Especially, the self-test of the used watchdog is often assumed to be mandatory by the microcontroller manufacturer.

Other examples for such a self-test are the Logic built-in self-test (LBIST) of the TriCore architecture.

The safety manual of the respective microcontroller has to be applied.

3.2 Integrity of volatile data

Since this document focuses on software, also the used volatile memory (e.g. SRAM or DRAM) needs to be integer.

Usually, there are the following techniques to achieve integrity of volatile memory:

> Tests of volatile memory

> Protection of volatile memory through error correcting codes (ECC)

The following sections provide details about these techniques.

3.2.1 Static tests of volatile memory

Tests of volatile memory can detect several failure modes of memory cells and addressing logic.

Tests of volatile memory need to be implemented depending on the failure rate of the selected memory. The failure rate is most dominated by the selected technology (e.g. SRAM, DRAM) and the manufacturing process quality.

Tests of volatile memory can be used to increase the diagnostic coverage of the system. Tests of volatile memory may be required at start-up and periodically.

A fault in the implementation of the test of volatile memory is a *latent fault*.

3.2.1.1 Initial test of volatile memory

An initial test of volatile memory can be implemented in the start-up code before the memory is used initially.

Testing memory initially at start-up assures integrity of the data and code that is loaded into the memory. Additionally, the data cache can be disabled at start-up to easily allow testing of the volatile memory.

Some microcontrollers provide built-in self-tests to check the volatile memory, e.g. Infineon TC27x.

If the test of volatile memory has to be implemented in software, we recommend using the March test algorithm. The March test algorithm provides the least time complex (O(n)) behavior and detects the most failure modes.

Caution
Consider disabling the data cache for tests of volatile memory. The data cache may render the implemented test ineffective, because test patterns are not written to nor read from the memory chip.

A test of volatile memory at start-up has to be implemented by the user of the MICROSAR stack.

We assume that the flash bootloader has no influence on the application software and it does not prevent an intended application start. Thus, the flash bootloader is not considered here.

3.2.1.2 Periodic tests of volatile memory

Depending on how often the ECU is powered on and a self-test at start-up is performed and the failure rate of the memory chip, a periodic memory tests is not necessary.

The AUTOSAR RAM Test component could be used to test volatile memory periodically.

The periodic memory tests could also be implemented as Complex Driver.

The Vector AUTOSAR RAM Test component is currently not developed according to ISO 26262.

3.2.2 Protection of volatile memory through error correcting codes (ECC)

The volatile memory should be protected by error correcting codes (ECC), to protect the system against transient faults in volatile memory from radiation or strong electric fields.

ECC-mechanisms do not protect against most failure modes mentioned in the previous section, but they are the most efficient way (from a software perspective) to protect against Single Event Upsets (SEUs).

Enabling the use of ECC-mechanisms depends on the selected hardware architecture. We recommend **enabling the use of ECC-mechanisms as early as possible in the boot process** (e.g. in start-up code).

3.2.2.1 Redundant storage of data

If no ECC-mechanism is available for the selected hardware architecture, storing the data redundantly at two separate memory locations (and at best inverted), may be an alternative under some circumstances. Special care has to be taken that all safety-related data is stored redundantly.

The redundantly stored values have to be compared at every read. The write and read operations are at best wrapped by functions performing the inversion, redundant storage and comparison.

Thus, it is usually easier to use ECC-mechanisms, because they do not introduce new functions and are an effective and transparent protection.

Redundant storage of data does not cover transient faults in registers. A hardware architecture with lock-step is suggested to protect against these kinds of faults.

Redundant storage of data is not supported by the Vector MICROSAR stack.

3.3 Integrity of non-volatile data

Non-volatile data must be protected against random corruption to show the required *diagnostic coverage* for an ASIL.

Non-volatile data comprises interrupt vector table, executable code, and initialized variables. Non-volatile data stored using NVRAM Manager is addressed in section 3.3.2.

A hamming distance of four is required to claim a high *diagnostic coverage* with respect to random corruption (see [8], D.2.4.1).

We recommended protecting the complete flashed binary using a CRC to ensure that the binaries on host and target are identical.

We recommend using the CRC-32 (see [2]) to protect the complete flashed binary file. CRC-32 offers a sufficient hamming distance for quite large data sizes.

CRC-32 has a hamming distance of at least four for data sizes of up to 11450 byte (see [10]). If the complete flashed binary file is larger than 11450 bytes, then more than one CRC-32 has to be calculated to adequately protect that file.

Depending on the type of binary flashed (e.g. SREC, ELF, Raw-Binary) not the complete non-volatile memory has to be checked, but only distinct sections.

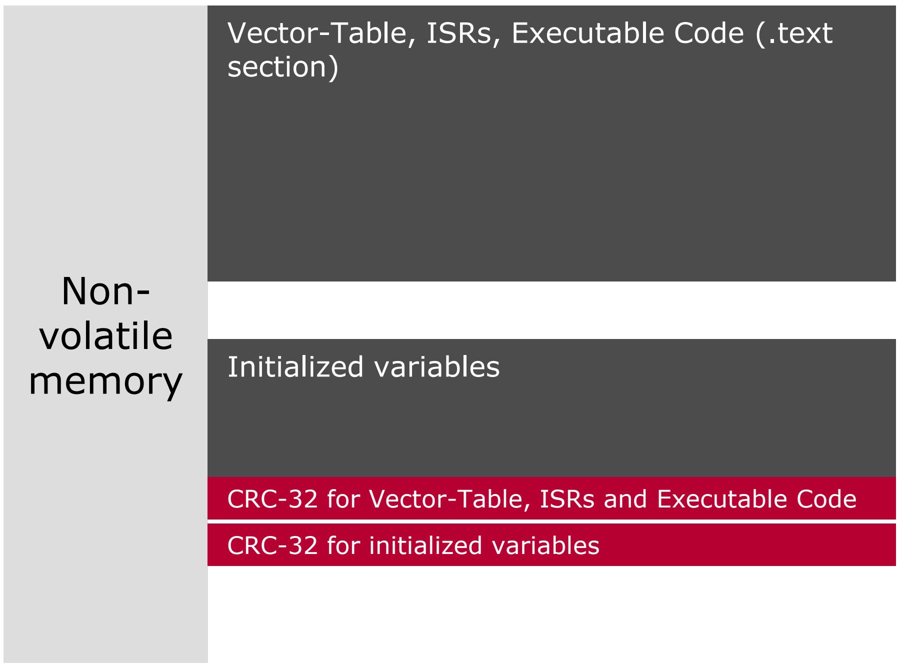
Non-volatile memory
Vector-Table, ISRs, Executable Code (.text section)
Initialized variables
CRC-32 for Vector-Table, ISRs and Executable Code
CRC-32 for initialized variables

Figure 3-1 Example for protecting non-volatile memory with two CRCs

3.3.1 Consistency of configuration and calibration data

Configuration and calibration data must be verified together with the configured and calibrated software (see [9], Appendix C).

