

Compliance Documentation MISRA-C:2004

Version 3.0

2017-10-09

Application Note AN-ISC-8-1213

Author	Andreas Raisch
Restrictions	Customer Confidential – Vector decides
Abstract	This document explains the MISRA-C and HIS Metric compliance enforcement. It also contains the project specific MISRA and code metric deviations including their justifications.

Table of Contents

1	Overview	3
1.1	Purpose, goal	3
2	Enforcing MISRA-C Compliance.....	4
2.1	Software Engineering Context	4
2.2	Programming Language and Coding Context	4
2.3	Project specific MISRA Subset	4
2.4	MISRA Compliance Matrix	4
2.5	Hints on Compiler Selection.....	6
2.5.1	Deactivated QA-C Rules for MICROSAR Product Analysis	7
2.6	Deviation Procedure	7
2.7	Usage of C99 Features	8
2.7.1	Usage of Inline Functions	8
2.7.2	Usage of C99 Data Types.....	9
2.7.3	No Usage of “//” Comments	9
3	Enforcing Code Metric Compliance	10
3.1	HIS Metric Compliance Matrix	10
4	Appendix	13
4.1	MISRA Justification for “Project Deviations”	13
4.2	Code Metric Justification for “Product Deviations”	23
5	Additional Resources	26
6	Contacts	26

Revision List

Version	Editor	Date
---------	--------	------

1 Overview

1.1 Purpose, goal

The purpose of this document is to explain the process executed and limitations known to claim compliance of the product MICROSAR to “MISRA-C:2004 guidelines for the use of the C language in critical systems” [MISRA-C] and to HIS source code metrics [HIS-CODE].

In addition, the document also describes the handling of code metrics.

The products are developed in a product-line approach, so the wording “project” in [MISRA-C] is mapped to “product” in this context, too. Product-line approach means that a large set of reusable software-components are developed and maintained to fulfill the functional needs of different OEMs and TIER1s for any ECU in any vehicle.

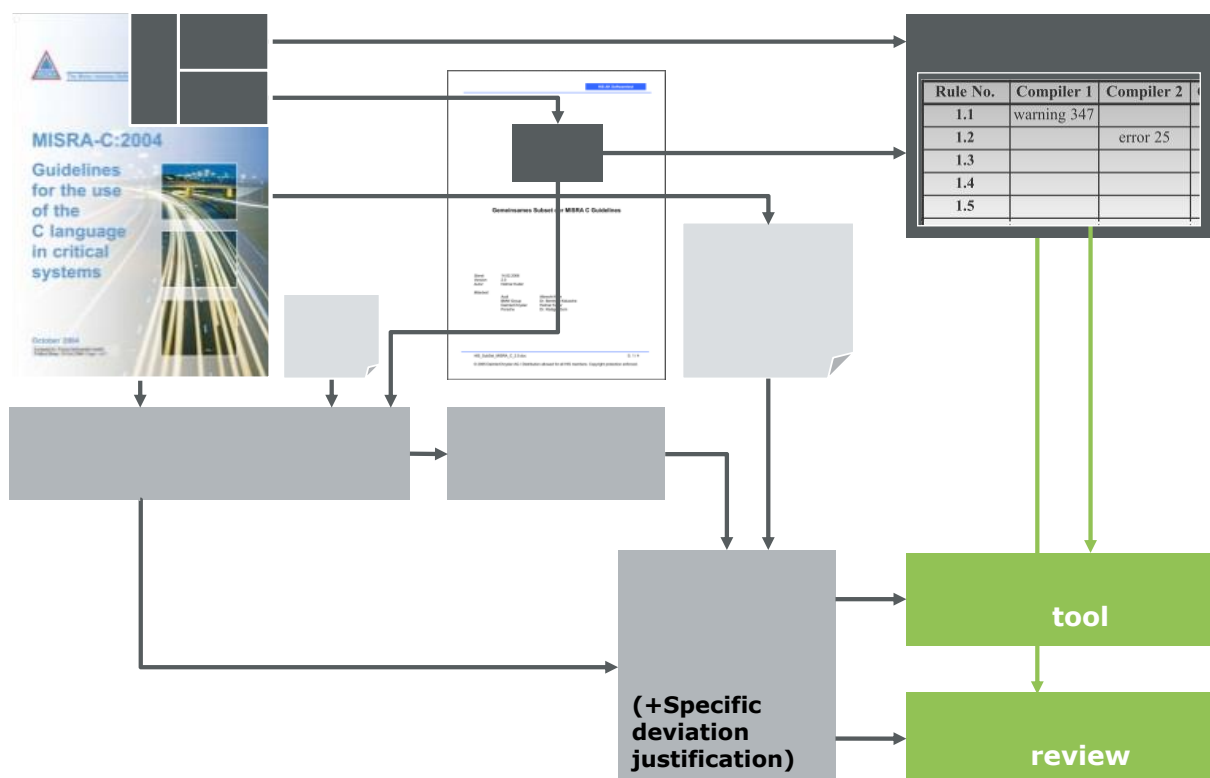


Figure 1-1 Overall MISRA process: MISRA standard – code style guides – code – compliance matrix - ...

2 Enforcing MISRA-C Compliance

2.1 Software Engineering Context

If not otherwise stated, the product is developed according AUTOMOTIVE SPICE, level 3.

