Maturity	Process Owner	Check	Release	Description
Valid	Khosrau Heidary SV P ED T DA 2004/02/01	Andreas Franke SV P ED GS SW1 ECU1 2004/02/23	Jean-Marc Andre SV P ED 2004/02/24	Version 2.0 Major rework, integration of MISRA 1.4

Title: C Coding Rules

Purpose:

This document defines a set of C Coding rules, which should be applied by the powertrain software developers during the coding phase of the software development process, as described in P04903. The main purpose of the rules is to guarantee a C code, which is portable across various target systems in order to improve reusability of the software. Furthermore, the rules shall lead the developers to a common way of coding, thus increasing readability and maintainability of the generated code.

This document is not designed as a tutorial about the usage of the C language, neither as a style guide for C developers. It should rather be considered as a collection of additional rules and restrictions to the international C standard defined in ISO/IEC 9899, the latter being assumed to be well known by the user. It is not within the scope of this document to illustrate the content of this standard. It is the responsibility of each developer to conform to this standard.

Responsibilities and a description of the software development process can be found in P04903.

The content of this Method is also available as HTML files and as MS WORD document with embedded VBA search and filter macros. The Team is also responsible for the maintainance of these items.

Scope:

The rules listed in this document are mandatory for all SV P software developers. Especially the following projects must use this guideline:

- 32bit Engine Management Systems GS & DS (EMS2 and ECU06)
- Transmission Control Units - projects started in 2002 or later, all families

Table of Contents:

1. Terminology, Definitions and Abbreviations

1.1 Numbering

The presented rules are grouped in various topics. Each rule has a unique identifier, which consists of a capital letter representing the topic, to which the rule belongs, and a number to distinguish from the other rules of the topic, e.g. C.3 (rule 3 of topic “Comments”).

1.2 Structure of the rules

Each rule is presented in a table of the following structure:

<ID> <rule>		
Description	<detailed description of the rule>	
Background	<additional information>	
Note	<additional hints for usage>	
Class of rule	[A | B] [+]	<optional further restrictions to scope>
Verification	<how to verify the rule>	
Example	Bad	Good
	`<example>`	
	<explanation of example>	
Reference	<references>	

The rule itself is given in the header of the table, together with the identifier. However, the full content of the rule is described in detail in the “Description”, which is a substantial part of the rule.

Further information which is not necessary in any case to apply the rule, but which may help to understand the purpose of the rule, is given either in a “Background” or in a “Note” field.

A class is assigned to each rule, which can be

- **Class A**: The rule has to be followed in any case. Breaking the rule must be approved by the management in a documented way and the reason must be explained.
- **Class B**: The rule might be violated in well-founded cases; a detailed explanation has to be given.

If a rule is function critical, the class of the rule is marked additionally with a **'+'**. Function critical means, that there is a high risk of a functional misbehavior if the rule is violated; especially when used in another environment (project or tool-chain).

The way, how the rule can be verified during the coding phase and in a code review, is listed in the “Verification” attribute. The verification can be done by either reviewing the

code manually or by using checking tools, like PC-Lint. The PC-Lint message number is given, where available.

In the given examples some formal things like memory allocation GMEM or obvious definitions and includes are dropped.

If appropriate, each rule contains a reference to another rule or some other document (e.g. ISO/IEC 9899). Some of the rules coincide with rules of the MISRA guidelines (Coding guidelines of the Motor Industry Software Reliability Association). Where this is the case, a reference has been added to the rule.

1.3 Tool requirements

The rules in this document are based on the assumptions, that

- the compiler is ISO/IEC 9899-compliant,
- signed numbers are represented using the two's complement,
- inline-functions are inserted at compile time and not located separately.

This should be ensured by the generic compiler options.

Verification of the coding rules with LINT assumes, that

- the generic options are used,
- the generic script is used (LINT does an unit checkout for one module),
- LINT is only performed on the module to be checked.

If this is not fulfilled additional checks have to be performed manually at code review.

1.4 Summary of the Rules

General Rules

Comments

Data and Data Types

Operators and Expressions

Statements and Control Structures

Functions

Preprocessor directives

Form and Appearance

Namespaces

2. Process

2.1 General Rules

G.0 All code shall conform to ISO 9899 standard C		
Description	The only allowed extensions to the ISO standard 9899 (1990) are the use of inline functions and the use of the inline assembler.	
Background	Using non standard / non documented features might compromise portability and even lead to undefined behavior.	
Class of rule	A	Compiler dependent code may use intrinsic functions
Verification	Code review, PC-Lint no. 950	
Reference	MISRA guidelines, rule 1 ISO/IEC 9899 Rule S.3, F.6	

G.1 Use templates for header and code and fill them out properly		
Description	Use templates for header and code to comply with the coding architecture and fill them out properly.	
Background	The templates contain standard elements: - standard comment header - mechanisms to avoid multiple inclusion of header files - order of the declaring sections etc.	
Note	Don't mark the heading of empty contractions with "empty".	
Class of rule	A	
Verification	Code review	
Reference	Templates	

G.2 Apply the memory allocation rules

Description	The current memory allocation rules have to be applied to code and global data.	
Background	The memory allocation rules must be used to ensure portability of the code, runtime optimization and optimum usage of available memory.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`/* Global Data Definitions*/` `#pragma section DATA ".IRAM_8_I" ".IRAM_8" near-data` `u8 lv_toto;` `/* Exported function */` `#pragma section CODE ".A_CODE" standard` `void c_toto (void) { … }`	`/* Global Data Definitions */` `#define MEM_DATA MEM_DATA_PUBLIC_8` `#include <gmem.h>` `u8 lv_toto;` `/* Exported function */` `#define MEM_CODE MEM_CODE_100MS` `#include <gmem.h>` `void c_toto_100ms (void) { … }`
	In the “good” example the memory allocation mechanism is included for code and global data. This procedure is described in the “Memory allocation rules”.	
Reference	Memory allocation rules	

G.3 Apply the naming convention rules

Description	Apply the naming convention rules for naming of data, data types, functions, defines etc.	
Background	In order to make software readable and comprehensible, all developers shall use the same keywords and follow the same rules for building up names.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`flag bit_tgl_lbda;`	`flag lv_lam_tog;`
	The “bad” example does not comply with the naming convention rules. For logical values the keyword “lv” is used, “lbda” and “tgl” are invalid keywords.	
Reference	Naming convention rules	

G.4 Header files must not generate code		
Description	Header files are only used to export items. Therefore header files must not contain executable code and must not define variables. In other words, header files must allocate no memory.	
Note	Nevertheless variables may be declared, but only with the keyword extern. Exported definitions are: types, constants, variables, macros and function prototypes. Defining inline functions and macros in the header does not allocate memory and therefore it is allowed.	
Class of rule	B	
Verification	Code review	
Example 1	Bad	Good
	`flag lv_toto;`	`extern flag lv_toto;`
	In the “bad” example memory is allocated by defining a datum.	
Example 2	Bad	Good
	`toto = lamb;`	`#define SET_TOTO(a) {toto = (a);}`
	In the “bad” example the header file generates code. The “good” example exports a macro. Code will only be generated when the macro will be called within a .c-file.	

G.5 Avoid unused data, code and local preprocessor definitions

Description	Unused data, code and local preprocessor definitions must not appear in finished programs. Data or code, which is not used (accessed, called or exported according to the specification) within any configuration, should be deleted. Data and code, which are only required for a certain configuration (e.g. testing), should not allocate memory. (This should be steered by compilation switch.)	
Background	Avoid unused data, code and preprocessor keywords to save resources and increase readability.	
Note	Software instrumentation specific code that could effect software real time or dynamic (OS benching) must not be reached by main configuration and must be covered by compiler switch.	
Class of rule	A	B for preprocessor definitions
Verification	Code review, PC-Lint no. 506, 527, 528, 529, 550, 563, 681, 750, 751, 752, 753, 754, 774, 827, 944	
Example 1	Bad	Good

```
#define TOTO 1
u8 x;
u8 y;

#if TOTO == 1
  x = 0;
#else
  y = 0;
#endif
```

```
#define TOTO 1

#if TOTO == 1
   u8 x;
#else
   u8 y;
#endif

#if TOTO == 1
  x = 0;
#else
  y = 0;
#endif
```

In the “bad” example the variable y will never be used.

Example 2 Good

```
#if USE_FIFO_STACK_MES
const u8 c_fifo_stack_mes;
flag lv_fifo_stack_mes;
#endif


#if USE_FIFO_STACK_MES
/* Fifo and stack measurement activation / desactivation */
if (c_fifo_stack_mes == 0xFF ) {
   lv_fifo_stack_mes = 1;
}
#endif
```

In this example the code can be made effective by a compiler switch, which defines USE_FIFO_STACK_MES. If the code is deactivated by the compiler switch also the data definition should be deactivated.

Reference	MISRA guidelines, rule 52 Rule S.9

G.6 Do not redefine or replace reserved words

Description	Reserved C words, standard library function names, operators and other special characters of the C language may not be redefined through use of *#define* nor be replaced by other names. Also SV P-own standard data types may not be redefined.	
Note	The definition of special (SV P-own) standard data types is required to realize compiler independent code. This is mainly performed in the general header file *gtypes.h*, which is a standard include for SV P.	
Class of rule	A+	Local replacement of *extern* for combined definition and declaration of logistic or shared data is permitted.
Verification	Code review, PC-Lint no. 14, 15, 547, 683	
Example	Bad	Good

```
#include <gtypes.h>

#define INCASE if       /* replace */
#define U16  u16        /* replace */
#define s16  signed int /* redefine*/

s16 value_s16;
U16 value_u16;

INCASE (value_s16 < 0)
```

```
#include <gtypes.h>




s16 value_s16;
u16 value_u16;

if (value_s16 < 0)
```

In the “bad” examples keywords were replaced or redefined.

Reference	MISRA guidelines, rule 114, 115 gdatatypes.doc

G.7 Generation of executable must be done without any warning

Description	In general, generating the executable shall be done without any warning of the productive tool chain and PC-LINT. For the case, the warning can not be avoided, it has to be checked and documented.	
Note	Some compilers give a warning whereas others don’t. The aim is not to suppress all warnings, but to clean up the code causing the warning or document the warning. Switching off warnings in the tool configuration is not an option.	
Class of rule	B+	
Verification	Code review	
Example	Bad	Good

```
#define TOTO 1
u8  x;

if (TOTO)
  x = 0;
else
  x = 1;
```

```
#define TOTO 1
u8  x;

#if TOTO == 1
  x = 0;
#else
  x = 1;
#endif
```

In the “bad” example the else path will never be executed. A compiler may give the warning “statement not reached”.

Reference	PC-Lint user manual Rule C.7

2.2 Comments

C.1 Use adequate comments		
Description	Comments should complete traceability between detailed design choices made by the designer and coding implementation. Other kinds of comments, mainly based on semantic description of the programming language, should be avoided or limited. Misleading comments are worse than no comments. Comments are placed in front of the block they describe. These comments start at the first column of a line. All other comments following the corresponding code should be written either in the same or in a separate line.	
Background	Comments in source code are the **most important aid for software maintenance**. The use of the appropriate amount of comments makes the programs more readable and understandable. However, comments should only contain information that is adequate for reading and understanding the program. Comments should not just repeat information given in the specification.	
Note	Comments shall always be in English	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`lv_fct_en = 0;` `/* set lv_fct_en */`	`lv_fct_en = 0; /*disable fct funct*/`
	The comment in the “bad” example - is incorrect, - placed after the statement it describes and - contains no further information for the reader.	