One method to assure that only verified packages of configuration data, calibration data and software are used is to only distribute the verified packages as one flashable entity.

There may be other organizational means to assure the consistency of configuration data, calibration data and software. However, we recommend disabling post-build configuration. This in turn reduces complexity of the system.

3.3.2 Securing non-volatile data with use of NVRAM Manager

The NVRAM Manager component could be used to safely store and retrieve data from non-volatile memory. Data stored and retrieved using the NVRAM Manager is automatically secured using a CRC if configured.

The Vector NVRAM Manager component is currently not developed according to ISO 26262.

Thus, we recommend protecting the safety-related data, which would be stored using NVRAM Manager, with an additional CRC and unique ID that is generated and checked by your application. The CRC protects against corruption in non-volatile memory and by the NVRAM Manager. The unique ID protects against the swap of equally sized data by the NVRAM Manager. See also section 5.8.

The consistency between different data blocks can be established, if an additional write cycle counter is stored with each data block. The cycle counter can be saved on its own in the last block of a writing sequence to later identify if all blocks contain data from the same writing sequence. The data is consistent if all blocks contain the same cycle counter. This mechanism can be applied multiple times for different groups of blocks.

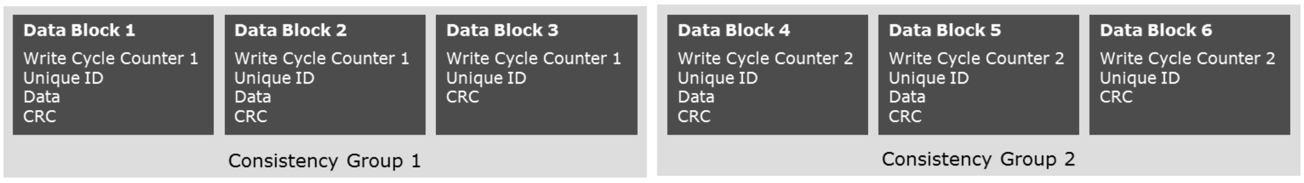

Figure 3-2 Establishing consistency of multiple data block groups

The NVRAM Manager can, however, never guarantee that always the last saved block of a dataset NVRAM block is retrieved. A reset may occur any time before critical data is written to the non-volatile memory. Thus, the system should have safe default values if no, corrupt or inconsistent data is detected in non-volatile memory.

3.4 Initialization of the microcontroller

The system should always be in the safe state during initialization.

Thus, all code for initialization prior to executing the safety-related functions (e.g. mechanisms) usually has to be developed according to the required ASIL. Faults during initialization may prevent safety mechanisms to be executed.

Faults during initialization may especially occur before calling `StartOS()` and before the watchdog is initialized.

Initialization of the microcontroller may also happen on an unexpected reset during normal operation.

Consider the time necessary for initialization of the microcontroller for your *fault reaction time*. For example, a system may need additional actions in software to achieve the safe state after a watchdog-initiated reset. This software is only executed after initialization of the microcontroller.

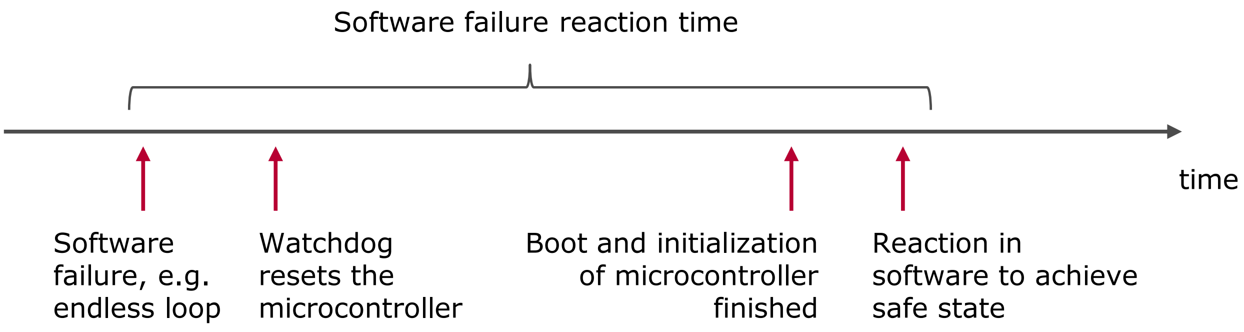
Software failure reaction time
time
Software failure, e.g. endless loop
Watchdog resets the microcontroller
Boot and initialization of microcontroller finished
Reaction in software to achieve safe state

Figure 3-3 Example for failure reaction time and initialization time (Reset state is not safe state)

3.5 Separation in memory

One measure to increase robustness of software is to separate different software components in memory. For mixed-ASIL systems or systems addressing multiple safety goals, separation in software is also necessary to show *freedom from interference*.

Separation in memory of different software components can be achieved by using a Scalability Class 3 (SC3) SafeContext operating system.

The software components with the same assigned ASIL and the same assigned safety goal can then be grouped into one application (see Figure 3-4).

