	CM-IS Coding Guidelines	Version, status V1.5, REL	
Owner CM-CI2	Coding Guidelines for C	Author Process Team Implementation	Date Aug 23, 2017

Abstract:
This document contains rules and recommendations regarding the programming Languange C, and it also contains the QA-C messages that are checked ensure the rules are respected.

Contacts
Hornecker Markus (CM-CI2/EVW2-E)
Markus.Hornecker@de.bosch.com

Maier, Christian (CM-CI2/ECF12)
Christian.Maier2@de.bosch.com

Happ, Peter (CM-CI2/ECF11)
Peter.Happ@de.bosch.com

Approved:
The rules and recomendation in this Coding Guidelines are valid from 23.08.2017

Signed/Released: V1.5 Process –Team Implementation, 23.08.2017

C Coding guidelines

inside.Docupedia Export

Autor:	Happ Peter (CM-CI2/ECF11)
Datum:	22.08.2017 13:07

Table of Contents

1 Change History

V0.1 2012-10-31 Moved to new format with wiki

v0.2 2013-12-06 Added new Blocks "Design Rules" and "Compiler Specific Behavior"

v0.3 2013-03-31 Added mapping to related QA-C Messages

v1.0 2014-06-06 Added Rules for handling compiler warnings in Block "Design Rules" and structure handling in Block "Compiler Specific Behavior"

v1.3 2017-02-17 Spaces shall now be used, tabs are not allowed anymore

v1.4 2017-07-06 Naming Convention changes for clarification

v1.5 2017-08-23 Update Link to "Rules for prevention of SW-errors"

2 References

Index	Particular	Link
1	ANSI C	iso-iec9899.pdf
2	QA-C Personalities (includes MISRA 2004 ruleset)	In RTC: /UnitTest/Tool.QA-C.Personalities
3	Doxygen installation and tool	In RTC: /DevelopmentEnvironmentExternalTools/Tool.Doxygen_v1.7
4	Templates for doxygen, which are then used as code documentation:	Standard Coding Templates
5	Old version of the Coding_and_Modeling_Guidelines (is relevant only for projects with ASCET-use)	CC\SoftwareProcessImprovement\Handbuch\P04_Implementierung\Codierungs_und_Modell doc \main\23
6	Rules for prevention of SW-errors	Rules_for_Avoiding_Software_Failures_En_final.ppt
7	Additional coding guidelines to cover Design principles for software unit design and implementation for ASIL-relevant modules	ASIL_Guidlines.xlsx
8	Mapping the modeling and coding guidelines in QA-C	Mapping_CodingGuidelines.xls CM-IS C Coding Standards to QAC rules mapping ✓

3 Abbreviations

Abbreviation	**Description**
R	Rule; this guideline is to be followed
E	This recommendation is meaningful, but not mandatory
Mnnn	Additional rules, additional to MISRA

4 OPL

Open point	Determined by
.	

5 Introduction, Objective

1. The following coding guidelines refer to the creation of C-code.
2. The objective of this guideline is to model the software development compliant with the market standards ANSI C [01] and MISRA [02]. The independence of current compilers is to be assured only in this manner, which, under certain circumstances, works only conditionally conforming to ANSI-C.
3. Complying with these guidelines simplifies the reuse (e.g. of code, ASCET models) in new projects and porting on different operating systems with diverse compilers.
4. Another prime objective is the legibility and maintainability to enable the colleagues to carry out further work or review on documents or code.
5. The module documentation is based on the result of the doxygen analysis. A standardized code documentation is required for an in-depth use of the report.
6. Other instructions are mainly incorporated for the purpose of error prevention [06].

6 Guidelines

Index	Guideline	Link
1	Naming conventions, Macros and Constants	C Coding Guidelines - Naming Convention
2	Data types	C Coding Guidelines - Data Types
3	Blocks and Functions, Libraries	C Coding Guidelines - Blocks and Functions
4	Instructions, Outputs and Operators	C Coding Guidelines - Instructions, Outputs and Operators
5	Design Rules (Interrupts, Copy right etc.)	C Coding Guidelines - Design Rules
6	Compiler-Specific Behavior	C Coding Guidelines - Compiler-Specific Behavior

7 General Information and Language

1. Emerging documents, code, models, etc. are to be composed in English. This pertains to, for example, names of classes, methods or even comments in the code. Should this not be possible in all the situations, the code fragment is to be described correspondingly with explanations.
2. The templates given under [02] are to be used for new code files.

8 Deviations from the Guidelines

Rule 0

All deviations from the rules specified here must be justified, documented and approved by the PT implementation, as long as they are not allowed as exceptions. A few rules are *Recommendations* (such are identified as explicit!) and should be applied as far as possible.

Exceptions to rule 0: No exception.

The rules and recommendations specified in this programming convention can be upgraded specific to the project. Recommendations can also become rules. Basically, it should be ensured that enhancements and changes do not violate these programming conventions.

The following sequence is applicable here:

1. These guidelines
2. The project manual, which can define the further rules/exceptions
3. The justified deviation documentation directly in the source code of the module.

Rule 1

Deviations from rules for SW generated with tool support are to be documented. The use of the appropriate tool is to be justified.

Exceptions to rule 1: No exception.

The document also governs the appearance of software generated with tool support. If the tool cannot support the rules prescribed in this document, a valid reason is required as to why the tool is still being used. The deviations are to be documented.

Rule 2

Rule deviations are additionally permitted if smaller modifications are carried out in the existing code. The pragmatic approach for this exception is to continue the existing code with the guidelines applied (provided these can be identified). The deviations are to be documented.