C.2 Add comments at definitions		
Description	Each definition of data (not described in specifications), macros and functions has to be commented to describe its purpose if not obvious.	
Background	Use comments at definition to increase readability and maintainability of code.	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`u8 lv_lam_tog;` `#define MAX(x,y) \` `(((x) > (y)) ? (x) : (y))`	`u8 lv_lam_tog; /* sync. flag for lambda controller */` `/* macro returns max value of x or y */` `#define MAX(x,y) \` `(((x) > (y)) ? (x) : (y))`
	In the “good” example the variable and the macro are described at definition with comments.	
Reference	Rule C.3	

C.3 Add comments at interface declarations		
Description	Add comments to the interface declarations in the header file.	
Background	The interface description increases readability and maintainability of code. For integration of object files it provides additional information.	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`extern u8 lv_lam_tog;`	`extern u8 lv_lam_tog; /* sync. flag for lambda controler */`
	The exported interface is commented in the “good” example. See also rule C.2.	
Reference	Rule C.2	

C.4 Comments must not be nested		
Description	Nested comments are not allowed.	
Background	Using nested comments may lead to unintended omitted comment end, which is hard to find.	
Note	The X-Tools comment-block is already a comment	
Class of rule	A+	
Verification	PC-Lint no. 602	
Example	Bad	
	`/* start` `  lv_toto = 0` `/* call function c_xx() */` `c_xx();`	`/* start */` `/* lv_toto is 0 */` `/* call function c_xx() */` `c_xx();`
	The comment in the “bad” example is nested. It is not obvious whether the second line should be a comment or a statement.	
Reference	Kernighan / Ritchie annex A.2.2 MISRA guidelines, rule 9	

C.5 Sections of code shall not be commented out		
Description	When certain sections of source code are required not to be compiled, this should be achieved by using conditional compilation (e.g. *#if* or *#ifdef* constructs).	
Background	Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`lamb_1 = 0;` `/* lamb_2 = 0; */`	`#define USE_LAMBDA_REF SINGLE_LS` `lamb_1 = 0;` `#if (USE_LAMBDA_REF != SINGLE_LS)` `lamb_2 = 0;` `#endif`
	The code in the “good” example is excluded by a compiler switch.	
Reference	MISRA guidelines, rule 10 Rule G.5	

C.6 Use old C-style comments only		
Description	C++ like comments shall not be used in C code.	
Background	Although many C compilers support this form of comment (denoted by //), they are not a part of ISO standard C.	
Class of rule	A	
Verification	PC-Lint no. 950	
Example	Bad	Good
	`// comment`	`/* comment */`
	In the “bad” example C++ comment is used.	
Reference	MISRA guidelines, rule 1 Rule G.0	

C.7 Each suppression of PC-Lint messages shall be explained

Description	Top priority is to avoid suppressing any PC-Lint warning. A comment that suppresses a PC-LINT message must be well explained. The scope of the LINT comment should be as small as possible. Lint warnings have to be suppressed at the cause (e.g. at macro or stub definition) and not where they appear.	
Background	Suppressing unavoidable PC-Lint messages makes a PC-Lint output more clear and prevents this message to be analyzed in every code review again.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`u16 c_xxx(u16 x) {` `  u32 i = 0;` `  while (i <= x) i++;` `  return (x / i);` `}`	`u16 c_xxx(u16 x) {` `  u32 i = 0;` `  while (i <= x) i++;` `  return (x / i); /*lint !e795*/ /*` `  i always > 0  -> no div by zero */` `}`
	The “bad” example generates the PC-Lint warning 795 “Conceivable division by 0“. The “good” example suppresses this warning and describes the reason why. The suppression is only effective for the particular code line.	
Reference	PC-Lint user manual, chapter: suppressing errors	

C.8 Any non-obvious integer constants must be commented

Description	Any non specified constants (like resolution adaptations, offset corrections, etc.) have to be commented in order to clarify the origin of the constant.	
Background	This improves the readability of the code and makes code reviews easier.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`foo = bar / 10;`	`/* resolution of bar is 10 times` `finer than that of foo */` `foo = bar / 10;`
	It is good practice to explicitly show how to calculate the adaptation constant out of the different resolutions involved.	

2.3 Data and Data Types

D.1 Use predefined data types		
Description	Use only the predefined data types (see references). Use the smallest data type, which fulfills your needs in terms of range. Data types are defined in the general header file *gtypes.h*, which is a standard include for SV P. An exception to this rule is the rule "D.3 Use the *int* data type exclusively for loops, index addressing, etc."	
Background	The use of the standard ISO – but compiler dependent – data types (such as char, int, long, long long) causes problems, because the elementary data types of C are very vague in defining the data type size. Therefore types with a fixed size and behavior are defined compiler independent. Their implementation is of course compiler dependent.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 970, 971	
Example	Bad	Good
	`int value_16;` `unsigned long value_32;`	`s16 value_16;` `u32 value_32;`
	In the "bad" example the size of the data is compiler dependent. The C specification only defines a minimum width for data types (e.g. *int* has a minimum width of 16 bits). With the Tasking C166 the size of value_16 would really be 16 bit but on a Diab Data for PowerPC the size of value_16 would be 32 bit.	
Reference	gdatatypes.doc ISO/IEC 9899 section 5.2.4.2.1, annex E Kernighan / Ritchie annex B.11 MISRA guidelines, rule 13 Rule D.3	

D.2 Avoid preprocessor defines for data type definitions		
Description	Use *typedef* for defining user defined data types instead of the preprocessor keyword #*define*.	
Background	C introduces *typedef* to define a data type. The usage of *typedef* provides the following advantages: – defined types can be supported by the debugger. – better compiler checks are possible. – extension of the user defined data type to a more complex one is possible (without unwanted side effects). – the same scope rules as for data definitions apply.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`#define U16_PTR u16 *` `U16_PTR ptr1, ptr2;`	`typedef u16 * u16_ptr_t;` `u16_ptr_t ptr1, ptr2;`
	In the “bad” example the ptr2 is **not** a pointer but a u16 only.	
Reference	MISRA guidelines, rule 90	

D.3 Use the *int* data type exclusively for loops, index addressing, etc.		
Description	Local (automatic) variables like loop counters and indexes shall be defined with the *int* (signed or unsigned) data type, if 16 bit is enough. Otherwise use the predefined data types. Do not reference the variable of the data type *int* by pointers or a calibration system. Ensure that comparing the variable to predefined data type variables will not fail due to a different implementation of *int*.	
Background	Using data types smaller than the register size may cause lots of code overhead especially in loops or index addressing. You will get better results using the C standard type *int.* *int* is the native data type of the processor – therefore it is better to use this data type for loop counters and local variables which are in a register. The minimum size of the data type *int* is defined by the C standard to be 16 bits.	
Note	There is no defined overflow / underflow behavior for the data type *int*.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`u8 ctr;` `for (ctr = 0; ctr < 42; ctr++) {` `  …` `}`	`int ctr;` `for (ctr = 0; ctr < 42; ctr++) {` `  …` `}`
	The “bad” example may produce a lot of code and runtime overhead due to required limitations to the u8 data type.	
Reference	ISO/IEC 9899 section 5.2.4.2.1, annex E Kernighan / Ritchie annex B.11 Rule D.1, D.20 gdatatypes.doc	

D.4 Do not use the *flag* or *int* data type if you need a defined overflow / underflow characteristic		
Description	If a defined overflow/underflow characteristic is required, use the predefined data types.	
Background	The data types with an explicit given bit width (s8, s16, s32, …; u8, u16, u32, …) provide a defined overflow/underflow behavior. The data types *flag*, *int* and the floating point data types do not.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`unsigned int value;` `…` `value++;`	`u16 value;` `…` `value++;`
	In the "good" example you can expect that *value* is always between 0x0000 and 0xFFFF. In the "bad" example also values above 0xFFFF can be possible.	
Reference	gdatatypes.doc	

D.5 Flags shall contain 0 and 1 only		
Description	The data type *flag* is defined to describe a boolean information only. Store only 0 or 1 in a variable of this type. If more information needs to be stored in a variable then bitfields or data types (u8, …) shall be used.	
Background	The definition of the data type *flag* depends on the available basic data types of the processor. If the processor provides a *bit* data type, the bit size of *flag* may really be 1! So do not store more information even if it seems to be possible in the current implementation.	
Class of rule	A+	
Verification	Code review	
Example 1	Bad	Good
	`flag lv_x;` `u16 my_data;` `lv_x = TRUE;` `lv_x = 0xFE;` `lv_x = my_data;`	`flag lv_x;` `u16 my_data;` `lv_x = 1;` `lv_x = 1;` `lv_x = (my_data != 0);`
	In the "bad" example you don't know if TRUE and FALSE is equal to 1 and 0. In the good example the values are defined exactly. The value 0xFE is assigned to a flag. This represents logical "TRUE" too, but does not work on every implementation. The right side of an assignment to flags should always be a Boolean expression.	

Example 2	Bad	Good
	`/* Processor 1: flag as int */` `typedef int flag;`	
	`/* Processor 2: flag as bit */` `typedef bit flag;`	
	`/* wrong usage of flag: */` `flag lv_status = 0x10;` `if (lv_status & 0x10)` `…`	`/* don't use flag in this case: */` `u8 status = 0x10;` `if (status & 0x10)` `…`
	The "bad" example will not work if flag is really only a bit.	
Reference	gdatatypes.doc Rule O.10	

D.6 Do not use a pointer to a flag

Description	Do not use a pointer to a variable of data type *flag*. Use a different data type, if indirect access is required.	
Background	If the data type *flag* is based on the type *bit,* the bit size may be not addressable. Do not access bits by pointers.	
Note	It is recommended not to use the data type *flag* for arrays and inside structures. The current usage of the data type *flag* prevents using a real *bit* data type for it.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`flag lv_status;` `flag *ptr = &lv_status;`	`u8 status;` `u8 *ptr = &status;`
	The "bad" example will not work (or even get compiled) if flag is really a (non-addressable) bit.	

D.7 Do not initialize global and static variables with initial value 0

Description	Any global or static variable, which should be initialized with 0 during startup, shall not be initialized explicitly.	
Background	The initialization of global or static variables with 0 will be done at startup time by the RAM test. An explicit initialization would result in additional startup time. The RAM test has to be configured accordingly.	
Class of rule	B	This rule does not apply to data which are not cleared by RAM test.
Verification	Code review	
Example	Bad	Good

```
u32 v1 = 0;
static u32 v2 = 0;
u32 v3;
u32 v4;

/* initialization function called at
startup */
void init(void)
{
   v1 = 0;
   v2 = 0;
   v3 = 1;
   v4 = (u32)NC_XXX;
}

void func(void)
{
   u32 v5 = 0;
   …
}
```

```
u32 v1;
static u32 v2;
u32 v3 = 1;
u32 v4;

/* initialization function called at
startup */
void init(void)
{
   v4 = (u32)NC_XXX;
}

void func(void)
{
   u32 v5 = 0;
   …
}
```

In the “bad” example the memory areas of v1 and v2 will be re-initialized twice with 0 during startup, (if not optimized away by compiler). The variables v3 and v4 with initial values other than 0 can be initialized either via initialization function or at their definition.

D.8 All automatic variables shall have been assigned a value before being used		
Description	Each local automatic (register, stack) variable has to be initialized before being used, even with initial value 0.	
Background	Read access must not happen to a local variable before it has been explicitly initialized. This is also true if a compiler implicitly initializes all variables, because another compiler may initialize differently.	
Class of rule	A+	
Verification	PC-Lint no. 530, 603, 644, 645, 771, 772	
Example	Bad	Good
	`void foo(void)` `{` `  u32 local_var;` `  int i = 0;` `  any_var = local_var + 1` `  for (i = 0; i < 10; i++)` `  …` `}`	`void foo(void)` `{` `  u32 local_var = 0;` `  int i;` `  any_var = local_var + 1` `  …` `  for (i = 0; i < 10; i++)` `  …` `}`
	In the “bad” example the result of the operation may be undefined, because local_var has not been initialized correctly. The local data may be initialized at definition, but avoid double initialization as for "i" in the "bad" example.	
Reference	MISRA guidelines, rule 30	

D.9 Change global data only in the module which defines them		
Description	No other module than the one, which defines a global datum, is allowed to change its value in memory. The link between a datum and the module it belongs to is established via the specification.	
Note	A module should be understood as a functional unit, which may consist of .c, .h and .grl files (but not necessarily one .c file).	
Background	This rule should help to avoid unintended side effects. There are some exceptions to this rule (e.g. “handshake bits”), which have to be documented in the code.	
Class of rule	B+	
Verification	Code review	
Example	Bad	Good
	`module_1.h` `----------` `extern u32 a;` `module_1.c` `----------` `u32 a;` `   …` `   a++;` `module_2.c` `----------` `u32 *tmp;` `   tmp = &a;` `   *tmp--;` `or:` `module_2.c` `----------` `extern u32 a;` `   a--;`	`module_1.h` `----------` `extern u32 a;` `module_1.c` `----------` `u32 a;` `   …` `   a++;` `module_2.c` `----------` `u32 tmp;` `   tmp = a;` `   tmp--;`
	In the “bad” example the variable a, which is defined in module_1, is changed by module_2. This is avoided in the “good” example by copying the content of a into a tmp variable and changing that one instead.	