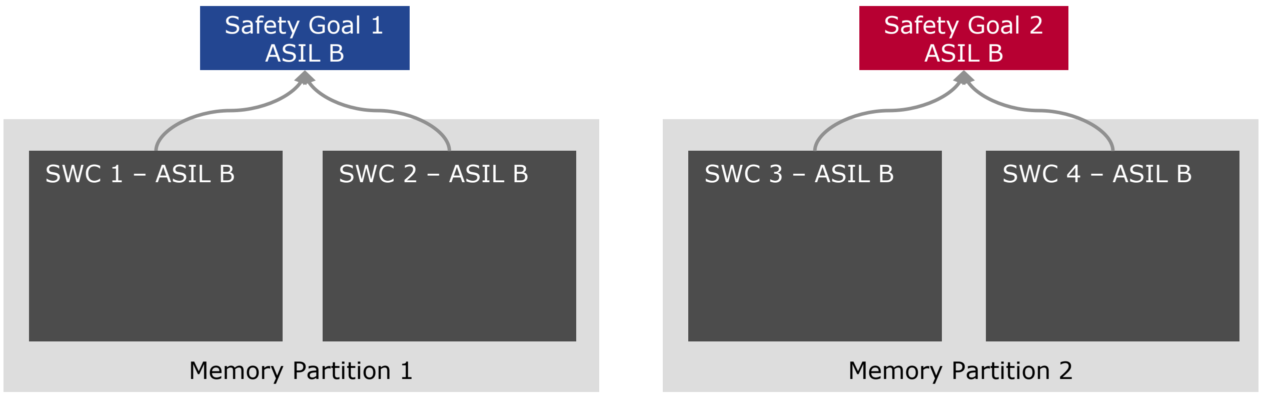

Figure 3-4 Memory partitioning for mixed-ASIL systems

The SC3 SafeContext operating system with the highest ASIL of the software components has to be used. For example, if the highest ASIL of all software components is ASIL D, the operating system has to have an ASIL D as well.

The SC3 SafeContext operating system has to run in supervisor mode and, thus, its malfunction may lead to a violation of a safety goal of any software component running on the same processor. In multi-core systems the software components of all cores need to be considered for the ASIL of the operating system.

Memory partitioning is the most effective way to ensure *freedom from interference* between the partitions. If accepted by the OEM, other means, like static analysis may be applied to prevent cascading failures.

3.6 Separation in time

The use of SafeWatchdog or an SC4 operating system may be adequate depending on the safety requirements.

The use of SafeWatchdog is always recommended. It may be supported by an SC4 operating system.

The use of SafeWatchdog is recommended, if the system is always in a safe state during reset and there is only one safety-related function implemented on the microcontroller. In case of a fault, the SafeWatchdog can reset the system to bring up the microcontroller again.

The use of an SC4 operating system is recommended, if either the system is not always in a safe state during reset or there is more than one safety-related function implemented on the microcontroller. In case of a software fault in one of the SWCs, the SC4 operating system can still schedule another SWC that transforms the system into a safe state or implements another safety-related function. In case of a hardware fault, there is no advantage with respect to the use of SafeWatchdog.

For further information refer to the Safety Manuals of the SafeWatchdog (see [12], [13] and Safety Manual for respective Watchdog Driver).

The Vector SC4 SafeContext operating system is currently developed according to ISO 26262, but the timing protection mechanisms are not considered a safety goal currently. Especially, the order and time of scheduled tasks is not guaranteed by any Vector OS at the moment.

3.7 Scheduling

Most safety-related functions require a hard real-time behavior of the software, since the safe state must always be reached in a certain period of time.

A deterministic behavior increases the achievable test depth with respect to timing.

We, thus, recommend implement high ASIL applications with schedule tables (see [9], Table 3, 1f) and with a restricted use of interrupts (see [9], Table 3, 1g).

Of course, a safe system can also be implemented using interrupts, cyclic tasks and priority based scheduling. The effort for verification of such systems is in general higher than for time-triggered systems.

Additionally, time-triggered bus systems allow easy (implicit) synchronization of more than one ECU.

We further recommend:

> Using one schedule table for initialization and shutdown

> Using one schedule table for cyclic tasks per mode

> Using only basic tasks to keep the system even more simple

AUTOSAR allows the parallel use of more than one schedule table. We definitely recommend just using one schedule table at a time.

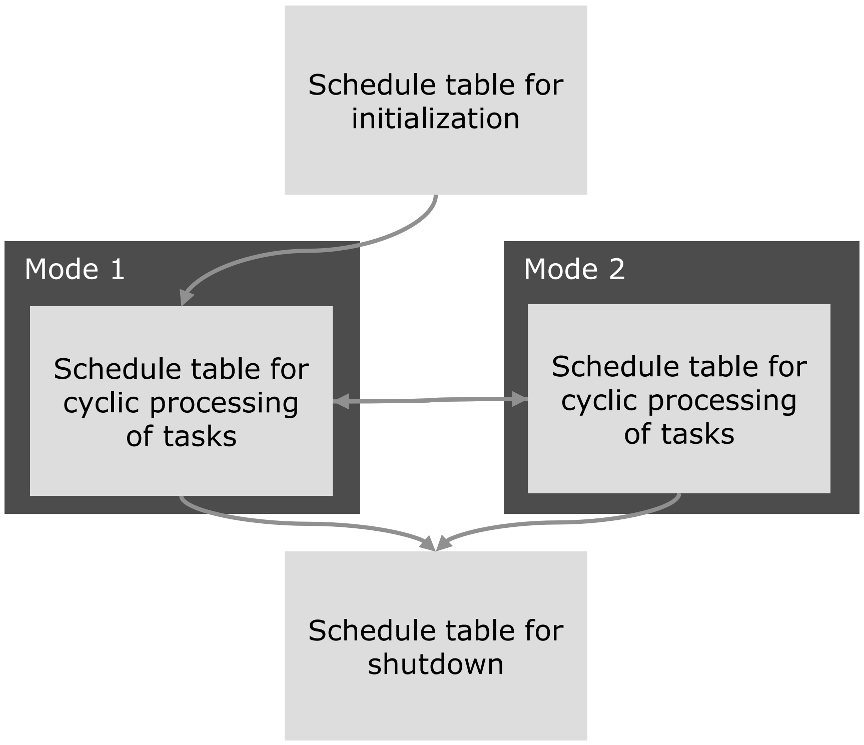
Schedule table for initialization
Mode 1
Schedule table for cyclic processing of tasks
Mode 2
Schedule table for cyclic processing of tasks
Schedule table for shutdown

Figure 3-5 Example for the use of schedule tables and different modes

3.8 Communication

As described in section 2.1, the system should be as deterministic as possible. A deterministic behavior is supported by avoiding queues in communication.