Exceptions to rule 2: No exception.

9 References to the Document

Rule 3

References to rules of this document must always have the version number of the document.

Exceptions to rule 3: No exception.

Changes in the document can lead to changes in the numbering of the rules and recommendations.

10 C Coding Guidelines - Naming Convention

10.1 Abbreviations

Abbreviation	Description
R	Rule; this guideline is to be followed
E	This recommendation is meaningful, but not mandatory
Mnnn	Additional rules, additional to MISRA

10.2 Type of suffixes

Rule 1

The following endings are to be used:

data type	suffixes
unsigned integer 8 Bit	_uint8
unsigned integer 16 Bit	_uint16
unsigned integer 32 Bit	_uint32
unsigned integer 64 Bit	_uint64
signed integer 8 Bit	_sint8
signed integer 16 Bit	_sint16
signed integer 32 Bit	_sint32
signed integer 64 Bit	_sint64
char	_c
Pointer	_p
Structure	_s
Union	_u
Array	_a
Bit/Boolean	_b
Enum	_e
Separate type defs	_t

10.3 Module names

Rule 2

All software module names should be identifiable by a unique name.

The module name must be self-explanatory and must be selected with regard to the function of the module.

The first letter of each word in the module name is to be written in the upper case.

The subsequent alphabets are to be written in the small case.

Number sequences are not allowed

Example

ActuatorTest
VehicleStateManager
CarOutline

10.4 File names

Rule 3

All files which belong to a C-module carry the identifier of the module. <Module name>_<Addition>.<Extension>
Addition -> [a..z]
Extension -> c|h

For a better description of what type of Additions are allowed and which functionality should be in this Files, please look at the standard Templates: Standard Coding Templates

10.5 Function name

Rule 4

An identifier should not be longer than 31 characters. Therefore, under certain circumstances, it should be followed by a meaningful but self-explanatory abbreviation of the module name or function name. A possible abbreviation of the module name should then be continued to be used in all the local function names.

Public (interfaces) functions have the following naming convention: <Module name>_<Function name>()

Example: CarOutline_GetComponentId()

The <Function name> begins with a capital letter in such a case and no underlines are used, but rather words are separated using capital letters.

Local (module-internal) functions are subject to the same specifications as the public functions; however, they begin with the abbreviation of the module *written in small letters*.
For local functions, the abbreviation of the module name is left to the discretion of the developer provided it does not deteriorate the legibility and does not violate other rules. A) is to be complied with here.

There is only one space between <Module name> and <Function name>.

Example: caroutline_GetCounterX()

10.6 Variable names

5	R	Private RAM- and ROM data of a module are retained in a structure.
		This is defined as static and is consistently:
		<modul name>_this
		Example: TANK_this
		
		Mixed use of small and capital letters is allowed in variable names "varName", the first letter is however always a *small letter*.
		The Data type is represented in the suffix.
		An identifier should not be longer than 31 characters.
		
		Naming convention: <varName>_<typ>Example: myCounter_uint08.
		Exceptions within the functions are the customary counting variables i, j, k (Rule no.: 102)
		An identifier should not have the same name in different name spaces, with the exception of structure and union member names.
		
		`Example :`
		`struct { int16_t key; int16_t value; } record;`
		
		`int16_t value;                \* Rule violation - 2nd use of value \*`
		
		`struct { int16_t key; int16_t value; } record1; /*no violation*/`

10.7 Use of Defines and Macros

10.7.1 Macros

Rule 6

Macros begin with the name of the module followed by an underline. Only capital letters are allowed in the module abbreviation. Only capital letters are allowed even in the macro names. A macro is *always* completed with "()".
Naming convention: <module name>_<name>()

Example: *TANK_DO_XY()*

10.7.2 Constants

Rule 7

Constants begin with the abbreviation of the module followed by an underline. Only capital letters are allowed in the abbreviation "ModuleName" and in the name of the constants.
Naming convention:<name of the module>_<name of the constant>

Example: TANK_NUMBER_OF_SENSORS

No variables are allowed as value of the constants - please use macros or functions in such cases.

```
Example:
Not so!:     #define TANK_MY_VALUE_X (tnk_i_uint8 * 10)
But rather:  #define TANK_GET_MY_VALUE_X() (tnk_i_uint8 * 10)
Or so:       uint8 tank_GetMyValueX() { return (tnk_i_uint8 * 1
0); }
```

Rule 8

The name of the constant must explain the usage and source of the value to avoid 'magic' numbers. If that explanation would be too long write a comment next to the declaration.

```
Example:
Not so!:     #define LED_FIVE 5
But rather:  #define LED_NUMBER_OF_LED_LEFT_SIDE 5
```

11 C Coding Guidelines - Data Types

11.1 Abbreviations

Abbreviation	**Description**
R	Rule; this guideline is to be followed
E	This recommendation is meaningful, but not mandatory
Mnnn	Additional rules, additional to MISRA

11.2 General information

Index	Type	Rule	QAC Rule
8	R	Basic types are not used directly in the code, but are rather determined globally via typedef, *e.g.* via "typedef unsigned short ui16". These are defined in Platform_Types and incorporated via Std_Types.h.	No matching rule found
9	R	Names of type identifiers (typedef) are not used for other purposes. This means that they cannot be re-defined.	1507, 4605
10	R	Object-IDs and function-IDs are declared before use (*e.g.* prototypes).	3450, 541, 544, 545, 620, 621, 3313