D.10 Limit the scope of each object as much as possible

Description	The visibility of an object should be restricted to the module, the function or the block it belongs to. The following aspects have to be taken into account: – Data which are used locally only shall be defined at function or block scope. – Implementation-specific objects with a scope beyond a function or a block have to be defined as *static*. – Module-local objects which are not accessed by a calibration system (i.e. which are not defined as visible within the SRS) have to be defined as *static*. – Any global (i.e. exported) object must not be *static*.	
Background	Declaring an object as *static* reduces the visible namespace. *Static* is a good information hiding of implementation-specific variables. Better optimization can be performed for static objects. It is considered good practice to avoid global identifiers as far as possible. Avoid conflicts due to double definition, though. Display data must not be *static* due to the required access by calibration tools.	
Note	Static variables might not appear in the map file.	
Class of rule	A	
Verification	Code review, PC-Lint no. 759, 765, 766	
Example 1	Bad	Good
	`MODULE1.C:` `/* private variable for MODULE1 */` `u8 var;` `MODULE2.C:` `/* private variable for MODULE2 */` `u8 var;`	`MODULE1.C:` `/* private variable for MODULE1 */` `static u8 var;` `MODULE2.C:` `/* private variable for MODULE2 */` `static u8 var;`
	If module 1 and module 2 are both used, the “bad” example will not work, since the compiler might map “var” to the same memory cell (in case of an old K&R compiler behavior) or produce an error message in the link stage due to multiply defined symbols.	
Example 2	Bad	Good
	`static u8 var;` `void func(void)` `{` `  var = 1;` `  if (var > 0)` `  …` `}`	`void func(void)` `{` `  u8 var;` `  var = 1;` `  if (var > 0)` `  …` `}`
	In the “bad” example the scope of the variable “var” is the whole module, but it is used only locally.	
Reference	MISRA guidelines, rule 22, 23	

D.11 Do not use the *static* storage class for temporary results		
Description	Do not declare variables, which are only used to store temporary results, as *static*. Declare them as automatic variables instead (i.e. don't assign any storage class to the variable).	
Background	A static variable will be stored in the memory, an automatic variable in registers or on the stack. Thus, using automatic variables will save memory and allow compiler optimization.	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`u32 var1, var2, var3;` `void func(void)` `{` `  static u32 tmp_var;` `  tmp_var = var1 + 2 * var2;` `  var3 += tmp_var;` `}`	`u32 var1, var2, var3;` `void func(void)` `{` `  u32 tmp_var;` `  tmp_var = var1 + 2 * var2;` `  var3 += tmp_var;` `}`
	In the "bad" example the variable tmp_var will be stored in memory, although it could be kept in a register.	

D.12 Do not use the *register* storage class		
Description	The *register* storage class must not be used.	
Background	The register storage class is only a suggestion to the compiler to provide fast access to the object. A good compiler should be able to decide for itself what to put in registers. Using the *register* storage can even prevent the compiler from optimization.	
Class of rule	A	
Verification	PC-Lint 961	
Example	Bad	Good
	`{` `  register u32 tmp_var;` `  …` `}`	`{` `  u32 tmp_var;` `  …` `}`
	Usually the variable tmp_var will be held in a register, anyway.	
Reference	MISRA guidelines, rule 28	

D.13 Do not use µC specific modifiers like near/far in portable code		
Description	The *near* / *far* type modifier is not defined by the ISO standard and thus must not be used.	
Background	The *near* / *far* modifier for pointer, data or functions is a compiler specific extension. Its usage is not portable.	
Class of rule	A	For portable code
Verification	Code review, PC-Lint 950	
Example	Bad	Good
	`near u16 *ptr1;` `far  u16 *ptr2;`	`u16 *ptr1;` `u16 *ptr2;`
	The “bad” example is not really portable because the type modifier might not exist. If it would exist on the different platforms the behavior might be different.	

D.14 Use “volatile” casts if needed for reading updated values

Description	Cast a variable to "volatile" if and only if all of the following conditions are fulfilled: - the variable is read - the code reading the variable is inside a function which reads the variable more than once - the variable is expected to be updated in between these read accesses (e.g. by an interrupt, by a higher prior pre-emptive task or by hardware) - it is necessary to see this update	
Background	If volatile is not used, the compiler might optimize sequential accesses to a variable by caching the value in a register. If the task is interrupted and the variable is changed, the change will not be visible to the lower prior task. The casting to volatile is more flexible and optimized than defining the variable itself as volatile. With the volatile cast the consumer task can decide in each case if a volatile access is necessary, it is not imposed by the producer task in advance.	
Note	Applying this rule the definition and the declaration of variables as "volatile" are no longer necessary.	
Class of rule	A+	not required for memory mapped I/O
Verification	Code review	
Example	Bad	Good
	`flag lv_flag;` `void task1(void){` `  /* high prior producer task */` `  lv_flag = 1;` `  …` `}` `void task2(void) {` `  /* low prior consumer task */` `  while (lv_flag)` `  …` `}`	`flag lv_flag;` `void task1(void){` `  /* high prior producer task */` `  lv_flag = 1;` `  …` `}` `void task2(void) {` `  /* low prior consumer task */` `  while ((volatile flag) lv_flag)` `  …` `}`
	The “bad” example may not terminate, because lv_flag is not reread in each iteration.	

D.15 Avoid stack overflow due to too many temporary data		
Description	Make sure that not more memory than available will be allocated on the stack during runtime. Take into account the size of the temporary objects and the function call depth. Meet the following limits if possible: – max. 8 local variables in non-leaf functions and libraries. – max. 20 local variables / 80 bytes array for leaf functions calling only library functions. – 4 nested function calls internal (excluding scheduler).	
Background	The stack size is limited. This limit must not be exceeded by allocating temporary data or implementing a deep function call stack. A stack overflow may cause undefined behavior.	
Note	Whenever it is possible pass large parameters (e.g. structures) by reference to functions. Protect them against changes by declaring the reference as pointing to `const`.	
Class of rule	B	
Verification	Code review	
Example 1	Bad	Good
	`{` `  u32 a[100];` `  …` `}`	`{` `  static u32 a[100];` `  …` `}`
	In the “bad” example 400 bytes will be allocated on the stack, whereas they will be allocated in data memory in the “good” example.	
Example 2	Bad	Good
	`u32 faku(u32 par)` `{` `  if (par >= 1)` `  return par * faku(par-1);` `  else` `  return 1;` `}`	`u32 faku(u32 par)` `{` `  u32 cnt = 1;` `  u32 fak = 1;` `  while (cnt <= par)` `  {` `  fak *= cnt;` `  cnt += 1;` `  }` `  return fak;` `}`
	In the “bad” example the function faku is implemented recursively. This may result in a stack overflow.	
Reference	MISRA guidelines, rule 70	

D.16 Do not use runtime memory allocation

Description	Don't use dynamic variables, e.g. don't use the heap functions malloc(), calloc(), free(), …	
Background	When using the heap extensively, it might either overflow (if memory is not freed) or be fragmented. In any case the system might not be able to allocate memory on the heap, which would lead to an undefined behavior.	
Class of rule	A+	
Verification	PC-Lint no. 421	
Example	Bad	Good
	`{` `  u8 *ptr;` `  ptr = (u8*)malloc(sizeof(u8));` `  …` `}`	`{` `  u8 var;` `  u8 *ptr;` `  ptr = &(var);` `  …` `}`
	In the "bad" example memory is allocated on the heap during runtime. This may fail due to heap fragmentation after some while. In the "good" example the linker/locator can check whether enough memory is available.	
Reference	MISRA guidelines, rule 118	

D.17 Prefer a well-named constant to an immediate value

Description	Constant values, which are used more than once, should be defined with the *#define* keyword (symbolic constants) or with the *const* type qualifier in order to improve maintainability and robustness.	
Background	The decision between *#define* and *const* should be driven by the fact that for data defined as *const*, type checking by the compiler is available and changing the value is possible with a debugger. A definition as *const* explicitly allocates memory for the data, whereas a *#define* implicitly allocates memory since it becomes part of the code section.	
Note	Constants in a *switch* statement can only be defined by *#define* or *enum*.	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`u32 a,b,x,y;` `x = a + 6250;` `y = b + 6250;`	`/* Timer Increment */` `#define TIMER_INC 400  /* 400ns */` `/* Offset 2,5ms */` `#define T_OFS (u32) (2500000 /` `TIMER_INC)` `u32 a,b,x,y;` `x = a + T_OFS;` `y = b + T_OFS;`
	In the "bad" example the constant has to be adapted each time it occurs.	
Reference	MISRA guidelines, rule 90 Rule D.24	

D.18 Declare input parameters for functions with type qualifier *const*		
Description	The type qualifier *const* should be used for function parameters to declare that the contents shall not be modified by the function (i.e. input parameter). This only applies to parameters passed by reference.	
Note	Whether a variable defined with a *const* qualifier is really protected against modifications depends on the compiler.	
Class of rule	B	
Verification	PC-Lint no. 818, 952, 953, 954	
Example	Bad	Good
	`void func(u8 *ptr)` `{` `  *ptr = 2; /* attempt to modify` `  value pointed to by` `  ptr succeeds */` `}`	`void func(const u8 *ptr)` `{` `  *ptr = 2; /*attempt to modify` `  value pointed to by` `  ptr is prevented */` `}`
	Assuming that ptr points to a value which shall not be modified by the function func, the “bad” example does not prevent such a modification, whereas the “good” does.	
Reference	MISRA guidelines, rule 81	

D.19 Cast immediate values (except 0..32767) explicitly

Description	Each immediate value (except 0..32767) has to be casted to a predefined data type (u16, …). The correct type and precision has to be guaranteed by this cast. Immediate values between 0 and 32767 do not need to be casted explicitly. The postfix identifiers 'L' (5L), 'U' (17U) must not be used.	
Background	This rule is an exception to rule D.20 for immediate values from 0 to 32767 to increase the readability. Constants are automatically promoted to signed integer if between 0 and 32767. Nevertheless if used in an unsigned or reduced data size context no problem can occur. This is why constants in that range do not have to be casted explicitly although it may be required by D.20. To avoid a very complex rule all other immediate values have to be casted although it may not be necessary according to D.20.	
Note	ISO does not allow type casts inside *#if* - constructs.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 122, 570, 573, 574, 620, 648, 650, 732, 734, 737, 915, 960, 961	
Example	Bad	Good
	`u16 a;` `u16 b;` `a=b/(u16)2;`	`u16 a;` `u16 b;` `a=b/2;`
	In the expression "b/2" the "b" is unsigned, but the "2" is signed. This would mean that the "2" has to be casted according to rule D.20. But due to the integer promotion rules an explicit cast is not necessary and without the code is more readable.	
Reference	MISRA guidelines, rule 18 Rule D.20	

D.20 Perform type casts explicitly if needed

Description	Avoid unnecessary explicit type casts. Perform type casts in the following cases: 1. Both operands of a binary operator (e.g. +, <, ==) have to be either unsigned or signed, but not mixed. If not perform the required typecast. 2. The left-hand operand of a binary non-relational operator has to be casted if its size is smaller than the expected size of the operation. 3. An explicit type cast is necessary to reduce the data size. 4. An explicit type cast is necessary for converting from signed to unsigned or back.	
Background	Implicit type conversions may vary on different systems.	
Note	The unary minus operator shall not be applied to an unsigned expression.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 524, 569, 570, 573, 574, 653, 712, 713, 731, 732, 734, 735, 736, 737, 776, 779, 912, 915, 916, 919, 961	
Example 1 **(not mixed unsigned – signed operands)**	Bad	Good
	`s16 x = -32000;` `u16 y = 2;` `if ((x / y) < 0)` `  …`	`s16 x = -32000;` `u16 y = 2;` `if ((x / (s16)y) < 0)` `  …`
	In the “bad” example x/y is evaluated to 16768 on 16 bit systems. In the good example it evaluates to -16000.	
Example 2 **(left-hand operand determines size of operation)**	Bad	Good
	`s16 var1;` `s16 var2;` `s32 var3;` `…` `var3 = var1 + var2 + var3;` `var3 = var1 + var2 + var3;`	`s32 var1;` `s16 var2;` `s16 var3;` `…` `var3 = var3 + var1 + var2;` `var3 = (s32)var1 + var2 + var3;`
	In the “bad” example the size of the first +-operation is probably "s16". A 16bit overflow may occur. In the good example the left-most operand is "s32", determining the size of the whole operation.	
Example 3 **(reduce data size)**	Bad	Good
	`u16 var1;` `u8  var2;` `…` `if (var1 < 256)` `{` `  var2 = var1;` `}` `else` `{` `  var2 = 0xFF;` `}`	`u16 var1;` `u8  var2;` `…` `if (var1 < 256)` `{` `  var2 = (u8)var1;` `}` `else` `{` `  var2 = 0xFF;` `}`
	In the "bad" example it is unclear whether data size is reduced intentionally or by mistake. LINT will issue a warning. In the "good" example the intention is documented.	