Thus, we recommend using only explicit sender/receiver communication with last-is-best behavior.

Using explicit sender/receiver communication brings two additional advantages. First, unnecessary copy operations are avoided. Second, the integrator has full control over the communication (when time-triggered scheduling is introduced in parallel).

We recommend using the end-to-end (E2E) protection mechanisms of AUTOSAR to protect communication between different ECUs.

Please refer to the Safety Guide of the RTE (see [5]) on how to protect intra-ECU communication.

Currently Vector does not provide an IOC mechanism developed according to ISO 26262.

For further information refer to the Safety Guide of the RTE (see [5]).

3.9 Input and output

The current Vector MICROSAR stack is developed according to a standard development process (QM). Only SafeContext, SafeWatchdog and SafeCOM (End-to-end protection library) are developed using a safety development process.

For most safety-related systems the software needs to assure input or output operations with certain integrity (ASIL). For example, the software needs to disable the actuator by toggling a digital output line of the ECU.

Thus, at least the MCAL has to be developed according to a safety development process.

Currently, there are two different methods to safely access the I/O. One way is to establish confidence in the I/O Hardware Abstraction. The other way is to develop a Complex Driver according to a safety development process.

The I/O Hardware Abstraction (IoHwAb) provided by Vector is a template. The customer has to apply its safety development process on the I/O Hardware Abstraction. Additionally, the RTE functions that are used to interact with the I/O Hardware Abstraction need to be tested by the customer.

If a Complex Driver is used for accessing the I/O, the Complex Driver has to be developed according to the customer’s safety development process. The Complex Driver can then be assigned the same ASIL as the application implementing the safety mechanisms. The Complex Driver and the application can then be put into the same memory partition.

We recommend not modelling the Complex Driver in the RTE and directly call functions of the Complex Driver. The RTE is then excluded from the communication between the application and the Complex Driver. Excluding the RTE decreases test efforts.

4 Example use-cases

In this section three example use-cases are described to show different possible setups for MICROSAR.

4.1 ECU with direct I/O

This use-case includes a system with a single ECU that performs a safety-related function. The ECU does not need any safety-related information through a communication bus.

An example of this use-case could be an ECU executing a control loop (analog input, control algorithm, analog output). For safety reasons the controlling actuator has to be additionally enabled by the application using digital outputs. The LIN communication is not safety-related in this example use-case.

The Complex Drivers are necessary, because the current Vector MICROSAR stack is not developed according to ISO 26262.

This use-case covers the simplest systems.

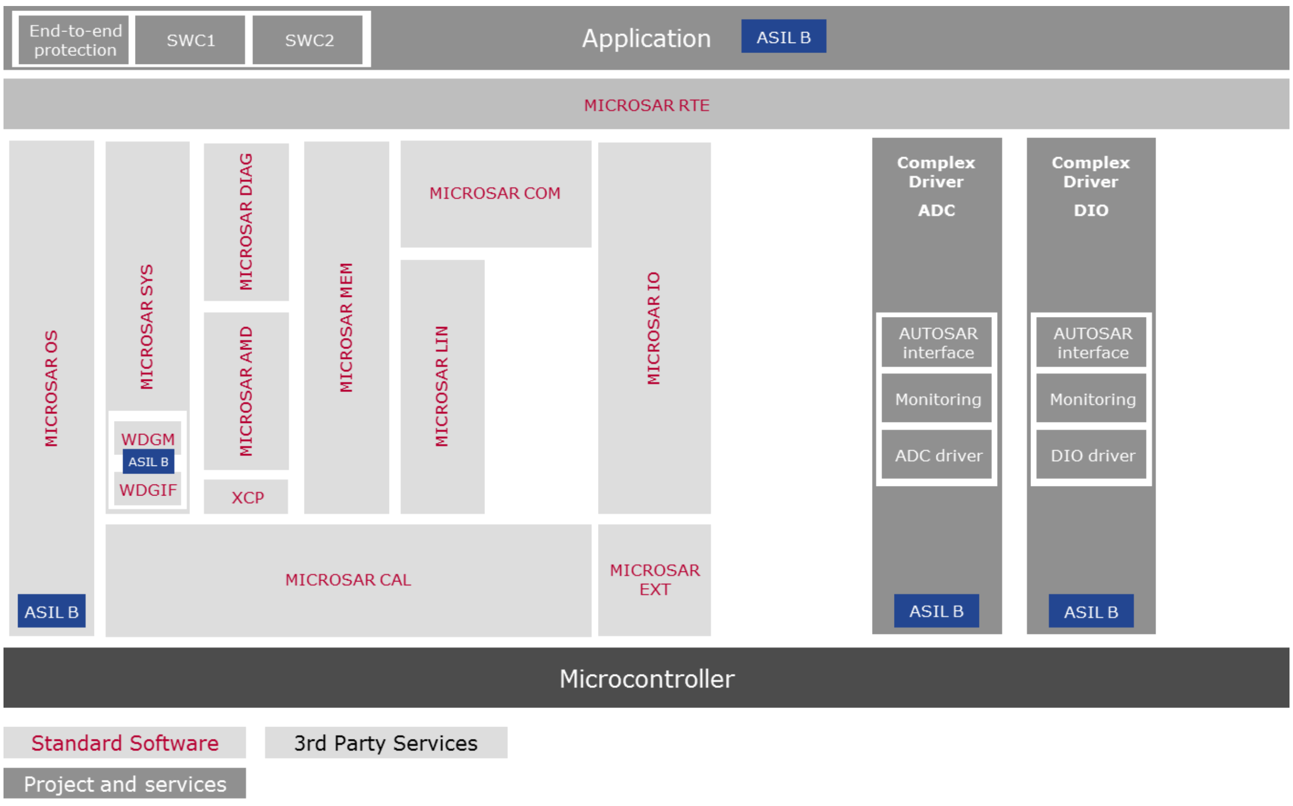

Figure 4-1 Example software architecture for an ECU with direct I/O

Property	Value
Number of cores of microcontroller	1
Safety-related I/O	Analog and digital
Safety-related communication bus	None
Highest ASIL	ASIL B
SC of operating system	SC3
MICROSAR BSW	Non-trusted
OS Applications	> Application with SWC1, SWC2, CD for ADC and CD for DIO > Application with MICROSAR BSW
Applied Vector products	SafeContext, SafeWatchdog, SafeCOM