Index	Type	Rule	QAC Rule

11	R	Every object (*e.g.* of a structure) should be used corresponding to its declaration, e.g.
		• Prohibited: Allocation of a ui32 to an address, from which four characters should begin:
		`struct xyz_msg`
		`{ uint8 a;   uint8 b;   uint8 c;   uint8`
		`d;} tnk_xyz_msg1_s;`
		`tnk_xyz_msg1_s = 1234L;   // NOT SO!`
		• Enumerations use only the names that are defined for them:
		`enum fruit { FRUIT_APPLE = 1, FRUIT_PEARS};`
		`enum fruit myFruit_e = 1; // prohibited`
		`enum fruit myFruit_e = FRUIT_APPLE; // good`
		`enum car { CAR_VW, CAR_AUDI };`
		`enum car myCar_e = FRUIT_PEARS; //`
		`prohibited`
			561, 1402, 1412

12	R	Values, which do not (are not allowed to) change, are also defined as constant. *Example*: Function parameters that are only to be read and which are transferred "by reference", are qualified as const for their protection, *e.g.* void xyz_Fkt01_v(const si16 * const param) ... *Remark*: The pointer and the value are constant here.	3227
13	R	Initialization of variables before their use. An initial storage zeroes is not assured depending on the start up routine that is used!	2961, 2962, 2963
104	R	Externally referenced global variables may be used as read only.	No matching rule found

11.3 Visibility

Index	Type	Rule	QAC Rule
14	R	No larger namespace for objects than necessary, *e.g.* better local instead of global.	1500, 1502, 1504, 3229
15	R	Definitions valid across files should as far as possible be static -> visible only in this file	No matching rule found
16	R	Prohibition of simultaneous internal and external links, *e.g.* static ui08 x_ui08;/* internally linked */ void xyz_Foo_v(void) { external ui08 x_ui08; /* externally linked */}	3224
17	E	Local variables with appropriate access functions should be defined instead of global variables. This improves the encapsulation and the overview.	No matching rule found
100	R	Prohibition of same variable names within and outside a statement block static ui08 x_ui08;/* outside the block */ void xyz_Foo_v(void) { ui08 x_ui08; /* within the block */}	No matching rule found

11.4 Enums

Index	Type	Rule	QAC Rule
18	R	None, only the first value or all values should be initialized in case of enum. *Prohibited*: enum { FRUIT_APPLE, FRUIT_ORANGE = 10, FRUIT_LEMON /* 0, 10, 11 */ }; good enum { FRUIT_APPLE = 0, FRUIT_ORANGE = 10, FRUIT_LEMON = 11 }; super: enum { FRUIT_APPLE = 1, FRUIT_ORANGE, FRUIT_LEMON /* 1, 2, 3 */}; -> Also see rule 16	723
19	R	The names of defined constants (#define) and enumeration types (enum) are assigned a unique prefix as described in 10.2. Upper case/lower case is also according to 10.2) Examples : #define SMAA_TURNMODE_CLOCK 1 #define SMAA_TURNMODE_COUNTERCLOCK 3 enum SMAA_ POINTERINDEX { SMAA_POINTERINDEX_UNDEF =0 SMAA_POINTERINDEX_TACHO, SMAA_POINTERINDEX_SPEED, SMAA_POINTERINDEX_FUEL, SMAA_POINTERINDEX_TEMP };	No matching rule found
20	E	Indices of enums start with one (not zero) -> less prone to error, *e.g.* enum fruit { FRUIT_UNDEFINED, FRUIT_APPLE, FRUIT_ORANGE /* 0, 1, 2 */}; Thus, zero can be used as "undefined status".	No matching rule found

11.5 Structures

Index	Type	Rule	QAC Rule
21	R	Curved brackets for clarifying the structure of arrays/structures: prohibited si16 y_si16[2][2] = { 1, 2, 3, 4 }; correct: si16 y_si16[2][2] = { **{** 1, 2 **}**, **{** 3, 4 **}** };	693, 694

22	R	All the members are completely specified in case of a struct or a union, i.e. the number of bits always results in a complete byte:
		Correct:
		struct message
		{
		uint8 little: 4;
		uint8 x_set : 1;
		uint8 Reserved: 3;
		} myMessage_s;
		
		Prohibited:
		struct message
		{
		uint8 little: 4;
		uint8 x_set : 1;
		} myMessage_s;
			No matching rule found

Index	Type	Rule	QAC Rule

23	R	All members of a struct or a union are assigned a name, and they are accessed only by means of this name.
		struct message
		{ /* only for bit fields */
		uint8 little: 4;
		uint8 x_set : 1;
		uint8 y_set : 1;
		uint8 Reserved: 2;
		} myMessage_s;
		myMessage_s.little = 0xD; /* okay */
		myMessage_s = 0x3F; /* prohibited */
			No matching rule found

11.6 Unions

Index	Type	Rule	QAC Rule

22	R	All the members are completely specified in case of a struct or a union, i.e. the number of bits always results in a complete byte:
		Correct:
		union message
		{
		uint8 little: 4;
		uint8 x_set : 1;
		uint8 Reserved;
		} myMessage_u;
		
		Prohibited:
		union message
		{
		uint8 little: 4;
		uint8 x_set : 1;
		} myMessage_u;
			No matching rule found

24	R	unions are not misused in order to access parts of a big file type, better: via Bitmasks	No matching rule found

23	R	All members of a struct or a union get a name, and they are accessed using this name only.
		union message
		{
		uint8 all;
		uint8 little: 4;
		uint8 x_set : 1;
		uint8 y_set : 1;
		} myMessage_u;
		myMessage_s.all = 0xD; /* okay */
		myMessage_s = 0x3F; /* prohibited */
			660

101	R	Same identifier not allowed for structures and unions.
		struct record { int number;};
		union record decimal;
			547