Example 4 (converting unsigned to signed)	Bad `s16 a;` `u16 b;` `…` `a = u16_min_u16_u16(b, 0x7FFF);`	Good `s16 a;` `u16 b;` `…` `a = (s16)u16_min_u16_u16(b, 0x7FFF);`
	In the "bad" example it is unclear whether the conversion to signed is done intentionally or by mistake. LINT will issue a warning. In the "good" example the intention is documented.	
Reference	MISRA guidelines, rule 39, 43, 44, 48	

D.21 *removed*

D.22 Pointers to objects must have the correct type

Description	Pointers to objects shall always have the correct type.	
Background	Alignment of pointers to different types might be different. Conversions of different pointer types can cause alignment problems. Make no assumption about the size of an object. Pointers to objects shall have the correct type.	
Note	Also valid for pointers to functions including return value and parameters	
Class of rule	A+	
Verification	Code review, PC-Lint no. 519, 549, 609, 610, 611, 740, 923	
Example 1 (pointer arithmetic)	Bad	Good
	`u16 *ptr_x;` `u8 x[5];` `int i;` `ptr_x = (u16 *)&x;` `for (i = 0; i < sizeof(x); i++)` `{` `  *ptr_x = i;` `  ptr_x++;` `}`	`u8 *ptr_x;` `u8 x[5];` `int i;` `ptr_x = &x;` `for (i = 0; i < sizeof(x); i++)` `{` `  *ptr_x = i;` `  ptr_x++;` `}`
	In the “bad” example the pointer addresses wrong elements because its increment width is 2. On different processor architectures there might be a misaligned data access.	
Example 2 (copy data from NVRAM buffer to fill a structure)	Bad	Good
	`u8 buffer[512];` `STRUCT_XXX destination;` `(void)LIB_MEM_Copy( &destination,` `  &buffer[0],` `  sizeof(destination) );`	`u8 buffer[512];` `STRUCT_XXX destination;` `(void)LIB_MEM_Copy( (void *)` `  &destination,` `  (void *)&buffer[0],` `  sizeof(destination) );`
	The library function LIB_MEM_Copy expects two pointers of type (void *). Therefore a cast is performed in the “good” example to provide the proper pointer types.	
Reference	MISRA guidelines, rule 45	

D.23 Use pointers and elementary data types as declared

Description	Define pointers if you want to access data indirectly. Never do a conversion between integer data types and pointers.	
Background	The size of pointers and pointer arithmetic are processor and compiler dependent. Distinguish between pointers and integer types – they are different even if they have the same size.	
Class of rule	A+	B for processor dependent code (e.g. communication SW, driver SW for memory I/O)
Verification	PC-Lint no. 507, 511, 923	
Example 1 (Using u32 as pointer)	Bad	Good
	`u32 ptr;` `u8 a;` `void f(void){` `  ptr = (u32)&a;` `  *((u8*)ptr) = 0;` `}`	`u8 *ptr;` `u8 a;` `void f(void){` `  ptr = &a;` `  *ptr = 0;` `}`
	The “bad” example uses an elementary data type as pointer. This is not portable since a pointer might be build up differently on different architectures (e.g. linear, segment + offset, page number + offset, …)	
Example 2 (Converting u32 to pointer in CCP)	Good	
	`u8 *tmp_ptr;` `tmp_ptr = (u8*) ( LIB_PRC_ShlU32 (ccp_rcv.data[2], 24 )` `  + LIB_PRC_ShlU32 (ccp_rcv.data[3], 16 )` `  + LIB_PRC_ShlU32 (ccp_rcv.data[4], 8 )` `  + (u32) ccp_rcv.data[5] );`	
	In this example an address is received via CAN in CCP. When building up the address, a conversion from u32 to (u8 *) cannot be avoided. An explicit cast is performed for this purpose. The rule does not apply in this case.	
Reference	MISRA guidelines, rule 45	

D.24 Do not use the enum type to define variables

Description	Variables of *enum* type are not usable due to high RAM-consumption. So use *#define* or enumeration sets together with the smallest possible data type (mostly u8) for state variables. In the latter case use only tagless *enums*. In an enumerator list, the '=' construct shall not be used to explicitly initialize members.	
Background	Even if *enums* are visible to the debugger and more checks can be performed on this type it is not usable due to RAM-consumption. The RAM-consumption is up to the factor of four higher as compared to the u8 state variable (depending on compiler and microcontroller). When using an enumeration set together with another data type for the state variables, the *enum* shall not have a tag, because otherwise you will define a new data type, which is of no usage; plus, assigning a state to the state variable would cause a PC-Lint warning.	
Class of rule	A	
Verification	Code review, PC-Lint no. 641	
Example 1 (RAM consumption)	Bad	Good
	`enum {NC_STATE1, NC_STATE2} state;`	`#define NC_STATE1 0x00` `#define NC_STATE2 0x01` `u8 state;` `state = NC_STATE1;`
	The “bad” example is good in terms of code, but the state value will use the integer data type, which needs more RAM.	
Example 2 (enum usage)	Bad	Good
	`enum STATES {NC_STATE1, NC_STATE2};` `u8 state;` `state = NC_STATE1;`	`enum {NC_STATE1, NC_STATE2};` `u8 state;` `state = NC_STATE1;`
	In the “bad” example the *enum* has the tag STATES. This defines a new data type. An explicit type cast would be necessary when assigning the state NC_STATE1 to the variable state.	
Example 3 (use of initializers)	Bad	Good
	`enum { NC_BLA = 0x3A; NC_TOTO = 0xAA};` `u8 var;` `var = NC_BLA;`	`#define NC_BLA  0x3A` `#define NC_TOTO 0xAA` `u8 var;` `var = NC_BLA;`
	In the “bad” example an *enum* member is used to initialize an u8 variable. *Enums* are a special data type to contain enumerations only. They have nothing to do with the integer data types. In this case use *#define*. If you need to define state values it’s more useful to use *enum* type for definition of data sets together with u8 type (see example 2).	
Reference	MISRA guidelines, rule 32 Rule D.17	

D.25 Use *typedef* instead of name tags		
Description	*Typedef* is preferable to name tags	
Background	To improve readability and according to design choices, a precise type should be associated to each data. According to this recommendation and to increase portability, standard types could be redefined. Data encapsulation (using structure, union and enumeration) should be done according to the detailed design choices.	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`struct RECORD{` `  u32 key;` `  u8 value;` `};` `struct RECORD item0;`	`typedef struct {` `  u32 key;` `  u8 value;` `}RECORD;` `RECORD item0;`
	In the “bad” example the structure is not defined as a new type.	

D.26 Define bitfields for internal data only		
Description	Bitfields should be used if there are many internal flags inside a module. Bitfields should normally not be exported (use *flag* for interface data instead). Use only the modifier *unsigned* (or *signed* in special cases and only if size is 2 or more) to define members of a bitfield. Use dedicated names for the members, not only an enumeration.	
Note	2 methods were considered to be used: general data type with unnamed bits and a user defined bitfield with functional names. The version with the functional name provides a better error detection in case of the moving or removing of single bits. A bit inside a bitfield to interface with other functions is not a preferred solution since the implementation must be known.	
Class of rule	A	not relevant for µC accesses in compiler specific modules
Verification	Code review, PC-Lint no. 46, 806	
Example	Bad	Good
	`struct {` `  u8  lv_bit_0:1;` `  s16  lv_bit_1:1;` `  flag lv_bit_2:1;` `} bf;`	`struct {` `  unsigned lv_bla:1;` `  unsigned lv_toto:1;` `  unsigned lv_foo:1;` `} bf;`
	In the “bad” example there may occur problems using fix-size-data-types.	
Reference	MISRA guidelines, rule 111, 112	

D.27 Be aware of the correct usage of bitfields and structures

Description	Do not access the members of a bitfield without the bitfield access (e.g. by pointer or union mechanism). It is possible to define a "short name" by the C-preprocessor to a member of a bit field (e.g. `#define LV_FOO bf.lv_foo`), but beware of self referencing macros (e.g. `#define lv_foo bf.lv_foo`).	
Background	If a bitfield is accessed not with the bitfield members itself, but also with a pointer access / union then the code relies on the bit order which is not defined.	
Note	When defining a "short name", avoid recursive macros like "`#define lv_foo bf.lv_foo`". This could lead to an endless replacement: `bf.bf.bf.bf.…` All the members of a structure (or union) shall be named and shall only be accessed via their name.	
Class of rule	A+	
Verification	Code review, PC-Lint 531	
Example 1	Bad	Good
	`bf = 0x0003;` *or:* `((u16)bf) = 3;`	`bf.lv_bla = 0;` `bf.lv_toto = 1;` `bf.lv_foo = 1;`
	In the "bad" example the execution (if even accepted by the compiler) is dependent on the internal representation of the bitfield.	
Example 2	Bad	Good
	`bf = 0;` *or:* `((u16)bf) = 0;`	`const bf_type bf_0 = {0,0,0,0,0};` `bf = bf_0;`
	Even setting the bitfield to zero may be compiler dependent. Just think about a bitfield with 33 members. Prefer a well defined constant bitfield instead.	
Reference	ISO/IEC 9899 section 6.7.2.1 §10 MISRA guidelines, rule 113	

D.28 Use *unions* for exclusive use of the members only

Description	Use *unions* for exclusive use of the members only. Do not use the side effects. Use standard mechanism only to read parts of a word. They are implemented compiler dependent, but their use is compiler independent.	
Background	The use of unions to rearrange the byte order or read only parts of a word is not portable due to the different endians of the processors.	
Class of rule	A+	B for non-portable code
Verification	Code review	
Example	Bad	Good
	`union {` `  u8  byte[2];` `  u16 word;` `} var;` `var.word = 0x1234;` `if (var.byte[0] == 0x12) …`	`u16 var = 0x1234;` `if (u8_get_u16_u8(var, 1) == 0x12) …`
	In the “bad” example the execution of the if-clause is dependent on the byte ordering of the compiler / processor.	
Reference	BITOP Kernighan / Ritchie section 6.8 MISRA guidelines, rule 110	

D.29 Distinguish between array and pointer

Description	*[ ]* and * are different types in C even if they can be mixed up easily. Use arrays as arrays and pointers as pointers only.	
Background	Possible illegal access to memory.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 545, 663	
Example	Bad	Good
	`u8 *ptr = "ABC";` `u8 a[]  = "DEF";` `*ptr = 'X'; /* writing to ROM ! */` `a = "XYZ";  /* forbidden (array is a constant address) */`	`u8 *ptr = "ABC";` `u8 a[]  = "DEF";` `a[0] = 'X';     /* -> "XEF" */` `ptr = "XYZ";  /* new address */` `/* assigned to ptr, "ABC" is */` `/* unchanged in memory */`
	In both examples “ptr” points to a constant expression “ABC” in ROM, whereas “a” is a constant pointer to an array in RAM with initial value “DEF” in ROM. In the “bad” example writing ‘X’ to a ROM area will fail (not necessary in an emulator environment, if area is configured writeable). The address represented by “a” is not changeable. This statement will be rejected by the compiler. In the “good” example the members of the array “a” are copied to RAM during initialization. This RAM area is changeable. The pointer “ptr” can be redirected to another address in ROM where “XYZ” is stored.	