Table 4-1 Properties of ECU with direct I/O use-case

4.2 ECU with direct I/O and safety-related bus communication

This use-case includes a system with multiple ECUs that perform a safety-related function. The ECUs are connected by a FlexRay bus. The FlexRay bus is used to communicate safety-related information.

An example of this use-case could be two ECUs for battery management of hybrid cars. One ECU is responsible for protecting and wear-leveling the cells. The other ECU controls the charger. The communication between them is safety-related and protected by the end-to-end protection library, i.e. SafeCOM.

The shown ECU also has safety-related direct I/O, e.g. for measuring cell voltage. This safety-related direct I/O is implemented using Complex Drivers. The Complex Drivers are necessary, because the current Vector MICROSAR stack is not developed according to ISO 26262.

The time synchronization using schedule tables may be helpful in such a scenario.

This use-case covers the more advanced systems.

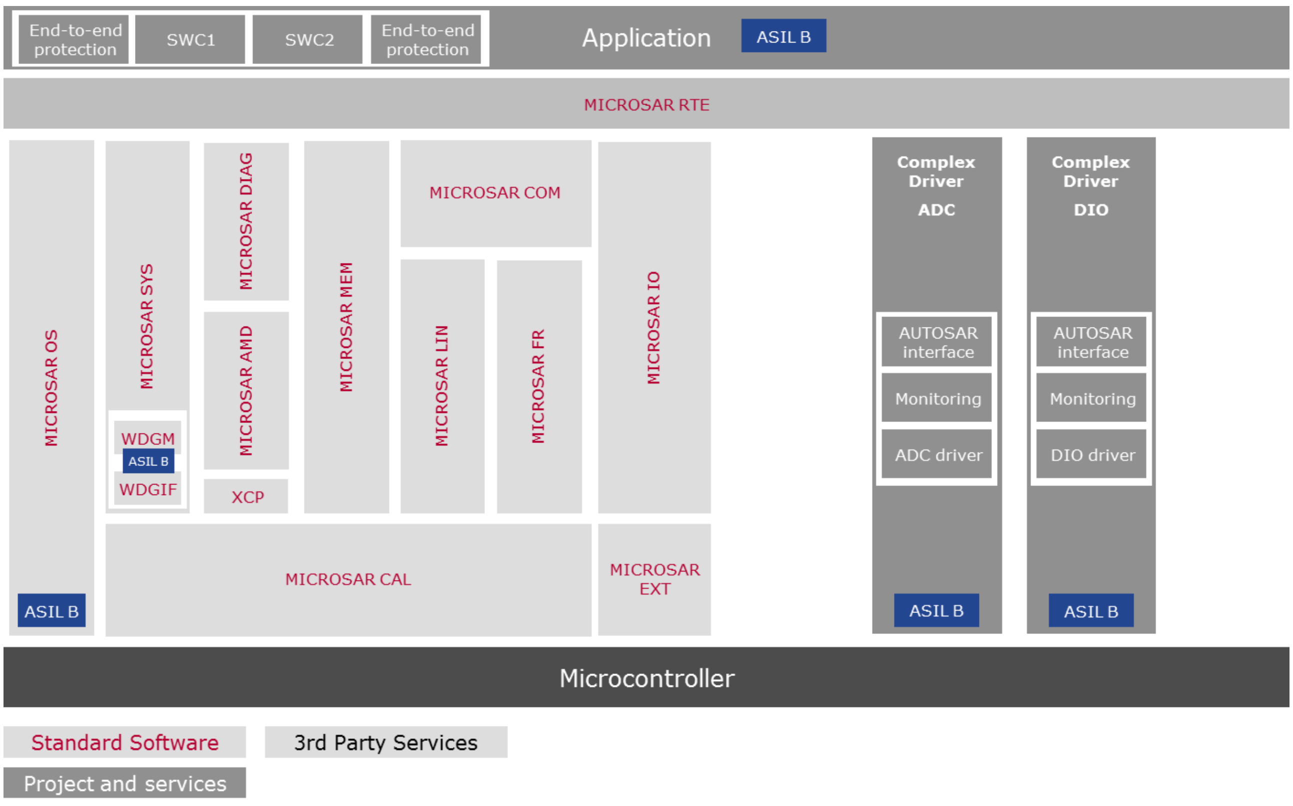

Figure 4-2 Example software architecture for an ECU with direct I/O and safety-related bus communication

Property	Value
Number of cores of microcontroller	1
Safety-related I/O	Analog and digital
Safety-related communication bus	FlexRay
Highest ASIL	ASIL B
SC of operating system	SC3
MICROSAR BSW	Non-trusted
OS Applications	> Application with SWC1, SWC2, CD for ADC and CD for DIO > Application with MICROSAR BSW
Applied Vector products	SafeContext, SafeWatchdog, SafeCOM

Table 4-2 Properties of ECU with direct I/O and safety-related bus communication use-case

4.3 Mixed ASIL SWCs with safety-related bus communication

This use-case includes a system with multiple ECUs that perform a safety-related function. The ECUs are connected by a FlexRay bus and a CAN bus. Both buses are used to communicate safety-related information.

An example of this use-case could be a gateway ECU that on the one hand controls the head lights via CAN bus, on the other hand controls the display information from the airbag system, which is received from the FlexRay bus. Each safety-related communication is protected by the end-to-end protection library, i.e. SafeCOM.