2.2 Programming Language and Coding Context

Used programming languages are

- > C99 but using C90 subset only (see chapter 2.7 *Usage of C99*)
- > ASM (rarely used, only for µC-specific codes like ISR, register access, ...)

Code is developed based on a code style guide and by applying code and header templates.

Selected code metrics are measured, documented and deviations are justified.

Runtime test coverage is measured, documented and deviations are justified.

2.3 Project specific MISRA Subset

[HIS-MISRA] defines all rules to be “required”, so no rule is “advisory”.


All 141 rules in [MISRA-C] are analyzed for product relevance. If a rule is judged to be not relevant, it is justified in chapter 2.4 *MISRA Compliance Matrix*.

2.4 MISRA Compliance Matrix

The compliance matrix for the product is:

- > All rules not explicitly listed in *Table 2-1* are checked by the tool QA-C (125 rules)
- > Rules listed in *Table 2-1* and marked “Code inspection” are manually checked (8 rules)
- > Rules listed in *Table 2-1* and marked “Not applicable” are not checked (8 rules); the justification is added within the table.

Rule	Description	Enforcement Strategy
1.3 (req)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.	Not applicable Justification: Multiple compilers or languages are not used in embedded software components developed at PES. The only exception is the usage of assembler code. Here however the assembler which belongs to the compiler is used so there is no issue with the object code interface.
1.4 (req)	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	Not applicable Justification: Since we use the compiler which is prescribed by our customer, we are not free in the compiler selection. There are still compilers which support only 6 characters, so the software has to support this. Since especially during implementation of higher layers the used compiler is not known it is not possible to check this in the development phase. This is verified during delivery test with the prescribed compiler.

Rule	Description	Enforcement Strategy
		 Note [MISRA-P1] addresses this topic in Permit/MISRA/C:2004/5.1.A.1.
1.5 (adv)	Floating-point implementations should comply with a defined floating-point standard.	Not applicable Justification: Floating point operations are not used in embedded software components developed at PES.
2.4 (adv)	Sections of code should not be 'commented out'.	Code inspection
3.2 (req)	The character set and the corresponding encoding shall be documented.	Not applicable Justification: Character strings are not used in embedded software components developed at PES.
3.3 (adv)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account. Comment: There is a potential issue in case of the division of negative signed integers because the compiler behavior is not standardized.	Not applicable Justification: Division of negative signed integers is not used in embedded software components developed at PES.
3.5 (req)	If it is being relied upon, the implementation-defined behavior and packing of bit-fields shall be documented.	Not applicable Justification: In the embedded environment the handling of bit-fields is documented for each compiler and hardware in the compilers' user manual. Beyond this, tests minimize the risk indicated by this rule.
3.6 (req)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	Not applicable Justification: The C standard library or other libraries are not used in embedded software components developed at PES.
5.7 (adv)	No identifier name should be reused. Comment: the term "system" in [MISRA-C] relates to the unit on which the MISRA analysis is applied, i.e. a software component. See MD_MSR_5.7 for deviations to this rule.	Code inspection
10.5. (req)	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand	Code inspection
17.1 (req)	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Code inspection
17.2. (req)	Pointer subtraction shall only be applied to pointers that address elements of the same array	Code inspection
17.3 (req)	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Code inspection

Rule	Description	Enforcement Strategy
18.2 (req)	An object shall not be assigned to an overlapping object. Comment: This is related to use the same memory by different objects (variables) at the same time without using unions.	Code inspection
18.3 (req)	An area of memory shall not be reused for unrelated purposes. Comment: This is related to use the same memory for different purposes at different points of time (example: dynamic memory management) without using unions.	Code inspection
20.3 (req)	The validity of values passed to library functions shall be checked.	Not applicable Justification: The C standard library is not used in embedded software components developed at PES. Our own libraries provide sufficient internal checking in contrast to the C standard library.
Partly 19.8 (req) (QA-C rule 850)	QA-C rule 850 "Macro argument is empty", mapped to MISRA-C:2004 rule 19.8 "A function-like macro shall not be invoked without all of its arguments". Rule deactivated because of bug in used QA-C version (CR 12354, Message 0850 was sometimes generated incorrectly when performing macro substitution with a macro defined with an empty replacement list)	Code inspection

Table 2-1 Rules not checked by MISRA analysis tool PRQA- QA-C

2.5 Hints on Compiler Selection

The complexity of an [AUTOSAR] based software system exceeds typically the maximum numbers of structs, unions, includes and nested #if levels defined for C90. These C90 limits originate in early C compiler implementations with limited RAM to store definitions. Nearly all current compilers support sufficient support for huge numbers of structs, unions, include and nested #if levels.

Such compilers also support more than 31 characters to identify internal and external identifiers.

Vector implements AUTOSAR compliant to the AUTOSAR SWS with focus to stay within the given limits. Nevertheless, combining the different BSW modules to a stack, this will result in exceeding C90 limits. Because overstepping the limits depends on the concrete project and the used databases no value forecast can be given¹.

We strongly recommend that the ECU project validates the used compiler against its limits concerning maximum numbers of structs, unions, includes and nested #if levels and significant characters for internal and external identifiers.

Additionally we strongly recommend that the ECU project configures its static code checker tool in a way that the concrete compiler limits are part of the ECU project verification activities.

¹ A typical reason for exceeding the number of 1024 macros is using the COM macro API. Enabling the macro API is a decision done by the ECU integrator to improve runtime and memory consumption. Negative effects are e.g. multiple MISRA deviations.