11.7 Pointer

Index	Type	Rule	QAC Rule
25	R	No casts of any other type from or after a pointer, e.g. wrong: ui32 x = y_p; *Exception:* Access to HW addresses	303, 305, 306, 308, 310, 313, 314, 316, 317
26	R	No use of pointer arithmetic, except for navigation in arrays or similar.	488
27	R	Maximum two levels of pointer indirection, i.e. maximum two "asterix"	3260, 3261, 3262, 3263

11.8 Typecasts

Index	Type	Rule	QAC Rule
28	R	Do not use unary minus on unsigned-types, For example, *Prohibited*: x = -my_ uint8 *Correct*: x = 0 – my_uint8	3101, 3102
29	R	Avoid implicit casts with possible loss of information, *e.g.* bad: my_ uint8 = my_sint16 – 5;	2900, 2901, 2902, 2903, 3733, 3734, 3744, 3746, 3755, 3756, 3758, 3766, 3768, 3770, 3777, 3778, 3780, 3782, 3788, 3790, 3794, 3799, 3800, 3801, 3802, 3803, 3804, 3805, 3806, 3807, 3810, 3811, 3812, 3813, 3814, 3815, 3816, 3817, 3818, 3819, 3821, 3822, 3823, 3824, 3825, 3826, 3827, 3828, 3829, 3830, 3831, 3850, 3851, 3853, 3855, 3857, 3863, 3864, 3865, 3867, 3869, 3871, 3876, 3877, 3878, 3879, 3880, 3881, 3933, 3934, 3944, 3946, 3955, 3956, 3957, 3958, 3966, 3967, 3968, 3970, 3977, 3978, 3980, 3982, 3988, 3990, 3992, 3994, 3999, 4000, 4001, 4002, 4003, 4004, 4005, 4006, 4007, 4010, 4011, 4012, 4013, 4014, 4015, 4016, 4017, 4018, 4019, 4021, 4022, 4023, 4024, 4025, 4026, 4027, 4028, 4029, 4030, 4031, 4050, 4051, 4053, 4055, 4057, 4063, 4065, 4067, 4069, 4071, 4076, 4077, 4078, 4079, 4080, 4081, 3779, 3781, 3783, 3789, 3791, 3792, 3793, 3797, 3848, 3852, 3854, 3856, 3858, 3860, 3866, 3868, 3870, 3873, 3935, 3945, 3959, 3969, 3979, 3981, 3983, 3989, 3991, 3993, 4052, 4054, 4056, 4058, 4060, 4064, 4066, 4068, 4070, 4073, 4074
30	R	Avoid unnecessary casts, *e.g.* bad: x_uint32 = (uint8)my_ uint8;	3212, 4121, 4126, 4127, 4128
25	R	No casts of any other type from or after a pointer, e.g. wrong: uint32 x = y_p; *Exception:* Access to HW addresses	No matching rule found
31	E	Explicit casts for operations with different precision, *e. g.* uint16 a_uint16 = 65535; ui16 b_uint16 = 10; uint32 e1_uint32 = a_uint16 + b_uint16;/* wrong!!! -> 9 */ uint32 e2_uint32 = (ui32) a_uint16 + b_uint16; /* correct -> 65545 */	No matching rule found

11.9 Miscellaneous

Index	Type	Rule	QAC Rule
32	R	Bit field elements of type signed int are at least two bits long	3660
33	E	Prevention of the register-storage class -> the compiler is considerably efficient.	2011

12 C Coding Guidelines - Blocks and Functions

12.1 Abbreviations

Abbreviation	**Description**
R	Rule; this guideline is to be followed
E	This recommendation is meaningful, but not mandatory
Mnnn	Additional rules, additional to MISRA

12.2 Declaration

Index	Type	Rule	QAC Rule
64	R	All identifiers (variables, constants, defines, functions,…) are declared before use with corresponding C-syntax. (e.g. functions as prototypes)	1335, 1336
65	R	Each function has an associated prototype, which is visible "from both sides" (for definition & call) -> Prototype in the header, which is incorporated in all the files being used via include	No matching rule found
66	R	Data types and names of the parameters and return values are identical and not contradictory in case of declaration (prototype) and definition (implementation).	1331, 1332, 1333
67	E	The naming convention takes place according to the specifications in chapter 9.	No matching rule found
68	R	Names of standard LIB functions are not used again (i.e. a new function, which improves the previous one, also gets a new name, e.g. Abs2)	No matching rule found

12.3 Definition

Index	Type	Rule	QAC Rule

69	R	Functions are always defined at the file level (not in a block!).
		Reasons: Overview, visibility
		`+Thus,+ \\`
		`File \\`
		`\-> Function definition \\`
			No matching rule found

Index	Type	Rule	QAC Rule
		```
		\-> Function definition \\
		\-> … \\
		\\
		+and not:+ \\
		File \\
		\-> Function definition \\
		\{ \\
		Function (as is always possible) \\
		\} \\
		\-> Function definition
		```
70	R	Function definitions (implementation) in header files are prohibited.	No matching rule found
71	R	Functions do not call themselves (even indirectly) – "recursive functions". Reason: Error prevention, stack overflow	3670, 1520

72	R	Only use spaces, and indent **4** spaces at a time.
		Hint: You can configure your Editor to replace tabs with spaces.
		```
		bad:
		void XYZ_DoBla_v(ui08 b_ui08)
		{
		 ui08 a_ui08 = 0;
		 if (1 == b_ui08)
		 {
		  xyz_Bla2_v();
		  a_ui08++;
		 }
		}
		```
		```
		good:
		void XYZ_DoBla_v(ui08 b_ui08)
		{
		    ui08 a_ui08 = 0;
		    if (1 == b_ui08)
		    {
		        xyz_Bla2_v();
		        a_ui08++;
		    }
		}
		```
			No matching rule found