D.30 *removed*

D.31 Braces must be used to indicate and match the structure in the non-zero initialization of arrays and structures

Description	Adding braces to indicate the structure of the data increases the readability and avoids incorrect initialization.	
Class of rule	A	
Verification	PC-Lint no. 940	
Example	Bad	Good
	`u8 any_var[3][2] = {1,2,3,4,5,6};`	`u8 any_var[3][2] = {{1,2},{3,4},{5,6}};`
	The braces in the good example clearly show the grouping of data and prevent accidental incorrect initialization.	
Reference	MISRA guidelines, rule 31	

D.32 Non-constant pointers to functions must not be used

Description	"Non-constant" means pointers whose value is calculated at run-time. If the value of a pointer is known at compile time, either because it is a constant or because it is copied from some constant value or look-up table, then such use is permitted.	
Background	There is a danger with pointers calculated at run time that an error will cause the pointer to point to an arbitrary location.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`void foo (void) { a = b; c = d; }` `if (lv) ptr = foo; else ptr = foo + 10;` `(*ptr)();`	`void foo_2 (void) { c = d; }` `void foo_1 (void) { a = b; foo_2(); }` `if (lv) ptr = foo_1; else ptr = foo_2;` `(*ptr)();`
	The "bad" example may work on one certain compiler, but if the assembler code changes due to any other compilement the program will not work any more.	
Reference	MISRA guidelines, rule 104	

D.33 Do not use floating point

Description	Floating point variables and calculation should not be used. Use fixed point implementation (supported by the Mathlib functions) instead.	
Background	The use of floating point may reduce portability (due to non-standard conformant HW implementation) and increase task switching overhead (a special OS and interrupt configuration is needed to save additional registers). Numerical behavior of floating point calculations frequently leads to mistakes (accumulation of errors, accuracy problems, …)	
Note	If required, floating point may be used at the interface to customer software or third party software.	
Class of rule	A	B for using floating point for compile time calculations
Verification	Code review, PC-Lint 777, 960	
Example 1	Bad	Good
	`f32 a = (f32)0x3FFF0000;` `while (a < (f32)0x40000000) {` `  /* do something */` `  a += 1.0;` `}`	`s32 a = (s32)0x3FFF0000;` `while (a < (s32)0x40000000) {` `  /* do something */` `  a += 1;` `}`
	The left example loops forever (in this numerical range the precision of f32 is already smaller than one, so adding one does not change its value).	
Example 2	Bad	Good
	`s16 a = (s16)10000;` `s16 b;` `…` `b = (s16) ( (f32) a * (f32) 0.375 )` `…`	`s16 a= (s16)10000;` `s16 b;` `…` `b = s16_mul_s16_u8_fak05(a, 192)` `…`
	In the "bad" example it is not very efficient to convert the fixed point data to floating point, to do the multiplication and to convert it back.	
Reference	MISRA guidelines, rule 15, 16, 50, 65	

D.34 Do not use incomplete types in structures and unions

Description	In the definition of a structure or union type, all members of the structure or union shall be fully specified.	
Background	An incomplete type is a type of unknown size.	
Class of rule	A	
Verification	PC-Lint no. 43	
Example	Bad	Good
	`extern struct {` `  u8 b[];` `  u8 c;` `} a;` `extern u8 d[];`	`extern struct {` `  u8 b[NC_SIZE_B];` `  u8 c;` `} a;` `extern u8 d[];`
	The compiler can not determine the offset of "c" in the structure "a" of the "bad" example. In the "good" example the size of each struct-member is known and every offset can be calculated. For the array "d" it is not necessary to know the size for referencing. Although a given size would increase the static checkability.	
Reference	MISRA guidelines, rule 108	

D.35 Avoid overlapping data storage

Description	Avoid overlapping data storage. When data is accessed from overlapped memory (e.g. unions, buffers) it has to be explained by a comment why it is ensured that the memory contains the information wanted.	
Background	This rule refers to the technique of using memory to store some data, then using the same memory to store some other data at some other time during the execution of the program. This practice is not recommended as it brings with it a number of dangers. For example: a program might try to access data of one type from the location when actually it is storing a value of the other type (e.g. due to an interrupt); the two types of data may align differently in the storage, and encroach upon other data; data may not be correctly initialized every time the usage switches.	
Class of rule	A+	
Verification	Code review	
Example 1	Bad	Good
(Avoidance)	`void fct(void) {` `  u16 tmp;` `  tmp = n + n_dif;` `  n_new = tmp;` `  if (tqi_sp > tqi_av)` `  tmp = tqi_sp;` `  else` `  tmp = tqi_av;` `  tqi_sp2 = tmp;` `}`	`void fct(void) {` `  u16 tmp_n;` `  u16 tmp_tqi;` `  tmp_n = n + n_dif;` `  n_new = tmp_n;` `  if (tqi_sp > tqi_av)` `  tmp_tqi = tqi_sp;` `  else` `  tmp_tqi = tqi_av;` `  tqi_sp2 = tmp_tqi;` `}`
	Especially when using X-Tools-Contractions in the "bad" example it is not easy to determine the current content of "tmp". When using different local variables their contents are more obvious. A good compiler should optimize the register usage for non-overlapping use of the variables.	
Example 2	Good	
(Documentation)	`void fct(void) {` `  /* buffer size is big enough for E2P and CAN */` `  u16 buffer[NC_BUFFER_SIZE];` `  /* Here buffer is used for E2P writing */` `  LIB_MEM_Set(buffer, 0, NC_BUFFER_SIZE);` `  TRL_E2P_Write(buffer, NC_BUFFER_SIZE);` `  /* buffer is not needed for E2P anymore */` `  …` `  /* Here buffer is used for CAN Comunication */` `  TRL_CAN_Tx(HANDLE1, buffer);` `  TRL_CAN_Tx(HANDLE2, buffer);` `  /* buffer is not needed for CAN anymore */` `}`	
	For efficiency reasons "buffer" is used for different purposes. The validity of the contents of "buffer" is documented in the code. No further access should be made to "buffer" expecting a certain value. To increase robustness a semaphore could be introduced.	
Reference	MISRA guidelines, rule 109	

2.4 Operators and Expressions

O.1 Prefer range checking to equality checking		
Description	When checking conditions, in particular the end condition of a loop, check the range with "<", ">", "<=" and ">=" instead of checking the equality with "==" or "!=".	
Background	The robustness of the software should be increased, especially by avoiding endless loops.	
Class of rule	B+	
Verification	Code review	
Example	Bad	Good
	`unsigned int loop_ctr;` `loop_ctr = 0;` `do` `{` `  …` `  loop_ctr++;` `}` `while ( loop_ctr != 100 );`	`unsigned int loop_ctr;` `loop_ctr = 0;` `do` `{` `  …` `  loop_ctr++;` `}` `while ( loop_ctr < 100 );`
	Within the "bad" example the robustness of the code is higher, if the loop is also stopped at each counter value above 100 (if the value 100 is missed due to whatever reason).	

O.2 Clarify operator priority by using parentheses		
Description	The priority of operators should be shown by using parenthesis, especially within complex expressions. However do not add too many parentheses so as to clutter the code and make it unreadable.	
Background	Although the priority of operators is clearly defined, it is better to clarify the priority explicitly by parenthesis. This improves readability and forces the developer to be aware about the order of execution.	
Class of rule	B	
Verification	Code review, PC-Lint no. 834, 961	
Example 1	Bad	Good
	`if (lv_a && lv_b && b * c + d == 2)` `  …`	`if (lv_a` ` && lv_b` ` && ((b * c) + d) == 2)` `)` `  …`
	In the “bad” example the order of evaluation can not be seen at first glance.	
Example 2	Bad	Good
	`x = ((a + b) – c) + d;`	`x = a + b – c + d;`
	All operators have equal precedence. Therefore parentheses are not required, but they are cluttering the code.	
Reference	MISRA guidelines, rule 47	

O.3 Do not use statements which depend on the order of evaluation or on logical shortcuts

Description	Every statement has to be written in a way that does not depend on the order of evaluation. Use only functions in an expression, which don't modify global resources. Functions, which perform an action, are not allowed to be used in expressions with a shortcut.	
Background	Special care has to be taken for functions with actions behind it. Functions that modify global variables and return a result are potentially problematic. A caller of the function has to know whether a function performs an action (e.g. modification of global variables) or not. This can be described in function documentation or by a prefix/postfix of the function name.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 564, 960	
Example 1 (order of evaluation)	Bad	Good
	`x = b[i] + i++;`	`x = b[i] + i;` `i++;`
	In the "bad" example it is not clear, what the value of i is, when accessing b[i]. It depends on the order of evaluation. Therefore in the good example i is incremented within a separate statement.	
Example 2 (order of evaluation)	Bad	Good
	`u8 a[10] = {0,1,2,3,4,5,6,7,8,9};` `u8 b;` `u8 index = 4;` `u8 c_f1(void) {` `  return a[index];` `}` `u8 c_f2(void) {` `  index++;` `  return a[index];` `}` `void c_f3(void) {` `  b = c_f1() + c_f2();` `}`	`u8 a[10] = {0,1,2,3,4,5,6,7,8,9};` `u8 b;` `u8 index = 4;` `u8 c_f1(void) {` `  return a[index];` `}` `u8 c_f2(void) {` `  return a[index+1];` `}` `void c_f3(void) {` `  b = c_f1() + c_f2();` `}`
	In the "bad" example the result b is not defined, it depends on the order of evaluation. The result might be 9 or 10. In the "good" example no global variable is modified within function c_f2(). The result of the calculation is well defined (b = 9).	

<table>
<tr><td>Example 3
(shortcut due to optimiza-tion)</td><td>Bad</td><td>Good</td></tr>
<tr><td></td><td><pre>
/* Element 5 has to be changed */

u8   index;
flag lv_a;
u8   b[5];

flag foo1(void) {
   index++;
   return 1;
}

void foo2(void) {

   index = 3;

   if (lv_a && foo1())
   { … }
   b[index] = 0xFF;
}
</pre></td><td><pre>
/* Element 5 has to be changed */

u8   index;
flag lv_a;
u8   b[5];

flag foo1(void) {
   index++;
   return 1;
}

void foo2(void) {

   flag lv_tmp;

   index = 3;
   lv_tmp = fool();
   if (lv_a && lv_tmp)
   { … }
   b[index] = 0xFF;
}
</pre></td></tr>
<tr><td></td><td colspan="2">In the “bad” example the function call foo1() is not evaluated if lv_a == 0. So the value of index is not incremented and element 4 is changed.</td></tr>
<tr><td>Reference</td><td colspan="2">MISRA guidelines, rule 33, 46</td></tr>
</table>

O.4 Use the *?* operator only in special cases

Description	Use the *?* operator only if the program becomes faster and/or smaller. The used expressions should be less complex in that case. In standard cases and for more complex expressions, use the *if*-instruction instead.	
Background	The readability and maintainability of the code decreases significantly, when using complex expressions. However the use of *?* could decrease memory consumption and runtime and could be effective in special cases, using simple expressions.	
Class of rule	B	
Verification	Code review	
Example 1	Bad	Good
	`e = (d && (a > (b * c))) ? a + c : b;`	`if (d && ( a > (b * c)))` `{` `  e = a + c;` `}` `else` `{` `  e = b;` `}`
	In the “bad” example the readability is not acceptable. Furthermore the use of *?* has no advantage. The *if*-instruction should be used in such cases.	
Example 2	Bad	Good
	`u16 tmp_n_min;` `if (lv_at)` `  tmp_n_min = c_n_min_at;` `else` `  tmp_n_min = c_n_min_mt;` `if (lv_ena` `  && (n > tmp_n_min)` `)` `...`	`if (lv_ena` `  && (n > (lv_at ? c_n_min_at` `  : c_n_min_mt)` `)` `...`
	In "bad" example, the implementation is quite ineffective. In this case it makes sense to use the *?* operator.	