Priority	Rule	Rationale
		runtime, ...), product-line code maintainability, ...

Table 2-3 Priority of measures to prevent MISRA deviations

This document addresses a product-line approach, therefore chapter 4.1 *MISRA Justification for “Project Deviations”* list the **Product Deviations**. The maintenance of the Product Deviation list is part of the product-line development process.

If a part of the product has a **Specific Deviation**, the justification is done locally. The Specific Deviation is documented within the code and verified in the code inspection process, too.

Each deviation is marked as shown in Figure 2-1. The justification is either a link on a Product Deviation (chapter 4.1 *MISRA Justification for “Project Deviations”* within this document) or a link to a Specific Deviation justified within the source code itself.

The justification in the source code is based on this template:

```
/* Justification for module-specific MISRA deviations:

    : MISRA rule xx.y
    Explanation of reason for deviation on <RuleNo>
    Explanation of risk for <RuleNo>
    Explanation how to prevent risk occurrence

*/
```

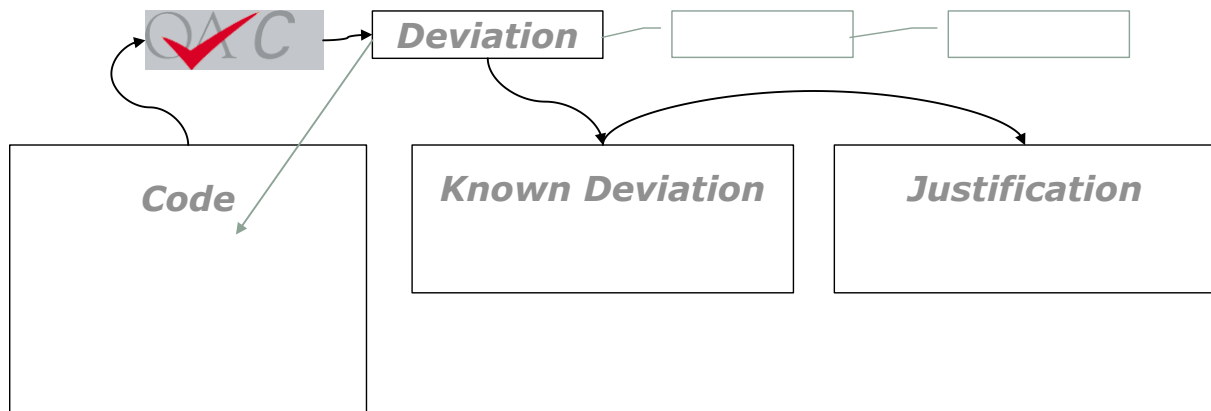


Figure 2-1 Style for justifying a deviation in code

2.7 Usage of C99 Features

The product uses the C90 language subset plus the here listed C99 features.

2.7.1 Usage of Inline Functions

The product uses *inline functions* to reduce complexity and improve the readability and testability of the code.

The code style guide requires the use of the [AUTOSAR] compiler abstraction concept LOCAL_INLINE when using *inline functions*. This approach provides means to define a compiler specific static storage class definition as required by e.g. MISRA_C:2012, “rule 8.10 An *inline function* shall be declared with the static storage class”.

**Note**

Not all (C90) compilers support “static inline”. Therefore the mapping can be adapted to the supported compiler and usage from “static inline” to “inline” or to “” (i.e. explicit function call instead of compilation time inline).

**Note**

Compiler.h delivered with the product is adapted to the ordered compiler. When using a different Compiler.h (e.g. from MCAL package), please verify correct LOCAL_INLINE assignment.

**Note**

[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.C.1.

2.7.2 Usage of C99 Data Types

The code style guide requires the use of the [AUTOSAR] platform abstraction concept to declare and define variables.

**Note**

Not all (C90) compilers support types to define 64 bit data types. Such a compiler cannot be used to compile the product in case it contains components requiring 64 bit data types. Therefore use of 64 bit data types is restricted to (1) components with typical usage on at least 32bit μ C (e.g. ethernet stack) and (2) when explicitly defined by AUTOSAR (e.g. as data type in interface).

**Note**

Platform_Types.h delivered with the product is adapted to the ordered platform. When using a different Platform_Types.h (e.g. from MCAL package), please verify correct type definition.

**Note**

Components developed for a specific platform and compiler (e.g. MCAL or OS) may use the 64 bit data types directly.

**Note**

[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.A.1.

2.7.3 No Usage of “//” Comments

Usage of “//” comments is not allowed for the product.

This is covered by MISRA-C:2004, rule 2.2 “Source code shall only use C-style comments.”

3 Enforcing Code Metric Compliance

3.1 HIS Metric Compliance Matrix

The code for the product shall comply with HIS source code metrics [HIS-MISRA].

Analysis of the HIS source code metrics in the product context has shown that only few of the metrics are helpful to measure and vote the products quality.

The product has a high configurability, so metric deviations may occur in all or in just a few, specific setups. The components of the product are thus analyzed for a set of defined use-cases and metric deviations are analyzed. If the analysis result is “accept metric deviation”, a justification is added to the code.