73	R	Setting brackets in case of function blocks - always in the front, not indented.
		```
		e.g.:
		void xyz_DoBla_v(void)
		*\{*
		```
			No matching rule found

Index	Type	Rule	QAC Rule
		```
		...
		if (1 == xyz_isRunning_b)
		*{*
		...
		*}*
		*}*
		```

74	R	No identical identifiers in the nested namespace,
		```
		e.g.:
		*ui08 a_ui08* = 34;
		for (...)
		{
		*ui32 a_ui08*;
		a_ui08 = 123;
		...
		}
		```
			619, 780, 782

12.4 Parameter

Index	Type	Rule	QAC Rule
75	R	The parameter names are specified in the prototype for all parameters.	1335, 1336
76	R	Functions without any parameters are defined with the parameter type void, and declared, e.g.: ui08 xyz_FunctionA_ui08(**void**) { ... }	3007
77	R	The number and type of parameters while calling the functions are appropriate for the prototype. Implicit type conversion (e.g. signed integer to unsigned integer) is not allowed. Reason: Prevention of errors due to overrun or undefined behavior during the conversion.	422, 423, 3319, 3320

78	R	"Unqualified" transfer parameter types must correspond to the prototype declaration during the call, e.g. overwriting that is not permitted:
		```
		*const* ui08 *a_ui08 = 1*;
		void xyz_Foo_v(ui08 *b_ui08)
		{
		*b_ui08 = 5;
		}
		xyz_Foo_v(&a_ui08);
		```
			429, 430, 431, 432

79	R	No void-parameters while calling a function,
		```
		e.g.:
		ui08 xyz_MyFunction_ui08(void)
		```
			No matching rule found

Index	Type	Rule	QAC Rule
		`{      ...    }   xy_ui08 = xyz_MyFunction_ui08(*-void-*);`	
80	R	The validity of the values, which are transferred to LIB functions, must be examined. e.g.: Check for exceeding the limit value, zero-pointer,…	No matching rule found

12.5 Return values

Index	Type	Rule	QAC Rule
81	R	Every function explicitly has a return type, e.g. **void** xyz_Foo_v(int a) { ... }instead of only xyz_Foo_v(ui16 a_ui16) /* implicit integer!! */{ ... }	2050
82	E	Every function has only one output (return at the end of the function) Origin: MISRA rule 82	2889
83	R	All functions with an authentic return value - should define a return-allocation - indicate a value for return with a valid type (such as prototype) Refer rule E82	2888, 745, 1325, 755, 756, 757, 758
84	R	void-functions do not specify any return value. Thus, it is correct: Omit the return Refer rule E82	746, 747
85	R	Return values or return parameters, which (could) present an error state, must be queried and as far as possible, an appropriate error processing should be executed (even for global error variables that are to be avoided).	No matching rule found

12.6 Libraries

Index	Type	Rule	QAC Rule
80	R	The validity of the values, which are transferred to LIB functions, must be examined.	No matching rule found
86	R	The error variable error of the standard C-library is not used,	No matching rule found
87	R	The macro offsetof the LIB stddef.h is not used.	No matching rule found
88	R	The I/O-LIB stdio.h is not used for releases.	No matching rule found
89	R	The functions atof, atoi and atol of the LIB stdlib.h are not used (due to lack of error handling).	No matching rule found
90	R	The functions abort, exit, getenv and system of the LIB stdlib.h are not used.	No matching rule found
91	R	The time handling functions of the LIB time.h are not used for releases.	No matching rule found
82	R	Reserved words and names of standard LIB functions are not redefined / overwritten or discarded (#undef).	602, 4607, 4623
68	R		

Index	Type	Rule	QAC Rule
		Names of standard LIB functions are not used again (i.e. a new function, which improves the previous one, also gets a new name, e.g. Abs2)	No matching rule found
93	R	The functions setjmp and longjmp of the library setjmp.h should not be used.	No matching rule found
99	R	The return value must be checked during the dynamic memory allocation. The memory may be used only in case of successful memory allocation.	No matching rule found

13 C Coding Guidelines - Instructions, Outputs and Operators

13.1 Abbreviations

Abbreviation	Description
R	Rule; this guideline is to be followed
E	This recommendation is meaningful, but not mandatory
Mnnn	Additional rules, additional to MISRA

13.2 Instructions, Outputs and Operators

Index	Type	Rule	QAC Rule

34	R	Encapsulated functions: Only C-code or only assembler-code, but not mixed.
		```
		_Wrong:_
		void xyz_Do01_v(void)/\*
		mixed -- prohibited \*/
		{
		....
		asm("......");
		}
		```
		```
		correct:
		/\* ASM - set prio level \*/
		void xyz_SetPL_v
		(newPrio)       /\* only
		ASM code \\*/\\{
		asm
		{
		%  reg newPrio;
		...
		movw newPrio, %r0
		%r0
		```
			3006