The shown ECU does not have safety-related direct I/O, e.g. does not make use of digital I/O to enter the safe state. Thus, no Complex Drivers are necessary.

The time synchronization using schedule tables may be helpful in such a scenario. In either case it is sensible to use an SC4 operating system, to guarantee freedom of interference between the different safety-related functions.

This use-case covers the more advanced systems.

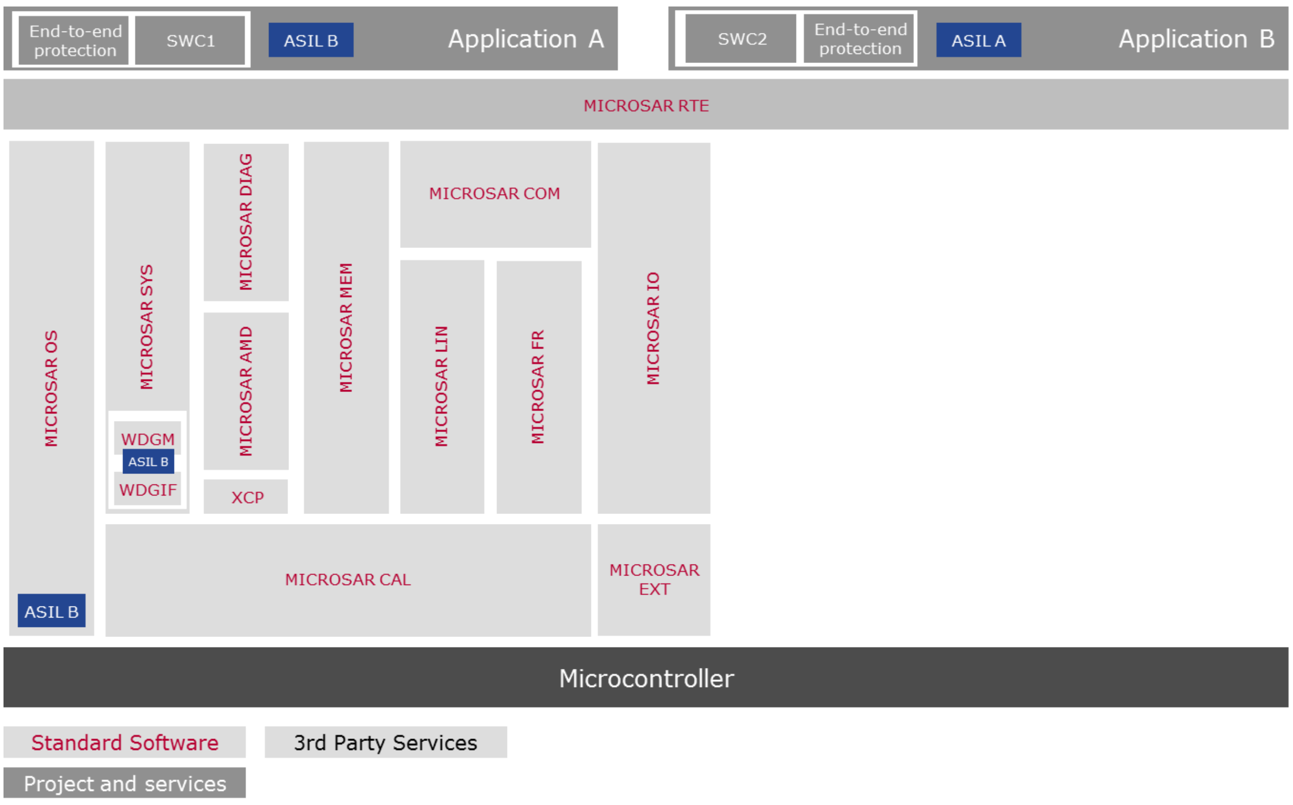

Figure 4-3 Example software architecture for an ECU with mixed ASIL SWCs and safety-related bus communication

Property	Value
Number of cores of microcontroller	1
Safety-related I/O	None
Safety-related communication bus	FlexRay, CAN
Highest ASIL	ASIL B
SC of operating system	SC4
MICROSAR BSW	Non-trusted
OS Applications	> Application with SWC1 incl. end-to-end protection library > Application with SWC2 incl. end-to-end protection library > Application with MICROSAR BSW
Applied Vector products	SafeContext, SafeWatchdog, SafeCOM

Table 4-3 Properties of Mixed ASIL SWCs with safety-related bus communication use-case

5 Recommendations for the MICROSAR stack

5.1 Initialization

We recommend creating your own start-up code that (if applicable):

1. Enables the lock-step
2. Tests the volatile memory
3. Tests the non-volatile memory for integrity when loading it into the volatile memory

5.2 ECU State Manager (EcuM)

The ECU State Manager (EcuM) component has to be developed in accordance to ISO 26262, since it usually has an effect on the functional safety of the system.

The customer has the chance to use Vector's EcuM code (generated and manually coded) to achieve the desired ASIL.

The customer may also ask Vector for verification support of the EcuM component.

We recommend keeping the EcuM as simple as possible by the following measures:

> Move the initialization of drivers into the StartPostOS Sequence (see [3]), i.e. BswM_Init(). Initializing drivers in StartPostOS is especially necessary for drivers with lower ASIL than highest ASIL of all software components.

> Do not use post-build loadable configuration

> Only initialize and configure drivers that have the same ASIL as the EcuM

If SafeWatchdog is activated during initialization, the time for StartPostOS Sequence (see [3]) needs to be considered for dimensioning the window of the SafeWatchdog.

5.3 Basic Software Mode Manager (BswM)

The Basic Software Mode Manager (BswM) component has to be developed according to ISO 26262, since it usually has an effect on the functional safety of the system as it influences the scheduling of the system.

The customer has the chance to use Vector's BswM code (generated and manually coded) to achieve the desired ASIL.

The customer may also ask Vector for verification support of the BswM component.

We recommend keeping the BswM component as simple as possible.

The BswM does not have to be developed according to ISO 26262, if SafeWatchdog also monitors control flow on a global level. A safety analysis has to show if using SafeWatchdog is appropriate.