HIS Metric (QA-C metric)	Rationale and Enforcement Strategy
Comment Density (COMF)	HIS: COMF > 0.2 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none">> Product consists out of static and generated code files, some very small, some large, some with just tables and defines, some with many complex code structures. Comments shall be applied where necessary to improve the readability, not to satisfy a metric.> Adding MISRA justifications comments in the code results (in relevant files) in more than 20% metric target is typically fulfilled but not the intended improvement in maintainability.> Vector focuses on the outcome of code inspection to improve also maintainability concerning good and helpful comments.
Estimated static path count (STPTH)	HIS: 1..80 This metric is measured and justified per deviation or class of deviations.
Cyclomatic Complexity (STCYC)	HIS: 1..10 This metric is measured and justified per deviation or class of deviations.
Number of GOTOs (STGTO)	HIS: 0 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none">> Use of GOTO is allowed in Vector code for limited use-cases.> Use of GOTO has to be already justified as MISRA deviation per use.> Mandatory metric STBAK is applied to check for illegal use of GOTO
Number of functions calling this function (STM29)	HIS: 0..5 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none">> The metric is aimed to detect “up to now unknown” functions in a “black-box system” which were critical due to be called by many other functions and shall thus be under specific focus.> By contrast, the product is designed to provide functionality to an (ECU) project based on defined standards (ISO, AUTOSAR, ...). Thus the usage of APIs is known, predefined and not in our control. Justifying the usage will indicate only the usage in e.g. our test-projects but not that in the concrete final project.
Number of distinct function calls (STCAL)	HIS: 1..7 This metric is measured and justified per deviation or class of deviations.

HIS Metric (QA-C metric)	Rationale and Enforcement Strategy
Number of function parameters (STPAR)	HIS: 0..5 This metric is measured and justified per deviation or class of deviations.
Number of statements in function (STST3)	HIS: 1..50 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none"> > Project analysis has shown that if STST3 is out of limit and the function is complex, also STCYC and STPATH are out of limits. So we focus on STCYC and STPATH for analysis.
Maximum nesting of control structures (STMIF)	HIS: 0..4 This metric is measured and justified per deviation or class of deviations.
Number of exit points (STM19)	HIS: 0..1 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none"> > Safe programming requires runtime checks. Due to runtime efficiency, the checks have to be disabled for production code. Thus the DET checks are typically macros and are added to the begin of a function. To keep code structure simple, additional returns are allowed here, too. > Use of return statement has to be already justified as MISRA deviation per use. > Vector focuses on the outcome of code inspection to check for correct usage of return statements.
Language set (VOCF)	HIS: 0..4 This metric is measured but deviations are not justified. (Metric calculation done according PRQA specification) Justification: <ul style="list-style-type: none"> > Value for very complex components has been found in HIS range (= ok), but nearly empty components have been reported as dramatically out of range (e.g. >2500). > Summary is, that analysis of metric values in the product context has not shown any significant hint on improving code maintainability or other benefits.
Number of recursions across project (STNRA)	HIS: 0 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none"> > Vector coding guidelines forbid the use of recursions to implement functional behavior. > Vector develops standard software according e.g. AUTOSAR, so API usage definition is mainly out of our scope > Vector code is just a part of a "final" ECU code, so recursions caused by customer-project specific API or configuration data use cannot be detected in our selected product test use-cases. Measurement is done to be aware of potential recursions. > (The tool QAC has no reliable capability to detect the absence of recursions due to insufficient pointer analysis)
Number of MISRA subset violations (NOMV)	HIS: 0 This metric is measured but deviations are not justified. Justification: <ul style="list-style-type: none"> > MISRA deviations are already justified per occurrence during the

HIS Metric (QA-C metric)	Rationale and Enforcement Strategy
	development process of each component. > The components are assembled to a large number of different projects with different configurations resulting in different number of MISRA deviations. Due to the product-line approach and the component-local handling, creating justification above the component level is not reasonable.
Number of MISRA subset violations per rule (NOMVRP)	HIS: 0 This metric is measured but deviations are not justified. Justification: > Same justification as for NOMV
Stability Index „S _i “	HIS: <=1 This metric is measured but deviations are not justified. Justification: > Basic idea of this metric is to show that the code has less changes from release to release. This is typically true for a concrete ECU project in its given timeline. The time-boxed product-line development has features of different products developed at the same time and each release is typically “production”. In addition, the code consists on a static part and a generated part. The generated part depends on the used configuration database. The reference database is updated with each time-box, too. Due to those effects creating justifications for the product is not reasonable.

Table 3-1 HIS code metric compliance matrix

4 Appendix

4.1 MISRA Justification for “Project Deviations”

This chapter contains the list of the agreed and released MISRA (Project) Product Deviations.

MISRA test reports references the Deviation ID, if applicable. If a deviation does not match the description in this chapter, it is handled similar to a Specific Deviation with a local justification.

MISRA has released [MISRA-P1] in 04/2016 with permits to some MISRA-C:2004 rules. Some of these permits are similar of the product deviations described in this document. If there is a dedicated MISRA permit, a link is added. [MISRA-P1] contains additional permits not contained in this chapter.