Index	Type	Rule	QAC Rule
		```
		}
		}
		void xyz_Do01_v(void)
		/* only C-code */
		{
		....
		xyz_SetPL_v(myPrio);
		}
		```
35	R	Use characters only according to ISO-C, valid characters are: • Alphabets A .. Z a .. z and _ • Numbers0 .. 9 • Space • Control characterEnd of line/page break (enter), horizontal & vertical tab • Special characters( ) [ ] { } < > + - * / \ % ^ ~ & Allowed for identifier: Alphabets and numbers, first character must be an alphabet.	No matching rule found
36	R	Multi byte characters and wide string literals are not used, *e.g.* x = L*"**Fred**"*;	No matching rule found
37	R	No nested C-comments (/* ... /* ... */ ... */). Single-line comments are possible in two ways: /* something simple */ Or meaningful for "Sub-titles" (always has identical width): /---- **public methods** --/ Multi-line comments begin from the second line with at least one asterix: /* something to say • more information */ or /*** • something to say * • more information * ***/ or meaningful as "Titles" for classification: /*== ======= METHODS ======= ==*/ Method headers have the following structure (for doxygen): /-- / /* FUNCTION: <function name> */ /*! @brief <short description, ...>	3108

Index	Type	Rule	QAC Rule
		• and so on ... **//** --*/ The comments for file headers are somewhat more complex and should be referred to in the templates [4]. Only examples can be taken in the templates. The information in the fields with big/small brackets (<>) must be replaced by the correct information while filling the templates.	
38	R	The operands && || and ?: should not involve any side effects: bad: if (isHigh_b || (x == i++)) ...	536, 537
39	R	The operands of && und || should be primary expressions (= IDs, constants or a bracketed expression), e.g. if ((a_ui08 == 0) && isHigh_b) ... or still better: if ((0 == a_uint8) ... • Compiler “complains” only in case of a “=”.	558, 559
40	R	No allocations in expressions with a logical return value: Not so: if (x_ui08 = y_ui08) ...	No matching rule found
41	R	Apply bit-wise operators (~ << >> & ^ |) only in unsigned int.	4130
42	R	No titles, e.g. Error: my_ui16 >> 16	No matching rule found
43	R	Do not apply sizeof on operations with side effects, e.g. bad: sizeof(++i) -> increment is not executed!	No matching rule found
44	R	Examine and note integer-Division implementation of the compiler: -5/3 = -1 Rest --2 (better) or --5/3 = -2 Rest +1	No matching rule found
30	R	Avoid unnecessary casts, e.g. bad: x_ui32 = (ui08)my_ui08;	No matching rule found
25	R	No casts of any other type from or after a pointer, e.g. wrong: ui32 x = y_p; Exception: Access to HW addresses	No matching rule found
45	R	Definite output in case of different possible evaluation sequences: *Wrong*, because dependent on compiler: x = b[i] + i++; *Wrong,* (which function is evaluated first?): x = f(&a) + g(&a); *Wrong,* since nested allocation: x = f(y = z/3); *Wrong*, (is i increased once or twice? - dependent on b): #define XYZ_GET_MAX(a, b) (((a) > (b)) ? (a) : (b)) z = XYZ_GET_MAX(i++, j);	No matching rule found
46	R	No not-zero-expression without any side effect, e.g. "a+1;"	No matching rule found

47	R	A zero-expression gets its own row (also no comments)
		
		`e.g. bad:`
		`for ( ... );`
		`Better:`
		`for ( ... )`
		`{`
		`;`
			2212, 2214

Index	Type	Rule	QAC Rule
		`}`	
48	E	Install dynamic runtime checks for better error checking in the code (where possible), *e.g.* by using assert • Examine scope of application of variables (prevention of overflow, division by zero, …) • Check valid objective of pointers • Prevention of the loss of the most significant bit in case of left shift • Examine validity of the array indication Possibly meaningful only before the release	2771, 2772, 2773, 2776, 2777, 2778, 2790, 2791, 2792, 2793, 2800, 2801, 2802, 2803, 2810, 2811, 2812, 2813, 2814, 2820, 2821, 2822, 2823, 2824, 2830, 2831, 2832, 2833, 2840, 2841, 2842, 2843, 2845, 2846, 2847, 2848, 2834, 2860, 2861, 2862, 2863, 2910, 2911, 2912, 2913, 2920, 2921, 2922, 2923, 2930, 2931, 2932, 2933, 2950, 2951, 2952, 2953, 2961, 2962, 2963, 2971, 2972, 2973, 2980, 2981, 2982, 2983
49	E	Enhanced setting of brackets for clarifying evaluation sequences, *e.g.* x_ui32 = (3 * a_ui16) + (b_ui16 / c_ui08);	No matching rule found

13.3 Control Structures

Index	Type	Rule	QAC Rule
50	R	The comma (,) is used as operator only in for-loops, *e.g.* okay: for(i = 0*,* j = 23; (i < 10) && (j > 15); i++, j--) { ... }	3417
51	R	No implicit logical conditions, *e.g. bad*: if (my_ui08) { ... } *Better*: if (0 != my_ui08) { ... }	1477, 1478, 3344, 4115, 4116
52	R	Do not ignore blocks through #if 0 conditions, *e.g.* #if 0 ... #endif *Exception:* For RTRT tests, blocks can be deactivated using compiler switches, *e.g.* #ifndef RTRT_TEST … #endif However, this is normally used only in old projects, in which only the changed parts of a module are tested with RTRT. Therefore, the process is to be reconciled with the SW-PM or SW-QM.	No matching rule found

53	R	No never-accessible "dead" code, *e.g.*: Code between switch and first case allocation
		switch(var)
		{ **x = xyz_Func_ui08();** /* NOT SO - is ignored */ case 1:... break; default: break; }
			2882

54	R	No use of labels except in case of switch-expression	No matching rule found
55	R	No use of goto	2001
56	R	No use of continue	2005

57	R	Code in the "body" of the construct "if", "else", "else if", "do ... while", "for" is **always** bracketed, e.g.
		`if (test_b == 1)`
		`\{`
		`xyz_DoThis_v();`
		`\}`
		`else`
			2212, 2214

Index	Type	Rule	QAC Rule

		`\{`
		`xyz_DoThat_v();`
		`\}`	

58	R	After a construct from "if" and "else if", an "else" must follow ,
		e.g.
		
		if (x < 0)
		{
		xyz_ DoThis_v();
		}
		else if (y < 10)
		{
		xyz_ DoThat_v();
		}
		else
		{
		xyz_HandleError_v();
		}
			3402

59	R	Every case-allocation within a switch-construct is completed with a break (do not use "fall-throughs"), except if it is intentionally blank.
		