O.5 Avoid dangling references

Description	Pointers are not allowed to point to variables of a shorter lifetime, so that undefined pointers (dangling references) may not occur.	
Background	This problem mainly arises, when addressing temporary data of functions. It is possible to pass the address of such data to sub functions, however it is not advisable to return the address or to store it globally.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 413, 613, 674, 733, 789, 794, 809	
Example	Bad	Good
	`/* outer block */` `u32 *ptr;` `/* inner block */` `{` `  u32 var = 10;` `  ptr = &var;` `}` `/* var "perishes" before ptr; the value is now undefined * /` `func(*ptr);`	`/* outer block */` `u32 *ptr;` `u32 var = 10;` `/* inner block */` `{` `  ptr = &var;` `}` `/* Now, var is still defined */` `func(*ptr);`
	Here, the variable *var* should have been declared in the outer block	
Reference	MISRA guidelines, rule 4, 106	

O.6 Use library functions for shift and inversion

Description	Use macros or library functions instead of C operations, which could have a compiler specific behavior, even in immediate expressions.	
Background	For writing portable code the used C operators must have the same behavior on several platforms. Due to the fact that some operations result in an undefined behavior the operations have to be replaced by macros, which ensure a well defined platform independent behavior.	
Note	The implementation of the functions has to be in a way that the compiler can expand them at compile time (e.g. as macros). Also for ISO conform operations it makes sense to use library functions, e.g. to gain a special overflow behavior.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 572, 701, 702	
Example (operators affected)	Operators which <u>have to</u> be replaced by a macro or library function	Operators which may also be used directly.
	`~ (Inv)` `<< >> (Shift)`	`/ (div – take care of div. By 0)` `* (multiplication)` `% (mod – take care of mod by 0, only for positive numbers)` `& | ^ (bitwise operations)` `&& || ! (logical operations)` `-- ++ (increment, decrement)` `?: (conditional operation)` `= += -= (addition, subtraction etc.)` `== >= != (comparison operators)` `-> . ( pointer, struct operators)`
	Never use operations which are compiler specific and which do not follow the ISO C standard.	
Example (using library functions)	Bad	Good
	`u8 value_8;` `u16 value_16;` `u16 result_16;` `result_16 = u16_add_u16_u16(` `value_16,` `value_8 << 8 );`	`u8 value_8;` `u16 value_16;` `u16 result_16;` `result_16 = u16_add_u16_u16(` `value_16,` `u16_lsl_u16_u8(value_8, 8) );`
	In the “bad” example the content of value_8 might be lost. An explicit typecast to u16 before shift is necessary.	
Reference	gdatatypes.doc LPRC, BITOP MISRA guidelines, rule 37, 38	

O.7 *removed*

O.8 Do not use shift operations for multiplication and division

Description	Always use mathematical functions (*, /) for multiplication and division with power of 2.	
Background	The shift operator shall not be used for multiplication and division with power of 2. There is no advantage in runtime using the shift operation due to compiler optimization. Plus, one may get in trouble when applying the shift operation to signed values.	
Note	Use shift operations ONLY on data that is interpreted as a bit pattern, having no resolution nor a physical meaning.	
Class of rule	A	B for compiler dependent software
Verification	Code review	
Example	Bad	Good
	`u16 value_16;` `u16 result_16;` `result_16 = value_16 >> 2;`	`u16 value_16;` `u16 result_16;` `result_16 = value_16 / 4;`
	In the “bad” example the readability is worse than in the good example. Moreover, if in a future version the value_16 may have to be divided by a value, which is not a power of 2, the expression will have to be changed completely.	

O.9 Perform bitwise operations only with unsigned data of same size

Description	Perform bitwise operations and comparisons only with unsigned data of the same size. Do not use the *flag* data type for bitwise operations.	
Background	Some compilers do not follow the standard for logical operations.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 701, 702	
Example	Bad	Good
	`void c_xxx(void){` `  s8  tmp_s8  = (s8)0xFF;` `  u16  tmp_u16  = 0xED;` `  if ((tmp_u16 ^ tmp_s8) == 0x12)` `  {` `  …` `  }` `}`	`void c_xxx(void){` `  u8  tmp_u8  = 0xFF;` `  u16  tmp_u16  = 0xED;` `  if ((tmp_u16 ^ (u16)tmp_u8)==0x12)` `  {` `  …` `  }` `}`
	In the “bad” example the if-condition may not be fulfilled if the operands are automatically promoted to int type before the operation.	
Reference	MISRA guidelines, rule 37	

O.10 Perform only allowed operations on the *flag* data type		
Description	Never use bitwise or arithmetic operations on *flag* data types – use logical operations only. Do not compare *flag* data types to 0 or 1 explicitly; use logical check in C language.	
Background	The *flag* data type is defined implementation specific (e.g. 1 bit or 8 bit).	
Class of rule	A	
Verification	Code review, PC-Lint no. 720	
Example	Bad	Good
	`flag lv_x;` `flag lv_y;` `void c_xxx(void){` `  if ( lv_x & lv_y)  { … }` `  if ( lv_x == 1 )  { … }` `  if ( lv_x == 0 )  { … }` `}`	`flag lv_x;` `flag lv_y;` `void c_xxx(void){` `  if ( lv_x && lv_y )  { … }` `  if ( lv_x )  { … }` `  if ( !lv_x )  { … }` `}`
	In the “bad” example bitwise operations are used to combine logical information. Flags should be checked only like in the “good” example.	
Reference	MISRA guidelines, rule 36, 49 Rule D.5	

O.11 Use library functions to handle overflow limitations		
Description	Always use the appropriate library if saturated math is needed.	
Background	A library function is required for overflow limitation (e.g. s16_add_s16_s16() function). This library shall cover all data types and conversions between them.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 648	
Example	Bad	Good
	`u16  value_1;` `u16  value_2;` `u16  result;` `result = value_1 + value_2;`	`u16  value_1;` `u16  value_2;` `u16  result;` `result =` `u16_add_u16_u16(value_1,value_2);`
	In the “bad” example an overflow is produced without saturation.	
Reference	ECMATH MISRA guidelines, rule 4	

O.12 Do not use definitions or functions of C standard libraries		
Description	Do not use standard library functions. Only use SV libraries that are verified and under VM control.	
Background	Undefined behavior of standard library functions shall be avoided.	
Note	The term "Standard Libraries" refers to ISO/IEC 9899, chapter 7	
Class of rule	A+	
Verification	PC-Lint no. 421, 829	
Example	Bad	Good
	`s16  res;` `s16  value;` `value = (s16)-32768;` `res = abs(value);`	`s16  res;` `s16  value;` `value = (s16)-32768;` `res = s16_abs_s16(value);`
	In the “bad” example the result of abs() is undefined, because the biggest result of abs() is 32767 (if the *int* data type has the size of 16 bit).	
Reference	MISRA guidelines, rule 116-127	

O.13 Check for division by 0, if required		
Description	Never perform divisions by zero. If you can't make absolutely sure, that the devisor never gets zero, do a runtime check before the operation or use mathematical library function (recommended).	
Background	The division by zero and the modulo by zero are not defined. Undefined operations, which happen occasionally (even after days or months of correct operation) are very critical and must be avoided under all circumstances.	
Note	The check at runtime only makes sense, if the occurrence of the value 0 can not be avoided at all by software design. In that cases a special treatment for the value 0 is required.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 414, 795	
Example	Bad	Good
	`u16 value_1;` `u16 value_2;` `u16 result;` `void c_xxx(void){` `  result = value_1 / value_2;` `}`	`u16 value_1;` `u16 value_2;` `u16 result;` `void c_xxx(void){` `  if (value_2 > 0)` `  {` `  result = value_1 / value_2;` `  }` `  else` `  {` `  result = (u16)0xFFFF;` `  }` `}`
	In the “bad” example the behavior is not defined. There might be an exception / undefined result. The good example shows, that a special treatment is necessary for value_2, if it is equal to zero.	
Reference	MISRA guidelines, rule 4	

O.14 Avoid pointer access to objects, which do not belong together

Description	Don't rely on the order of definition when accessing data via pointer. Combine objects to a single object (array/structure), if you need to access them with pointers. Relational operators shall not be applied to pointer types except where both operands are of the same type and point to the same array, structure or union.	
Background	Objects are not necessarily located in the order they are defined.	
Note	Processor dependant code that directly accesses memory by address can treat the memory as a plain array.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 946	
Example	Bad	Good
	`u32 a;` `u32 b;` `void f(void){` `  u32 *ptr = &a;` `  ptr++;` `  /* modify b? */` `  *ptr = 0;` `}`	`u32 a[2];` `void f(void){` `  u32 *ptr = &a[0];` `  ptr++;` `  /* modify a[1] */` `  *ptr = 0;` `}`
	The "bad" example defines two objects a and b which might not be located in a defined order.	
Reference	MISRA guidelines, rule 103	

O.15 *removed*

O.16 *removed*

O.17 The NULL pointer shall not be de-referenced

Description	De-referencing the NULL pointer (trying to access the object the NULL pointer points to) is not allowed. When a function returns a pointer, which could possibly be the NULL pointer, and this pointer is subsequently de-referenced, the program should first check that the pointer is not NULL.	
Background	The NULL pointer is explicitly used to denote an invalid pointer, not pointing to a valid object. Trying to dereference the NULL pointer might lead to undefined behavior or HW exceptions on some platforms.	
Note	Usually the NULL pointer is implemented as a pointer to address 0. On some platforms this might be a valid pointer. In this case by appropriate methods (e.g. modifying the memory mapping) it has to be ensured that no object is located at address 0 (near or far). Otherwise the NULL pointer can no longer be used to designate an invalid pointer, as it cannot be distinguished from a valid one.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 413, 613, 794	
Example	Bad	Good
	`u8* p_first_char;` `…` `p_first_char = GetReadBuffer();` `if (*p_first_char == 0x00) {` `  … /* a message starting with a zero byte*/` `}` `else {` `  … /* a normal message */` `}`	`u8* p_first_char;` `…` `p_first_char = GetReadBuffer();` `if (p_first_char != NULL){` `  if (*p_first_char == 0x00){` `  … /* a message starting with a zero byte*/` `  }` `  else {` `  … /* a normal message */` `  }` `}` `else {` `  … /* we got a NULL pointer */` `}`
	The function may return a NULL pointer. Before dereferencing the pointer it should be checked for being a NULL pointer (in any case a correct error handling should be provided).	
Reference	MISRA guidelines, rule 4, 107	

O.18 A stub via #define to a variable must always be casted		
Description	If a variable has to be stubbed to a different variable, an explicit cast is mandatory in any case.	
Background	If the stubbed variable is implemented later, the existence of the stub might be overlooked and thus the existing variable will be defined twice, causing undefined behavior. Adding the cast will cause a syntax error in this case and thus safely prevent problems.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	stub.h `#define sf_not_avail sf_avail` file.c `u8 sf_not_avail;` This will be resolved by preprocessor to: `u8 sf_avail;` This is a second definition for "sf_avail"!	stub.h `#define sf_not_avail (u8)sf_avail` file.c `u8 sf_not_avail;` This will be resolved by preprocessor to: `u8 (u8)sf_avail;` This is invalid syntax.
	In the good example it is not possible to end up with an accidental double definition as the illegal syntax prevents compilation of the code.	

O.19 The comma operator must not be used		
Description	Using the comma operator decreases the readability of the code and the same effect can be reached by other means. As this might not be possible within macros, deviations from this rule may be made within macro definitions.	
Background	Best readability	
Class of rule	A	B within macro definitions
Verification	Code review, PC-Lint no. 147, 960 (partly covered)	
Example 1	Bad	Good
	`lv_a = lv_b,` `lv_c = lv_d;`	`lv_a = lv_b;` `lv_c = lv_d;`
	The comma operator can be easily replaced increasing readability. Furthermore in the bad example there are two assignments in one statement.	
Example 2	Bad	Good
	`f (1, (g(&t) , t));`	`g(&t); f(1, t);`
	The comma operator can be easily replaced increasing readability.	
Reference	MISRA guidelines, rule 42 Rule O.20	

O.20 An assignment shall be a single statement		
Description	Assignment operators should not be used inside expressions, but should be written as a separate statement.	
Background	This helps to avoid getting "=" and "==" confused and increases static checkability.	
Class of rule	A	B within macro definitions
Verification	PC-Lint no. 720, 820	
Example	Bad	Good
	`if (lv_cdn_1 = lv_cdn_2) {` `  …` `}` `if (tqi_sp = tqi_sp_new) {` `  …` `}`	`lv_cdn_1 = lv_cdn_2;` `if (lv_cdn_2) {` `  …` `}` `tqi_sp = tqi_sp_new;` `if (tqi_sp != 0) {` `  …` `}`
	Both sides are functionally identical but in the bad example it is not clear whether the second "=" was omitted accidentally or not.	
Reference	MISRA guidelines, rule 35	

O.21 Operands of && and || shall be primary expressions		
Description	The operands of && and || shall either be a single identifier, a constant, a function call, or a parenthesized expression. All operands shall be of logical type (flag). Chaining of logical operators of the same type is allowed without additional parentheses.	
Background	Increases readability and ensures that the expression evaluates as it is intended by the programmer.	
Note	This rule represents a special case of rule O.2. But whereas O.2 is of class B, rule O.21 is of class A!	
Class of rule	A	
Verification	PC-Lint no. 720	
Example	Bad	Good
	`flag f();` `flag lv_x;` `flag lv_y;` `if (lv_x == lv_y && f()) {` `  …` `}`	`extern flag f(void);` `flag lv_x;` `flag lv_y;` `if ((lv_x == lv_y) && f()) {` `  …` `}`
	The bad example could easily be misunderstood as `if (lv_x == (lv_y && f())`	
Reference	MISRA guidelines, rule 34 Rule O.2	