Deviation ID	MD_MSR_1.1
Violated rule	1.1 (ISO C standard compliance)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.E.1.
Reason	To access directly to hardware which is required to develop operating systems, drivers and flash boot loaders the usage of special compiler features such as qualifiers and pragmas is usually required. Further such concepts are sometimes required for memory mapping.
Potential risks	The used language extension may not have been fully understood or used incorrectly. This can lead to wrong behavior of hardware dependent software parts.
Prevention of risks	During the code inspection hardware dependent software parts are checked. Each delivery is integrated and tested on the real target system where such extensions are used. Compiler selection and validation is done by ECU supplier.
Examples	Declaration of interrupt service routine: <code>__interrupt void CanTxInterrupt(void);</code> allocate constant data to CONST segment: <code>#pragma memory=constseg(CONST_DATA) :far</code>

Deviation ID	MD_MSR_1.1_639
Violated rule	1.1 (ISO C standard compliance)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.D.1.
Reason	The QA-C rule 0639 checks if the maximum number of members in a struct or union exceeds 127. This check is due to some older compilers do not support more than 127 members. In complex software architectures like the AUTOSAR based MICROSAR, the maximum of 127 is not sufficient to model the necessary structures.
Potential risks	A compiler might not correctly translate the software what might result in incorrect code.
Prevention of risks	Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation. Compiler selection and validation is done by ECU supplier.
Examples	-

Deviation ID	MD_MSR_1.1_715
Violated rule	1.1 (ISO C standard compliance)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.D.1.
Reason	<p>The QA-C rule 0715 checks if the maximum number of control structures exceeds 15. This check is due to some older compilers do not support more than 15 nesting levels.</p> <p>In complex software architectures like the AUTOSAR based MICROSAR, the maximum of 15 is in some cases not sufficient to model the necessary structures.</p>
Potential risks	A compiler might not correctly translate the software what might result in incorrect code.
Prevention of risks	<p>Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	-

Deviation ID	MD_MSR_1.1_810
Violated rule	1.1 (ISO C standard compliance)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.D.1.
Reason	<p>The QA-C rule 0810 checks if the maximum #include nesting level exceeds 8 levels. This check is due to some older preprocessors do not support more than 8 #include levels.</p> <p>In complex software architectures like the AUTOSAR based MICROSAR, the maximum of 8 #include level is not sufficient to model the necessary structures.</p>
Potential risks	A preprocessor might not correctly pre-process the software what might result in incorrect code.
Prevention of risks	<p>Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	-

Deviation ID	MD_MSR_1.1_828
Violated rule	1.1 (ISO C standard compliance)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.D.1.
Reason	<p>The QA-C rule 0828 checks if the maximum #if nesting level exceeds 8 levels. This check is due to some older preprocessors do not support more than 8 #if levels.</p> <p>In complex software architectures like the AUTOSAR based MICROSAR, the maximum of 8 #if level is not sufficient to model the necessary structures.</p>
Potential risks	A preprocessor might not correctly pre-process the software what might result in incorrect code.
Prevention of risks	<p>Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	-




Deviation ID	MD_MSR_1.1_857
Violated rule	1.1 (ISO C standard compliance)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/1.1.D.1.
Reason	<p>The QA-C rule 0857 checks if the maximum number of defined macros exceeds 1024. This check is due to some older preprocessors do not support more than 1024 macros.</p> <p>In complex software architectures like the AUTOSAR based MICROSAR, the maximum of 1024 macros per compilation unit is not sufficient to model the necessary structures.</p>
Potential risks	A preprocessor might not correctly pre-process the software what might result in incorrect code.
Prevention of risks	<p>Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	-

Deviation ID	MD_MSR_3.4
Violated rule	3.4 (Usage of #pragma directives)
Reason	The concept of AUTOSAR's Memory Mapping SWS introduces #pragma directives to create an optimized layout of the target platform's memory. The #pragma directives are therefore not localized to specific fragments in the code, but transferred to the file MemMap.h. Instead of specifying the #pragma directives at the appropriate position in the code, MemMap.h is included.
Potential risks	The used language extension may not have been fully understood or used incorrectly. This can lead to wrong behavior of hardware dependent software parts.
Prevention of risks	<p>During code inspection the Memory Mapping markers (so to say "logical sections") before each "#include <MemMap.h>" are verified and checked against the memory qualifier defined in the Compiler Abstraction file Compiler_Cfg.h. Each delivery is integrated and tested on the real target system where such extensions are used.</p> <p>The user documentation contains a table displaying which logical section of the Memory Mapping is addressed by which Compiler Abstraction memory qualifier.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	allocate constant data to CONST segment: <pre>#pragma memory=constseg(CONST_DATA) :far</pre>

Deviation ID	MD_MSR_5.1_777
Violated rule	5.1 (Identifiers (internal and external) shall not rely on the significance of more than 31 characters)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/5.1.A.1.
Reason	<p>The QA-C rule 0777 checks if all external identifiers are different within the first 31 characters. This check is due to some older preprocessors compare just the first 31 characters.</p> <p>In complex software architectures like the AUTOSAR based MICROSAR, naming rules to prefix the identifiers are allocating a significant number of characters. Many of the identifiers are generated based on their elements in the OEM-defined configuration files. Combining these two effects, the first 31 characters are not always unique.</p>
Potential risks	A compiler might not correctly pre-process the software what might result in incorrect code.
Prevention of risks	<p>Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	-

Deviation ID	MD_MSR_5.1_779
Violated rule	5.1 (Identifiers (internal and external) shall not rely on the significance of more than 31 characters)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/5.1.A.1.
Reason	<p>The QA-C rule 0779 checks if all internal identifiers are different within the first 31 characters. This check is due to some older preprocessors compare just the first 31 characters.</p> <p>In complex software architectures like the AUTOSAR based MICROSAR, naming rules to prefix the identifiers are allocating a significant number of characters. Many of the identifiers are generated based on their elements in the OEM-defined configuration files. Combining these two effects, the first 31 characters are not always unique.</p>
Potential risks	A compiler might not correctly pre-process the software what might result in incorrect code.
Prevention of risks	<p>Each delivery is integrated and tested on the real target system. In addition, preprocessors and compilers used in current MICROSAR projects are not expected to suffer from this (historic) limitation.</p> <p>Compiler selection and validation is done by ECU supplier.</p>
Examples	-