		`switch (var_ui08)`
		`{`
		`case 1:`
		`xyz_ DoXY_v(); /* NOT ALLOWED "FALL THROUGH" */`
		`case 2:`
		`xyz_ DoThat_v();`
		`break;`
		`case 3: /* BLANK "FALL THROUGH" - OKAY */`
		`case 4:`
		`xyz_ DoSomething_v();`
		`break;`
		`default:`
		`xyz_HandleError_v();`
		`break;`
		`}`
			2003

60	R	Every switch-construct has at least two sections (case) and ends with a default-section. Otherwise, it is better to use an if-else application. switch-constructs on a boolean are meaningless and are not used.	2002
61	R	In a for-loop, only its control allocations occur in the loop header. *e.g.* bad: for (i = 0; (**number_ui08--**) > 0; i++) ...	2471, 2472
62	R	The for-loop variable is not modified within the loop.	2467
63	E	No use of break except in case of switch-expression	767, 769, 771

103	E	Every not-zero statement should have a side effect. *e.g.*
		unsigned int a = 0U;
		unsigned int b = 0U;
		a++ && b; /* 'b' has no side-effect */
		b++;
			No matching rule found

14 C Coding Guidelines - Design Rules

14.1 Abbreviations

Abbreviation	Description
R	Rule; this guideline is to be followed
E	This recommendation is meaningful, but not mandatory
Mnnn	Additional rules, additional to MISRA

14.2 Exclusive Access to SW-areas

(Rule no.: 94)
The specification that every resource (ports, variables, ...) should only be written exclusively is applicable in general. Contrary to the explicit write access, reading of the resources by several participants is permitted. Detailed information is available in the architecture guidelines.

Exception: In case of bit fields or microcontroller pins that can be written by multiple tasks and/or interrupts, it must be ensured that during a read-modify-write sequence,i.e. during

- the reading of the bit field/port
- the modification of the individual bit and
- the writing back

in a code sequence, no other task/interrupt should write to this bit field/port.
This can be ensured, for example, in Fujitsu by blocking all tasks/interrupts.

Example:
```
DisableAllInterrupts();
Set_Pin_19_4(); //ReadModifyWrite-Sequenz
EnableAllInterrupts();
```

Other possibilities are:

- commands that are atomic / cannot be interrupted are used
  e.g. ORB or ANDB in the Fujitsu MB91F467 (AtomicBitManipulation.h)
- Use of pre-defined interfaces
  e.g. use of gpio_set/gpio_reset

14.3 Interrupt Context

(Rule no.: 95)

The processing of an interrupt context should always be coordinated and reviewed with the **software architect**. The software architect uses the Checklist for Interrupts for the review.

A **Call-Tree** should be documented and maintained for each interrupt. (Documentation in the module)

Generally, only the absolutely necessary activities should be implemented in the interrupt.
All the functions that can also be executed in a task should not be executed in the interrupt.

Naming convention: Every interrupt service routine is to be renamed with the suffix "...Isr". Further, a comment should refer to the context "Interrupt".

The **running time** is to be reduced to a minimum.

Access to **hardware** should be avoided. Designs, in which a HW-signal is expected or the duration for the writing/reading of the HW-interface cannot be ensured in all the cases, are permitted only in an exceptional case and after review with a SW-architect.

Appropriate mechanisms for the synchronization are to be provided during the communication with other tasks and access to **common data** from the interrupt. The processing should not be interrupted in any case particularly in the interrupt handler (e.g. due to maintenance at a semaphore or in a register value) and also not in case of abort conditions.

In the interrupt handler, only those functions may be used, which are explicitly intended for one such use. the **released functions** are maintained by the project / department

While using **external modules** and **callbacks**, the developer must check, in which context the callback is called. (Interrupt or task context). If the callback involves an interrupt, the rules described for interrupts are applicable.

In case of changes to the source code, the existing code is gradually adjusted to this rule.
Exceptions for the existing interrupt handler must be approved by the SW architects and the SW-Group manager.

```
*The following situations should be positively avoided in the
interrupt context:*

* Use of RTE in the interrupt.
* Use of the Netservices in the interrupt (since OS resources
that are not protected in the interrupt are used here)
* VxWorks: "semTake", "taskDelay" are prohibited in the
interrupt and return with an error value.
```

Remark: Exercise care while handling indirect interrupts. WDG Timer Callbacks take place in the interrupt!

14.4 Rules for copyright and copyright law

Rule no. 96:
Copyright information / information about the author should not be removed.

Exceptions: No exception.

Information about the rights of self-created software or external software that is used is to be documented in suitable form (preferably in the source code).

Rule no. 97:
Use of information protected by copyright in any form (source code, libraries…) in the product or tool software is not permitted.

Exceptions: There is an explicit permission by the author to use the information for product or tool software and the use is explicitly approved and documented by the person responsible for the project.

To avoid unplanned industrial property claims, the use of protected information in software, which is the subject of this document, is permitted only with an explicit approval.