5.4 Development Error Tracer (Det)

The Development Error Tracer (Det) component is not recommended for series production usage.

The Det component has to be developed according to ISO 26262 if used in series production, since it usually has an effect on the functional safety of the system.

The Det component usually runs in the OS application that also contains the safety-related SWCs. If the Det is not used, it does not have to be developed according to ISO 26262.

The customer has the chance to use Vector’s Det code (generated and manually coded) to achieve the desired ASIL. Qualifying the Det code may especially be sensible if no difference between tested and production code is desired.

The customer may also ask Vector for verification support of the Det component.

We recommend keeping the Det component as simple as possible.

5.5 Diagnostic Event Manager (Dem)

The Diagnostic Event Manager (Dem) may be part of the degradation concept. The Dem can be used to store the degraded state of e.g. the ECU. A degraded state may be a safe state. In this case the Dem needs to be developed according to ISO 26262.

The customer has the chance to use Vector’s Dem code (generated and manually coded) to achieve the desired ASIL.

The customer may also ask Vector for verification support of the Dem component.

5.6 NVRAM Manager

For the NVRAM Manager the recommendations of section 3.3.2 apply.

5.7 Run-Time Environment (RTE)

For further information refer to the Safety Guide of the RTE (see [5]).

5.8 End-to-End Protection (E2E)

We recommend using the E2E Protection Wrapper as described in section 12.1 of [4]. The E2E Protection Wrapper provided by Vector is developed according to ISO 26262.

5.9 Operating System (OS)

We recommend enabling the stack monitoring facility if the *diagnostic coverage* is not sufficient for the requirements of the system.

For further information refer to the Safety Manual of the SafeContext operating system.

5.10 Interrupt service routines (ISRs)

We recommend avoiding category 1 ISRs.

All category 1 ISRs must be developed according to ISO 26262, since ISRs can corrupt the system state in e.g. either memory or timing behavior. It is assumed that for category 2 ISRs memory protection is activated.

For further information refer to the Safety Manual of the SafeContext operating system.

5.11 Microcontroller Abstraction Layer (MCAL)

Microcontroller Abstraction Layer initialization is usually done by EcuM. Especially, the MCU and Port Drivers are usually initialized in a very early stage during start-up.

The DIO Driver is accessed by the CAN interface if the CAN transceiver has an additional port to separately enable the CAN transceiver.

If e.g. the DIO Driver is necessary for a safety mechanism, the DIO Driver cannot be used by ASIL software.

A solution using Complex Drivers and an appropriate MPU setup may be found.

6 Procedural requirements

Vector does not provide supporting material for a tool qualification of the compiler and debugger according to ISO 26262 Part 8. The customer is responsible to perform the necessary qualification activities.

Vector assumes that both compiler and debugger will be classified with a TCL 2. The debugger does not necessarily have to be taken into account, only if it is used for testing purposes. Otherwise, it does not matter how the debugger is classified.

If not stated otherwise the source code of software delivered by Vector must not be modified by the customer. It is recommended that the customer checks if the source code has been modified, e.g. by comparing it to the original delivery.

The customer has to assure that sufficient resources are available to operate the software delivered by Vector. This includes:

> selection of a microcontroller with sufficient RAM, ROM and calculation performance

> dimensioning of sufficient stack size

> selection of appropriate scheduling properties

7 Assumptions of Vector’s safety solution

This section resolves some of the assumptions made by different components of Vector’s safety solution.

7.1 Assumptions of RTE

The following assumptions are extracted from [5].

Assumption	Description	Can be shown by
ASS_RTE_1	The OS provides freedom from interference for different OS Applications with regards to memory. This means that code in one OS Application cannot destroy memory in another OS Application.	Using SafeContext OS with SC3.
ASS_RTE_2	The AUTOSAR Memory Abstraction for the target platform and the OS make it possible to assign RTE/BSW variables to specific OS Applications so that they can only be written by code that is executed within this OS Application. Moreover in case of Multicore, the RTE variables are mapped to noncacheable RAM so that they can be accessed by all cores.	Using a microcontroller with an MPU and SafeContext OS with SC3.
ASS_RTE_3	The tool chain initializes global variables or the API Rte_InitMemory is called before the OS is started. Rte_InitMemory initializes variables from different OS Applications. Therefore it needs to be started without memory protection.	Qualification for customer configuration of Rte_InitMemory and calling Rte_InitMemory prior to StartOS.
ASS_RTE_4	The OS allows non protected reads to RTE/BSW variables within the same and foreign OS Applications.	Using SafeContext OS with SC3 allows read access to every memory location.
ASS_RTE_5	Freedom from interference with regards to CPU runtime is provided through external means, for example with the help of a control flow monitor. The mechanisms for it are either implemented in a way that the RTE cannot deactivate them or a review is performed that checks that the RTE does not impact their operation.	Using SafeWatchdog to implement timing protection mechanisms.
ASS_RTE_6	The OS APIs that are used by the RTE in ASIL parts of the code can be called from different contexts without interference: > TerminateTask > SuspendOSInterrupts or osRteDisableLevel (MICROSAR OS) > ResumeOSInterrupts or osRteEnableLevel (MICROSAR OS) > GetSpinlock (Multicore Systems) > ReleaseSpinlock (Multicore Systems)	Using SafeContext OS with SC3.

Assumption	Description	Can be shown by
ASS_RTE_7	The OS provides at least the APIs > SuspendOSInterrupts or osRteDisableLevel (MICROSAR OS) > ResumeOSInterrupts or osRteEnableLevel (MICROSAR OS) with the same or higher ASIL than the SWCs In Multicore Systems, the OS also needs to provide the APIs > GetSpinlock > ReleaseSpinlock with the same or higher ASIL than the SWCs	Using SafeContext OS with SC3.
ASS_RTE_8	The RTE configuration is chosen in such a way that the OS/System mechanisms for freedom from interference (memory and runtime) can also be used to implement freedom from interference for SWCs with different ASIL. This makes it necessary to map SWCs with different ASIL to different OS Applications. All OS Applications with SWCs that do not have the highest ASIL need to be non-trusted. This includes the OS Application of the BSW.	Appropriate configuration. Coaching by Vector on configuring a safe system may be ordered separately.
ASS_RTE_9	The RTE configuration is chosen in such a way that no OS APIs need to be called in the RTE APIs or the TASK bodies that violate the safety requirements of the ASIL SWCs. The RTE code calls the following OS APIs: > SetRelAlarm > CancelAlarm > SetEvent > GetEvent > ClearEvent > WaitEvent > GetTaskID > ActivateTask > TerminateTask > Schedule > ChainTask > GetResource > ReleaseResource In case of multicore systems also the API > GetCoreID	Using SafeContext OS with SC3.