Deviation ID	MD_MSR_5.6
Violated rule	5.6 (No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.)
Reason	<p>This MISRA rule addresses potential misunderstandings by using the very same identifier multiple times in a different context.</p> <p>MICROSAR deviates partly from rule 5.6 by allowing deviations in case „Standards“ like AUTOSAR, ISO/ASAM/... and OEM specifications require the same identifier names in different name spaces. Note, that this use case is the only accepted application of this deviation.</p>
Potential risks	Readability of the code is reduced by such ambiguities.
Prevention of risks	Code inspection and test.
Examples	<p>The identifier names “length” and “data” specified by AUTOSAR both for structure members and functions parameters lead to a violation of rule 5.6 if the related header files of these two modules are included.</p> <pre> [SWS_SoAd_00681]: Std_ReturnType SoAd_ReadDhcpHostNameOption(SoAd_SoConIdType SoConId, uint8* length, uint8* data) [SWS_Csm_00076] typedef struct { uint32 length; Csm_AsymPublicKeyArrayType data; } Csm_AsymPublicKeyType; </pre>

Deviation ID	MD_MSR_5.7
Violated rule	5.7 (No identifier name should be reused)
Reason	<p>This MISRA rule addresses potential misunderstandings by using the very same name multiple times with different meanings in different context. In general, this is addressed by naming conventions for external and globally visible identifiers like typedef, functions, global data and similar and explicitly addressed by rules 5.2 to 5.6.</p> <p>MICROSAR deviates partly to rule 5.7 by allowing the reuse of parameter names function-local variable names</p> <p>These specific MISRA deviations are not annotated in the code because the rule is checked by review and it can on module level not decided, if any other module in the later project may use the same parameter name or not. A general deviation for each API and local function reduces the code readability on important topics.</p> <div>  <p>Note 1 Complex software architectures like AUTOSAR define the function prototypes including the name of the parameter. The name of the API is compliant to rule 5.7 but the used parameter name is often the same for different APIs.</p> </div> <div>  <p>Note 2 Enforcing a unique parameter and function-local variable name code style rule implies that the function name has to be part of the resulting identifier name. As the function's name has already been build based on e.g. a module-prefix and other constraints, the resulting identifier name will be very long and thus unreadable and error-prone.</p> </div> <div>  <p>Note 3 This procedure follows the spirit of MISRA-C:2012 where this rule has been relaxed.</p> </div>
Potential risks	MISRA addresses an absolute unique definition of all identifiers in a software system to reduce the risk of erroneous usage. This risk is sufficiently addressed by applying rule 5.7 to global and external identifiers and keep local identifiers "readable".
Prevention of risks	<p>Compliance to rules 5.2 – 5.6 and AUTOSAR naming convention prevents that a variable is unexpectedly misused.</p> <p>Applying a strict development process with independent module tests shows that each module is complete and independent to each other, so unexpected misuse by e.g. writing error of a parameter or local variable name is detectable.</p>
Examples	<pre>void <BSW_NAME_1>_GetVersionInfo (Std_VersionInfoType* { for(uint8 =0; i<10; i++) { } } void <BSW_NAME_2>_GetVersionInfo (Std_VersionInfoType*)</pre>

Deviation ID	MD_MSR_5.7
	<pre> { for(uint8 =0; i<10; i++) { } } </pre>

Deviation ID	MD_MSR_8.10
Violated rule	8.10 (global symbol only referenced in one translation unit)
Reason	<p>Since the actual usage of the software component by the customer is not known some of the provided interfaces may not be used. Also during MISRA analysis there may be some provided APIs which are not in use.</p> <p>If there is only one configuration it is possible to reference such symbols in a test module. In case of large test suites with many configurations this is not feasible.</p>
Potential risks	<p>Functions or global variables which are superfluous in a specific configuration may remain undetected which leads to a small resource overhead.</p> <p>Other software parts in the application could access symbols which should only be used internally.</p>
Prevention of risks	By code inspection of the different variants.
Examples	-

Deviation ID	MD_MSR_14.1
Violated rule	14.1 (unreachable code - API function not used in project)
	[MISRA-P1] addresses this topic in Permit/MISRA/C:2004/14.1.A.1.
Reason	<p>Since the actual usage of the software component by the customer is not known some of the provided interfaces may not be used. Also during MISRA analysis there may be some provided APIs which are not in use.</p> <p>If there is only one configuration it is possible to reference such symbols in a test module. In case of large test suites with many configurations this is not feasible.</p>
Potential risks	<p>Functions or global variables which are superfluous in a specific configuration may remain undetected which leads to a small resource overhead.</p> <p>Other software parts in the application could access symbols which should only be used internally.</p>
Prevention of risks	By code inspection of the different variants.
Examples	-