O.22 Tests of a value against zero should be made explicit, unless the operand is effectively Boolean		
Description	Testing data values, which are not Boolean, should be explicitly written in order to distinguish them from Boolean expressions.	
Background	Increases readability	
Note	This rule is somehow the complementary rule to O.10, where it is stated, that for Boolean values no explicit comparison should be used.	
Class of rule	A	
Verification	Code review, PC-Lint no. 720	
Example	Bad	Good
	`u8 state;` `if (state) {` `  …` `}`	`u8 state;` `if (state != 0) {` `  …` `}`
	In the good example it is clear what is wanted to be done with "state".	
Reference	MISRA guidelines, rule 49, Rule O.10	

2.5 Statements and Control Structures

S.1 Do not access directly processor status flags		
Description	Use standard functions (of math-library) or comparisons instead	
Background	Direct access to processor status flags (like carry, zero) is not portable (processor and compiler specific).	
Class of rule	A	Only valid for portable code
Verification	Code review	
Example	Bad	Good
	`u32 fuel_tot_cps;` `u32 fuel_cps;` `...` `fuel_tot_cps =` `  fuel_tot_cps + fuel_cps;` `if( C ) fuel_tot_cps =` `  (u32)0xFFFFFFFF;`	`u32 fuel_tot_cps;` `u32 fuel_cps;` `...` `fuel_tot_cps = u32_add_u32_u32(` `  fuel_tot_cps,` `  fuel_cps);`
	The “bad” example is faster but not portable. With a math routine an indirection is created, which makes the code independent.	

S.2 Do not access directly processor peripherals		
Description	Use provided services of the respective layer instead.	
Background	Direct access to processor peripherals is not portable.	
Class of rule	A	Only valid for portable code
Verification	Code review	
Example	Bad	Good
	`MIOS1.PWM2PULR.R = 0x3000;`	`BIOS_PWM_Dutycycle(2, 0x3000);`
	The code of the “bad” example only works on a MPC555.	

S.3 Do not use assembly language		
Description	Use a library / support function to achieve the functionality you want.	
Background	Direct access to processor instructions is not portable.	
Note	Encapsulate ASM instructions as far as possible	
Class of rule	A	B for non-portable code
Verification	Code review	
Example	Bad	Good
	`asm(" mtspr  eie,0");`	`LIB_PRC_EnableInt();`
	The “bad” example only works on PowerPC microcontrollers.	
Reference	MISRA guidelines, rule 2, 3	

S.4 Do not implement interrupts directly		
Description	Use provided services of the BIOS layer or OS.	
Background	Direct use of interrupts is not portable.	
Class of rule	A	Only valid for portable code
Verification	Code review	
Example	Bad	Good
	`#pragma interrupt externIntHandler` `void externIntHandler(void)` `{` `  …` `}`	`extern struct SIG_definition sig_int;` `BIOS_INT_SetHandler(0, &sig_int);` `void sig_int_fct(struct SIG_definition *sig){` `  …` `}`
	The “bad” example is both compiler dependent and controller dependent.	

S.5 The statement *goto* must not be used		
Description	In general the use of the statement *goto* is forbidden. The *goto* statement could be used for error handling if it makes the code more readable than nested *if* statements or complicated conditions. In this case it must be well documented.	
Background	The use of *goto* can make the code quite unreadable.	
Note	Labels should not be used, except in switch statements or to mark specific positions within the code for the debugger.	
Class of rule	A	Exception: *goto* for error handling
Verification	PC-Lint no. 801, 961	
Example	Bad	Good
	`int n = 0;` `loop:` `n++;` `if (n < 10) goto loop;`	`int n = 0;` `do {` `  n++;` `} while (n < 10);`
	The good example avoids the use of *goto* by using a standard loop construct.	
Reference	MISRA guidelines, rule 55, 56	

S.6 The macros *setjmp()*, *longjmp()* must not be used		
Description	The use of the macro pair *setjmp(), longjmp()* is forbidden.	
Background	By using the *longjmp()* a jump into a total different environment can be done. The behavior is hardly predictable.	
Class of rule	A	
Verification	PC-Lint no. 421	
Example	Bad	Good
	`jmp_buf recover;` `void f(void){` `  setjmp(recover);` `}` `void f2(void){` `  if (dramatic_error) {` `  longjmp(recover, 1);` `  }` `}`	`void f2(void){` `  if (dramatic_error) {` `  LIB_PRC_Reset();` `  }` `}`
Reference	MISRA guidelines, rule 122 Rule O.12	

S.7 Do not use the *continue* statement		
Description	The *continue* statement is an unstructured jump to the end of the loop. There is no means to describe this in the “Nassi-Shneiderman” diagram.	
Background	The use of *continue* can make the code quite unreadable and hard to maintain.	
Class of rule	B	
Verification	PC-Lint no. 960	
Example	Bad	Good
	`while (n < 10) {` `  ...` `  if (lv_cond) continue;` `  ...` `}`	`while (n < 10) {` `  ...` `  if (!lv_cond){` `  ...` `  }` `}`
	If in the bad example the continue statement is embedded in a long sequence of code it might not be clear to the reader that the part following the continue statement is only executed conditionally.	
Reference	MISRA guidelines, rule 57	

S.8 Use the *break* statement only in *switch* statements		
Description	Use the *break* statement only in a *switch* statement but not to terminate a loop.	
Background	The use of *break* can make the code quite unreadable and hard to maintain.	
Class of rule	B	
Verification	PC-Lint no. 960	
Example	Bad	Good
	`while (n < 10) {` `  if (lv_cond) break;` `  ...` `}`	`while ((n < 10) && (!lv_cond)) {` `  ...` `}`
	The good example avoids the use of the break by changing the conditions for the loop.	
Reference	MISRA guidelines, rule 58	

S.9 Avoid statements without effect		
Description	Statements like '*return*;' at the end of a void function, or *'a + b;*', which have no purpose or side effect, must not occur.	
Background	Unused code bloats the code up without any added value.	
Note	In processor dependent software it is sometimes necessary to have a read access to a *volatile* variable there it makes sense to have a statement like *'(volatile u16) a;'*	
Class of rule	A	
Verification	PC-Lint no. 505, 506, 522, 527, 681, 827	
Example	Bad	
	`void f(void)` `{` `/*1*/  if (1)` `/*2*/  { a = c; }` `/*3*/  ;` `/*4*/  a+b;` `/*5*/  if (0)` `/*6*/  { a = b; }` `/*7*/  e = 3;` `/*8*/  e = 5;` `/*9*/  return;` `/*10*/ d = b + c;` `}`	
	The statements in lines 1, 3, 4, 6, 7 and 10 in function f() do not have any effect. The statement in line 7 does not have an effect in a non-preemptive environment because it is overwritten by the statement in line 8. The "return" in line 9 has an effect, because it is not at the end of the void-function.	
Reference	MISRA guidelines, rule 52, 53 Rule G.5, S.18	

S.10 Avoid potential endless loops		
Description	If the original condition would allow for a potential endless loop, you may consider adding an additional condition to exit the loop. If you add an additional exit condition to avoid an endless loop, make sure that a proper error handling is available.	
Background	External events may not occur when expected.	
Class of rule	B+	
Verification	Code review	
Example	Bad	Good
	`u8 x[MAX];` `ptr = &x[0];` `while (*ptr < 0xFF) {` `  ...` `  ptr++;` `}`	`u8 x[MAX];` `ptr = &x[0];` `while ((*ptr < 0xFF) &&` `       (!lv_exit)) {` `  ...` `  ptr++;` `  if (ptr > &x[MAX]) {` `    /* we have reached the end of` `     the array x and have to exit` `     the loop */` `    lv_exit = 1;` `  }` `}` `if (lv_exit) ...`
	In the "bad" example it could be that there is no 0xFF in the array x to terminate the loop. The “good” example does only read the proper memory cells even if no 0xFF is found. An error handling should be done.	
Reference	MISRA guidelines, rule 4	

S.11 Check array indexes

Description	Indexes on arrays must be checked during coding phase and reviewing to ensure that index value points do not exceed the array range. If it can not be guaranteed by any other means (e.g. code review, static code check, integration review), generated indexes have to be checked in the code before accessing an array.	
Background	Reading or writing of other memory cells can cause unpredictable effects.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 415, 416, 428, 661, 676, 796, 797	
Example 1 (checked during run-time)	Bad	Good
	`#define NC_A_SIZE 10` `u32 a[NC_A_SIZE];` `void func(unsigned int idx)` `{` `  a[idx] = 34;` `}`	`#define NC_A_SIZE 10` `u32 a[NC_A_SIZE];` `void func(unsigned int idx)` `{` `  if (idx < NC_A_SIZE)` `  a[idx] = 34;` `}`
	In the “bad” example calling func() with idx >= 10 will overwrite other data in memory. This is avoided by index checking in the “good” example.	
Example 2 (checked in code review)	Bad	Good
	`u32 v[10];` `v[10] = 34;`	`u32 v[11];` `v[10] = 34;`
	The access to v[10] accesses the 11th element of the array, which is not defined in the “bad” example.	
Reference	MISRA guidelines, rule 4	

S.12 *For*-loop control variables must only be modified in the third condition; only expressions concerned with loop control may appear within a *for* statement		
Description	*For*-loop control variables (and only those) shall be modified only in the third condition of the *for* statement, not in the second condition or in the loop body. The expressions within the *for* statement must not have any other side effects than loop control.	
Background	Maintainability and robustness of the code.	
Class of rule	A	
Verification	Code review	
Example 1	Bad	Good
	`for( i = 1; i++ < 10; )` `{` `  a = a * i;` `  i++;` `}`	`for( i = 2; i < 11; i += 2 )` `{` `  a = a * i;` `}`
	In the “bad” example the *for*-loop control variable is modified in the second condition and in the loop body. The boundaries and increment are not clear to the reader.	
Example 2	Bad	Good
	`for( i = 1; i < 10; a = a * i)` `{` `  i++;` `}`	`for( i = 1; i < 10; i++ )` `{` `  a = a * i;` `}`
	In the “bad” example the *for*-statement contains side effects and the boundaries are not clear to the reader.	
Reference	MISRA guidelines, rule 66, 67	

S.13 The boundaries of a *for*-loop have to be fixed		
Description	The boundaries of a *for*-loop have to be fixed. They must not be modified by the body of the structure.	
Background	Maintainability of the code.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`u32 max_loop = 100;` `for( i = 0; i < max_loop; i++)` `{` `  ...` `  max_loop = max_loop / 2;` `  ...` `}`	`#define NC_MAX_LOOP 5` `for( i = 0; i < NC_MAX_LOOP; i++)` `{` `  ...` `}`
	In the “bad” example the upper boundary of the loop is changed in the loop body.	
Reference	MISRA guidelines, rule 67	

S.14 Always add a *default* case in a *switch* statement		
Description	Every *switch* block must contain a *default* case, even if the default case can happen only if there is an error. Add at least a *break* statement and a comment.	
Background	Robustness of the code.	
Class of rule	A	
Verification	PC-Lint no. 744	
Example	Bad	Good
	`switch (state) {` `  case 1:` `  state = 2;` `  ...` `  break;` `  case 2:` `  state = 1;` `  ...` `  break;` `}`	`switch (state) {` `  case 1:` `  state = 2;` `  ...` `  break;` `  case 2:` `  ...` `  break;` `  default:` `  state = 1;` `  ...` `  break;` `}`
	The good example forces the state machine back to a defined behavior in case of undetermined states.	
Reference	MISRA guidelines, rule 62	

S.15 Terminate each case in a *switch* statement with a *break*

Description	Each case (including the default case) shall be terminated with a *break* statement. If it is wanted to omit the *break* statement then it needs to be documented.	
Background	The complete code must be understood to judge whether the *break* statement was forgotten or not.	
Note	Even if the *break* in last case has no effect and is therefore in contradiction to rule S.9 it should be explicitly written.	
Class of rule	B+	
Verification	PC-Lint no. 616, 825	
Example 1	Bad	Good

```
switch (state) {
  case 0:
  case 1:
    do_something_for_case_0_and_1();
    break;
  case 2:
    do_something_for_case_2();
    break;
  default:
}
```

```
switch (state) {
  case 0:
  /*lint -fallthrough*/
  case 1:
    do_something_for_case_0_and_1();
    break;
  case 2:
    do_something_for_case_2();
    break;
  default:
    break;
}
```

The “good” example terminates each case with a break or documents that it was intentionally omitted.