Rule no. 98:
On principle, Open Source software should not be used (not even in parts) in the product or tool software.

Exceptions: A documented approval of the _CM OSS Officers[1]is available for the explicit use of these software components.Copyright / information about the author must be explicitly specified.

As a result of using Open Source software, the license model – of the Open Source software - being used bypasses the remaining software. This consequence could be that the own software, corresponding to the Open Source software license, must be disclosed. To prevent this, the use of the Open Source software is to be checked and released by the *CM OSS Officer*. Additional information about the Open Source software already released for use (Whitelist), the approval process as well as other details can be called via the inside portal[2]

14.5 Rules for compiler warnings

Rule no. 99:
Compiler warnings are treated as errors
Exceptions: Code coming from suppliers can contain warnings
To ensure high quality code, warnings encountered during the build process are treated as build errors

Rule no. 100:
Suppression of compiler warnings is not allowed
Exceptions: The project manual can specify a set of warnings which are suppressed for the whole project.
Compiler warnings must not be removed by suppressing the warning, but through code changes.

Footnotes

1 _ At the time of creation of this document (July 2011), these items were carried out by Dr. Ing. Wolfgang Detlefsen (ADIT/EAR CM-AI /PJM-OS).

2 https://inside-docupedia.bosch.com/confluence/display/opensource/Open+Source+Usage

15 C Coding Guidelines - Compiler-Specific Behavior

15.1 Bitfield Handling

(Rule no.: 101)
The GreenHills Compiler and maybe also other Compilers optimize Bitfields by reserving space only for the declared Elements of a Bitfield. But the Access-Witdh is always a Processor-Word. In this case Data that is in the same struct directly behind the Bitfield will also be read and written, this leads to a Read-Modify-Write-Problem when Data in the Bitfield and Data directly behind are written in different Tasks and/or Interrupts.

```
{
  uint8 Bit1:1;
  uint8 Bit2:1;
  uint8 reserved:6;
  uint8 otherData;          //<- Will be Read and Written when 
Bit1 or Bit2 is written
  uint8 otherData1;         //<- Will be Read and Written when 
Bit1 or Bit2 is written
  uint8 otherData2;         //<- Will be Read and Written when 
Bit1 or Bit2 is written
}
```

To prevent this, there are two possibilities:
1. Put Bitfields into a Sub-Struct; This is recommended for new Modules

```
typedef struct
{
  uint8 Bit1:1;
  uint8 Bit2:1;
  uint8 reserved:6;
} myBitfield_t;
{
  myBitfield_t Bits;
  uint8 otherData;
}
```

2. Fill up each Bitfield to Processor-Word-Size with reserved Bits; This is helpfull for existing Code

```
{
  uint8 Bit1:1;
  uint8 Bit2:1;
  uint8 reserved:6;
  uint8 reserved2;
  uint8 reserved3;
  uint8 reserved4;
  uint8 otherData;
}
```

15.2 Structure Handling

The GreenHills Compiler and maybe also other Compilers set up structures by the sizeof operater respecting allignment. In order to keep the allignment the compiler inserts stuff bytes into reserved memory of a structure. This may lead to problems by comparing two sructures of same type with ***memcmp*** function if at least one of the structures is created on stack. For saving execution time the structure variable on stack is not initalizied. Therefore it is possible to have a stuffbyte filled by old stack content and not expecting by 0.
So it is possible to compare two structures not equal because of a different stuff byte never written by application or initializied.

Following example is based on 16bit allignment.

```
typedef struct {
  uint8 array_a[15];
  uint8 array_b[11];
  uint8 array_c[3];
  uint16 var_a;
  uint8 var_b;
} my_struct_t;
```

This leads to following memory reservation on stack.

Memory								
	00	**02**	**04**	**06**	**08**	**0A**	**0C**	**0E**
0x00	array_a[1 2]	array_a[3 4]	array_a[5 6]	array_a[7 8]	array_a[9 10]	array_a[11 12]	array_a[13 14]	array_a[15] array_b[1]
0x10	array_b[2 3]	array_b[4 5]	array_b[6 7]	array_b[8 9]	array_b[10 11]	array_c[1 2]	array_c[3] ***stuffbyte***	var_a
0x20	var_b ***stuffbyte***							

Both stuff bytes now have a random value. This means any other application used the same memory adress before and left it with its last value before stack was freed. Now stack is reserved by above structure but it is not initializied with special values like 0 before. Therefore it is possible to have random values in stuff bytes. And this stuuff bytes are also compared by ***memcmp*** function.

To avoid this the structure can be rearranged. Reearranging is only possible if allignment is known.

```
typedef struct {
  uint8 array_a[15];
  uint8 array_b[11];
  uint8 array_c[3];
  uint8 var_b;
  uint16 var_a;
} my_struct_t;
```

This leads to following memory reservation on stack.

Memory								
	00	**02**	**04**	**06**	**08**	**0A**	**0C**	**0E**
0x00	array_a[1 2]	array_a[3 4]	array_a[5 6]	array_a[7 8]	array_a[9 10]	array_a[11 12]	array_a[13 14]	array_a[15] array_b [1]
0x10	array_b[2 3]	array_b[4 5]	array_b[6 7]	array_b[8 9]	array_b[10 11]	array_c[1 2]	array_c[3] var_b	var_a
0x20								

A better way is to initialize the structure manually by a ***memset*** function call. This will allways work because the whole reserved memory is set to 0 including the stuff bytes independent of allignment.

```
my_struct_t my_var_s;

(void)memset(&my_var_s, 0x00, (uint8)sizeof(my_struct_t));
```