Assumption	Description	Can be shown by
	is called.	
ASS_RTE_10	The RTE configuration is chosen in such a way that no SWC needs to directly call methods in (Service-) SWCs with lower ASIL and no (Service-) SWCs with lower ASIL needs to call methods in ASIL SWCs except for the case when the SWCs explicitly allow this kind of usage. If necessary, this work is delegated to wrapper SWCs in the same OS Application as the called/calling SWC. Direct calls can moreover be avoided when the server runnables are mapped to tasks.	Appropriate configuration. Coaching by Vector on configuring a safe system may be ordered separately.
ASS_RTE_11	The RTE configuration is chosen in such a way that the RTE APIs or TASKS for a SWC do not contain calls to BSW modules with lower ASIL than the SWC itself that might cause interference. If necessary, this work is delegated to wrapper SWCs in the same OS Application as the BSW modules. For external communication, the RTE proxies the calls to the Com module.	Appropriate configuration. Coaching by Vector on configuring a safe system may be ordered separately.
ASS_RTE_12	The RTE does not need to provide freedom from interference for communication. In an AUTOSAR system, the E2ELibrary that is directly called by the SWCs is responsible for Safe communication. Nevertheless, the RTE provides APIs that can be called by the E2ELibrary.	Using SafeCOM.
ASS_RTE_13	The Generated RTE code for ASIL SWCs is qualified according to the requirements of ISO 26262 by the integrator so that it reaches the same ASIL as the SWCs themselves. This is necessary because the RTE Generator was only developed with Vectors standard quality management (QM).	Qualifying the generated code according to ISO 26262 Part 8.
ASS_RTE_14	The hardware is suited for safety relevant software according to the requirements of ISO 26262. The hardware requirements are mostly determined by the SWCs that shall be supported by the RTE. The MICROSAR RTE does not impose other hardware safety requirements as those that are already required by the SWCs and the OS.	Development according to ISO 26262 for the whole item.
ASS_RTE_15	The development tool chain (for example editors, compilers, linkers, make environment, flash utilities) is suited for the development of safety relevant software according to the requirements of ISO 26262. All tools need to reach the appropriate Tool Confidence Level (TCL).	Tool evaluation. Vector can provide assistance on evaluation Vector tools.

Table 7-1 Assumptions of RTE

7.2 Assumptions of SafeWatchdog

The following assumptions are extracted from [12].

Assumption	Description	Can be shown by
282785	The software execution environment shall be able to run software according to requirements of up to the system's required ASIL. This also includes: > free from interference among the SW components (see 282807) > supervision by an extern measures (see 282795) > the hardware shall consist of an MCU with all required hardware to run according to system specifications (i.e. safe HW to detect/avoid e.g. bit-flips by means of start-up checks, cyclical checks, ECC check, ...). > the hardware shall be composed of components that are qualifiable up to the desired ASIL of the system.	Using appropriate hardware architecture and components, using SafeContext OS with SC3 and hardware watchdog with at least a different time base (depending on the required ASIL).
297946	The software execution environment shall provide methods for mutual exclusion.	Using SafeContext OS with SC3.
297948	Such methods are disabling of interrupts, locks, semaphores etc. Especially disabling of interrupts is often used to gain exclusive access to resources or perform multiple operations atomically.	Using SafeContext OS with SC3.
282807	The software platform shall provide an execution environment that is capable of running multiple software components with freedom from interference from each other.	Using SafeContext OS with SC3.
282809	The S-WdgM and the supervised application are considered as separate SW components with freedom from (unintended) interference. Freedom from interference can be achieved by e.g. microcontroller with MPU.	Using SafeContext OS with SC3 and assigning different OS Applications to the SafeWatchdog and the application.
282795	The system shall provide measures to detect timing violations of the MCU itself and trigger a corresponding fault reaction, if necessary: > There shall be an external safety measure that can handle a failure of the MCU. > It must be possible for the MCU to trigger the external safety measure if the MCU detects a fault within itself.	Appropriate system architecture.
282797	MCU faults need to be detected by external measures. An extern WD can detect MCU and S-WdgM fault.	Depending on the required ASIL, an external watchdog is necessary.
282789	The MCU shall be able to perform a safe startup to the point of where the S-WdgM is safely initialized.	See section 5.1.
265876	The FLASH memory correctness shall be checked at ECU startup time. An ECC or comparable check shall be used at run-	See section 5.1.

Assumption	Description	Can be shown by
	time.	
263975	The generated code holds a checksum over some significant fields (e.g. version) to check that: > the generated code belongs to the S-WdgM code according to version information and > the generated code is not overwritten by other code at the flashing process. The checksum is checked with every run of the function WdgM_Init (). A failed check yields WDGM_E_PARAM_CONFIG. Note that the checksum does not cover the complete configuration and cannot thoroughly detect when the configuration memory is corrupted (like bit flips).	See section 5.1.

Table 7-2 Assumptions of SafeWatchdog

8 Glossary and Abbreviations

8.1 Glossary

For definitions see section 1.3.

8.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CD	Complex Driver
DEM	Diagnostic Event Manager
DET	Development Error Tracer
E2E	End-to-end protection
EAD	Embedded Architecture Designer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
OEM	Original Equipment Manufacturer
PPORT	Provide Port
QM	Quality measures
RPORT	Require Port
RTE	Runtime Environment
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 8-1 Abbreviations

9 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com