Deviation ID	MD_MSR_14.2
Violated rule	14.2 (All non-null statements shall either (i) have at least one side-effect however executed, or (ii) cause control flow to change)
Reason	To support different kinds of dummy statements, a dummy statement is encapsulated in a macro like MSN_DUMMY_STATEMENT(x). If no dummy statement is required, the macro is empty and only a null-statement remains.
Potential risks	None, because in case of a dummy statement it is expected that there is no side effect.
Prevention of risks	Template based usage restricted to dummy statements.
Examples	<code>MSN_DUMMY_STATEMENT(errorId); /* PRQA S 3112, 3199 */ /* MD_MSR_14.2 */</code>

Deviation ID	MD_MSR_14.3
Violated rule	14.3 (empty statement must appear on a line on its own)
Reason	If macros are used in different code variants it may happen that after resolving the macro only a null statement remains. This happens typically if assertions are implemented by macros.
Potential risks	Wrong macro implementations could remain undetected.
Prevention of risks	Code inspection of the macros and test of different source code variants by the component test suites.
Examples	Assertions: <pre>#if defined(IL_ENABLE_SYS_TESTDEBUG) #define IlAssert(p,e) if ((p)==0) { (ApplIlFatalError((e))); } #else #define IlAssert(a,b) #endif /* if assertions are disabled the following line leads to a ";" after preprocessing but there are other characters in the source line */ IlAssert(idx <= 0x80, ILERR_ILLTIMERMASK);</pre>

Deviation ID	MD_MSR_14.7
Violated rule	14.7 (single point of exit at end of function)
Reason	<p>Note: Vector development "Process 3" compliant software has to comply to rule 14.7.</p> <p>A single exit point may lead to an inefficient or more difficult to understand code because the structure of the function can become more complex. Therefore it may be necessary to violate this rule for runtime and maintenance efficiency reasons.</p>
Potential risks	Function may e.g. terminate with locked interrupts. Control and data flow have multiple paths increased risk of adding errors during maintenance activities.
Prevention of risks	<p>Product design defines two "blocks" in each function, one block is "checking/validity action like DET checks", and another block is "functional behavior". "Checking/validity actions" are allowed to terminate immediately resulting in more than one exit in the function. "Functional behavior" checks are expected to terminate at the end of the function in one single exit.</p> <p>Based on this design rule, a justification is necessary for each deviating function.</p>
Examples	-

Deviation ID	MD_MSR_16.7
Violated rule	16.7 (A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object)
Reason	Many standards like AUTOSAR, ISO, but also OEM specifications ... specify the function prototypes without const statement. We have to comply with these standards.
Potential risks	Less precision in describing access rights on interface level may lead to incorrect use of the parameters.
Prevention of risks	Standards describe allowed parameter use and code style guide describes how to document parameter usage in the code to reduce risk of wrong usage.
Examples	-

([GUNV

Deviation ID	MD_MSR_18.4
Violated rule	18.4 (packing and unpacking of data, variant records)
Reason	For an efficient implementation of protocols the usage of unions is required. This is covered by the two acceptable deviations described in [MISRA-C]
Potential risks	Union data may be misinterpreted by mixing up the different variants.
Prevention of risks	Code inspection and test of the different variants in the component test.

Deviation ID	MD_MSR_19.1
Violated rule	19.1 (#include statements preceded)
Reason	AUTOSAR SWS Memory Mapping requires inclusion of MemMap.h multiple times in a file in order to select appropriate #pragma directives.
Potential risks	MemMap.h is provided by the ECU software integrator, hence many risks may occur, caused by wrong implementation of this file.
Prevention of risks	The ECU software integrator strictly has to adhere to the definitions of the AUTOSAR SWS Memory Mapping. Extensions to the file not described in the SWS may not be put into MemMap.h. This has to be verified by code inspection.
Examples	<pre>#define ECUM_STOP_SEC_CODE #include "MemMap.h"</pre>

Deviation ID	MD_MSR_19.4
Violated rule	19.4 (reserved C keywords or braces / parentheses in macro)
Reason	<p>Since efficiency is a primary implementation target it is necessary to use macros.</p> <p>Note that the proposed do-while-zero construct in [MISRA-C] leads to “useless statement” warnings with many compilers.</p>
Potential risks	Resulting code is difficult to understand or may not work as expected.
Prevention of risks	Code inspection and test of the different variants in the component test.
Examples	<pre>#define FRNM_CHECK_INIT(ApiId) \ { \ if(FrNm_NmState[FRNM_CHANNEL_IDX] == NM_STATE_UNINIT) \ { \ (void)Det_ReportError(FRNM_MODULE_ID, FRNM_INSTANCE_ID, \ (ApiId), FRNM_E_UNINIT); \ } \ } \ #endif</pre>

Deviation ID	MD_MSR_19.6
Violated rule	19.6 (#undef shall not be used)
Reason	AUTOSAR SWS Memory Mapping requires #undef statements in order to implement the #pragma selection mechanism in MemMap.h
Potential risks	MemMap.h is provided by the integrator; hence many risks may occur, caused by wrong implementation of this file.
Prevention of risks	The integrator strictly has to adhere to the definitions of the AUTOSAR SWS Memory Mapping. Extensions to the file not described in the SWS may not be put into MemMap.h. This has to be verified by code inspection.
Examples	<pre>#ifdef START_SEC_CONST_32BIT #pragma section MyConst32BitSection #undef START_SEC_CONST_32BIT #undef MEMMAP_ERROR #endif</pre>

Deviation ID	MD_MSR_19.7
Violated rule	19.7 (function should be used instead of macro)
Reason	Since efficiency is a primary implementation target it is necessary to use macros.
Potential risks	Resulting code is difficult to understand or may not work as expected.
Prevention of risks	Code inspection and test of the different variants in the component test.
Examples	<pre>#define Com_IsDirectOfTxModeTrue (Index) (COMDIRECTOFTXMODETRUE_MASK == (Com_GetMaskedBitsOfTxModeTrue (Index) & COMDIRECTOFTXMODETRUE_MASK))</pre>

Deviation ID	MD_MSR_19.8
Violated rule	19.8 (A function-like macro shall not be invoked without all of its arguments)
Reason	AUTOSAR SWS Compiler Abstraction allows configuration of standard addressing mode by #define'ing an empty compiler abstraction qualifier in Compiler_Cfg.h.
Potential risks	None.
Prevention of risks	None.
Examples	<p>In Compiler_Cfg.h:</p> <pre>#define EEP_CONST</pre> <p>In Eep.c (implementation of component):</p> <pre>CONST(uint8, EEP_CONST) Eep_Status;</pre>

Deviation ID	MD_MSR_19.13_0342
Violated rule	19.13 (The # and ## preprocessor operators should not be used)
Reason	“Rule 0342: K&R compilers do not support the ISO glue operator '##'.” AUTOSAR requires C90 and C90 supports ## operators.
Potential risks	None.
Prevention of risks	None.
Examples	–

4.2 Code Metric Justification for “Product Deviations”

This chapter contains the list of the agreed and released code metric Product Deviations.

Test reports references the Deviation ID, if applicable. If a deviation does not match the description in this chapter, it is handled similar to a Specific Deviation with a local justification.



Note

The QA-C error codes are defined by Vector and are no standard QA-C error codes.

Deviation ID	MD_MSR_STPTH
Violated metric	Estimated static path count defined by HIS shall be in range 1..80
QA-C error code	6010
Reason	No separation of functionality into sub-functions due to higher voted requirements for minimized stack and runtime usage applied on the code.
Potential risks	Understandability and testability might become too complex.
Prevention of risks	Design and code review + clearly structured and commented code.
Comments	Metric calculates the upper bound of possible (non-cyclic) paths in function's control flow. Metric addresses the TESTABILITY of a function – the higher the more test cases might be necessary.

Deviation ID	MD_MSR_STCYC
Violated metric	Cyclomatic complexity defined by HIS shall be in range 1..10
QA-C error code	6030
Reason	No separation of functionality into sub-functions due to higher voted requirements for minimized stack and runtime usage applied on the code.
Potential risks	Understandability and testability might become too complex.
Prevention of risks	Design and code review + clearly structured and commented code.
Comments	Metric indicates potentially inadequate modularization or too much logic in one function if the value is high, the function may tend to have complexity related issues. Metric addresses a HOTSPOT in the code.

Deviation ID	MD_MSR_STCAL
Violated metric	Number of distinct function calls defined by HIS shall be in range 0..7
QA-C error code	6050
Reason	Software structure is defined by AUTOSAR standard. Standard compliance is voted higher than metric threshold. In addition, a typical approach to reduce STCAL is deeper nesting of functions, this increases call stack usage and runtime.
Potential risks	Understandability and testability might become too complex due to fan-out to many functions.
Prevention of risks	Design and code review + clearly structured and commented code.
Comments	Metric indicates fan-out of a function, i.e. how many different functions are called. Metric addresses a HOTSPOT in the code. The fan-out depends on a mix of component-local design and on global design (AUTOSAR). Only component-local design could be adopted.

Deviation ID	MD_MSR_STPAR
Violated metric	Number of function parameters defined by HIS shall be in range 0..5
QA-C error code	6060
Reason	API is defined by AUTOSAR standard. Standard compliance is voted higher than metric threshold.
Potential risks	Stack usage and runtime too high for target uC.
Prevention of risks	Test of resulting code on target uC, user has to check stack usage in project context.
Comments	<p>Metric counts number of declared parameters in the function argument list what have impact on runtime and call stack usage.</p> <p>Vector is aware of such environmental conditions and applies larger numbers of function parameters only, if explicitly required by standards like AUTOSAR, ISO, customer, ... specs.</p>

Deviation ID	MD_MSR_STMIF
Violated metric	Number maximum nesting of control structures defined by HIS shall be in range 0..4
QA-C error code	6080
Reason	Function handles specific task in the overall component behavior; task has different scenarios to cover depending on local conditions – this results in deep nesting of control structures. Due to there is more common than different code, higher nesting level is accepted to keep code footprint small.
Potential risks	Code is difficult to maintain.
Prevention of risks	Design and code review + clearly structured and commented code.
Comments	Metric addresses the MAINTAINABILITY of the code. Expectation is, that deep nesting of control structures increases the effort to understand a function.

5 Additional Resources

No.	Document
[MISRA-C]	MISRA-C:2004 Guidelines for the use of the C language in critical systems (October 2004)
[MISRA-P1]	MISRA C 2004 Permits (First Edition).pdf (April 2016)
[HIS-MISRA]	Gemeinsames Subset der MISRA C Guidelines (Version 2.0, not public available)
[HIS-CODE]	HIS Source Code Metriken (Version 1.3.1, not public available)
[AUTOSAR]	http://www.autosar.org/

6 Contacts

For a full list with all Vector locations and addresses worldwide, please visit <http://vector.com/contact/>.