Note: Lint understands the comment “/*lint -fallthrough*/” after the last statement of the preceding case. Please note that the case-block in X32 implies braces; so the LINT-comment must be outside the body of the case-block.

Example 2	Good
	`` void c_ac_next(void) { switch (acr) { case NC_ACR_BEGIN: if (lv_var_acr1) { acr = NC_ACR1; break; } /*lint -fallthrough*/ case NC_ACR1: if (lv_var_acr2) { acr = NC_ACR2; break; } /*lint -fallthrough*/ case NC_ACR2: if (lv_var_acr3) { acr = NC_ACR3; break; } /*lint -fallthrough*/ case NC_ACR3: /*lint -fallthrough*/ default: acr = NC_ACR_END; break; } } ```
	In this example it appears to be useful to break the rule: Each time we call the function we want to set the variable acr to the next implemented actuator (lv_var_acrX = 1). Therefore we switch to the current value of acr and remain inside the switch construct until we find a set actuator flag. Only then we set the variable and leave the switch construct. Sticking to the rule would mean a lot of overhead in each case in order to find out the next available actuator. Such exceptions have to be well documented!
Reference	MISRA guidelines, rule 61

S.16 Put the most decisive criterion of a condition at the beginning		
Description	Put the least probable criterion within a "&&" condition and the most probable criterion within a "||" condition at the beginning of the expression. Put the most probable case at the beginning of a *switch*-statement.	
Background	This will reduce calculation time.	
Note	The goal is to set the costs (runtime) of a criterion in relation to the benefit (early decision). While highly decisive criteria or criteria with low runtime should be placed at the beginning of the expression, unspecific criteria or time consuming calculations have to be placed at the end in order to optimize overall runtime.	
Class of rule	B	
Verification	Code review	
Example 1	Bad	Good
	`flag x;` `if (lv_igk && lv_st) x = 1;` `if (lv_st || lv_igk) x = 1;`	`flag x;` `if (lv_st && lv_igk) x = 1;` `if (lv_igk || lv_st) x = 1;`
	Assuming, that it is more likely that lv_igk is set and lv_st is not set, the order of the criterions is wrong in the "bad" example.	
Example 2	Bad	Good
	`flag f;` `if ((u8_add_u8_u8(a, b) < c) && f)…`	`flag f;` `if (f && (u8_add_u8_u8(a, b) < c)) …`
	In the "bad" example the function will be called in any case.	

S.17 Do not use a Boolean value for a *switch()*-expression		
Description	A switch expression should not represent a Boolean value.	
Background	If the expression in the switch statement is effectively representing Boolean data, then in effect it can only take two values, and an if-construct is a better way or representing the two-way choice.	
Class of rule	A	
Verification	Code review, partly by PC-Lint no. 961	
Example	Bad	Good
	`s16 tqi_sp;` `…` `switch (tqi_sp == 0) {` `  case 1: /* tqi_sp is zero */` `  …` `  break;` `  default: /* tqi_sp is non_zero */` `  …` `  break;` `}`	`s16 tqi_sp;` `…` `if (tqi_sp == 0) {` `  /* tqi_sp is zero */` `  …` `}` `else {` `  /* tqi_sp is non_zero */` `  …` `}`
	The "good" example is much more intuitive and readable than the "bad" one.	
Reference	MISRA guidelines, rule 63	

S.18 A switch statement shall have at least one case		
Description	A switch expression should have at least one non-default case.	
Background	A switch statement consisting of only a default branch or being completely empty does not influence the linear program flow, but it confuses the reader and could hide possible side effects of the evaluation of the switch expression.	
Class of rule	A	
Verification	PC-Lint no. 764	
Example	Bad	Good
	`switch (WriteMessage()) {` `  default:` `  lv_written = 1;` `  break;` `}`	`(void) WriteMessage();` `lv_written = 1;`
	The "good" example is much more intuitive and readable than the "bad" one.	
Reference	MISRA guidelines, rule 64	

2.6 Functions

F.1 Pass only correct parameters to functions		
Description	The number and types of the actual function arguments (data and pointers) must correspond to those of the formal parameters. Explicit type cast has to be used where necessary.	
Note	Functions called with no parameters should have empty parentheses.	
Class of rule	A+	
Verification	PC-Lint no. 118, 119, 418, 422, 515, 516, 579, 605, 668, 671, 747, 802, 807, 818, 917, 918	
Example	Bad	Good
	`extern flag foo1(u16 a, u16 b);` `void foo2(void){` `  if (foo1(1, 2, 3)) …` `}`	`extern flag foo1(u16 a, u16 b);` `void foo2(void){` `  if (foo1(1, 2)) …` `}`
	The “bad” example passes to foo1() a wrong number of parameters.	
Reference	MISRA guidelines, rule 77, 78, 85 Rule D.20, D.22, D.23	

F.2 Always declare function prototypes for exported functions

Description	The developer of an exported function has to declare a function prototype in the respective header file. When this function is to be used in another module, the header has to be included. No explicit extern-declaration shall be done outside this header. The prototype must be known when calling the function.	
Note	For private functions (functions that are only visible within the module) no explicit prototype is needed in order to avoid redundancy. Instead, the developer can take care for the correct order of the functions definitions to ensure that the prototype of a function is available when it is called.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 402, 718, 746, 937	
Example 1 (extern declaration)	Bad	Good
	`module_1.h` `----------` `module_1.c` `----------` `void foo1(u16 a){` `  …` `}` `module_2.c` `----------` `extern void foo1(int a);` `static void foo2(void){` `  u16 b;` `  foo1(b);` `}`	`module_1.h` `----------` `extern void foo1(u16 a);` `module_1.c` `----------` `void foo1(u16 a){` `  …` `}` `module_2.c` `----------` `#include <module_1.h>` `static void foo2(void){` `  u16 b;` `  foo1(b);` `}`
	In the “bad” example the prototype of the function foo1() is not declared in module_1.h. In order to use this function in module_2.c an explicit declaration of the prototype inside the code is required.	
Example 2 (prototypes for private functions)	a) Declaration needed	b) Declaration not needed
	`static u16 foo1(void);` `static void foo2(void);` `static void foo2(void){` `  u16 a = foo1();` `  …` `}` `static u16 foo1(void){` `  …` `}`	`static u16 foo1(void){` `  …` `}` `static void foo2(void){` `  u16 a = foo1();` `  …` `}`
	In the example a) function prototype declarations are required due to the wrong order of the function definitions. In the example b) the prototypes can be skipped.	
Reference	MISRA guidelines, rule 70, 71	

F.3 Define private functions as *static*		
Description	Functions without external linkage should be defined *static* in order to reduce their visibility.	
Background	Hiding private functions increases safety and avoids conflicts between different modules.	
Class of rule	A	
Verification	Code review, PC-Lint no. 765	
Example	Bad	Good
	MODULE A.C `void debug(s32 value) {` `  static s32 save;` `  save = value;` `}` MODULE B.C `void debug(void) {` `  static s32 marker;` `  marker = 1;` `}`	MODULE A.C `static void debug(s32 value) {` `  static s32 save;` `  save = value;` `}` MODULE B.C `static void debug(void) {` `  static s32 marker;` `  marker = 1;` `}`
	In the “bad” example the two functions “debug” will raise a conflict when linked. In the good example the functions will be local and will not influence each other.	
Reference	MISRA guidelines, rule 23 Rule D.10	

F.4 Every function shall have exactly one return point		
Description	Every function shall have exactly one return point, which is at the end of the function. Functions without return value (void) don’t have any return statement.	
Background	If it is necessary to have more than one return point, this has to be documented at the beginning of the function.	
Class of rule	B	
Verification	Code review, PC-Lint no. 533	
Example	Bad	Good
	`s32 test(s32 si1)` `{` `if (si1 == 1)` `  return 1;` `  else` `  return 2;` `}`	`s32 test(s32 si1)` `{` `  s32 tsi;` `  if (si1 == 1)` `  tsi = 1;` `  else` `  tsi = 2;` `  return tsi;` `}`
	If the “bad” example is as short as it is here, it is not really bad. But if the code becomes more complex you do not have any overview of the return points.	
Reference	MISRA guidelines, rule 82 Rule S.7, S.8	

F.5 Use complete function prototypes

Description	The prototype has to have an explicit return type (even void), types and identifiers for each parameter (void if no parameters).	
Background	The C language definition allows the omission for some declarations. This decreases readability and is not desired. Without declaring each function explicitly the compiler can not perform prototype checking.	
Class of rule	A	
Verification	Code review, PC-Lint no. 745, 747, 917, 918, 937, 939, 960	
Example	Bad	Good
	`extern a();` `f()` `{` `  return 1;` `}` `s32 g(b)` `{` `  return b + 1;` `}` `h()` `{` `  /* a function without result` `  value and parameter */` `  return;` `}`	`extern s32 a(void);` `u32 f(void)` `{` `  return 1;` `}` `s32 g(s32 b)` `{` `  return b + 1;` `}` `void h(void)` `{` `  /* a function without result` `  value and parameter */` `}`
	Is “a” really intended to return int or s32? What data type should f() return? Does f() take parameters? What parameter b is expected by g? Does h() return any value, is there a return value missing at the *return*-statement?	
Reference	MISRA guidelines, rule 73, 75, 76	

F.6 Use inline functions where required

Description	Inline functions are a useful means to implement function-like structures without the overhead of function calls. They shall be used wherever this is suitable.	
Background	The ISO standard of 1990, which is the basis of this document, does not define inline functions. They are introduced in the 1999 standard. Since their usage can be very advantageous, they are explicitly allowed by this rule.	
Note	Define only short functions as inline, because they are located every time they are called. Inline-functions are more robust than a function-like macro. An advantage is the defined evaluation of the actual parameters and the assignment of correct types to the formal parameters. Do not use pointers to reference inline functions like real functions. The behavior is not well defined then. Inline functions are resolved at compile time. Conflicts may appear when using inline functions inside precompiled objects.	
Class of rule	B	
Verification	Code review	
Example 1	Bad	Good
	`s32 av(s32 x, s32 y)` `{` `  return ((x + y) / 2);` `}`	`inline s32 av(s32 x, s32 y)` `{` `  return ((x + y) / 2);` `}`
	In the “bad” example using the function “av” may cause a lot of function call overhead as compared to its functionality.	
Example 2 (inlining of activation conditions)	Bad	Good
	`void c_fct(void) {` `  if (lv_cdn1 && lv_cdn2)` `  {` `  /* formula section */` `  }` `}`	`extern void c_fct_body(void);` `inline void c_fct(void) {` `  if (lv_cdn1 && lv_cdn2)` `  c_fct_body()` `}`
	If the application condition is mostly NOT fulfilled it might be an idea to inline the check and call the body only if needed. But due to the 1:1-relation between spec. and SW do not place the check directly in the scheduler.	
Reference	MISRA guidelines, rule 93	

F.7 Functions must not call themselves, neither directly nor indirectly. Recursion is not allowed

Description	Functions must not call themselves or other functions calling them.	
Background	Recursive calls of functions can easily lead uncontrollable stack usage and thus to stack overflow. These errors are hard to detect.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`u32 facu(u32 n)` `{` `  if (n > 1)` `  {` `  return (n * facu(n - 1));` `  }` `  else` `  {` `  return 1;` `  }` `}`	`u32 facu(u32 n)` `{` `  u32 i;` `  u32 r = 1;` `  for (i = 1; i <= n; i++)` `  {` `  r = r * i;` `  }` `  return (r);` `}`
	The bad example has higher stack consumption and is less efficient. (Remark: Overflow checking for calculation has been omitted for clarity.)	
Reference	MISRA guidelines, rule 70 Rule D.15, F.2	

F.8 Functions with variable numbers of arguments must not be used

Description	Functions must have a fixed number of arguments; ellipsis notation "..." in prototypes must not be used.	
Background	There are a lot of potential problems with this feature. Especially it might cause confusion, makes testing difficult and hampers static code checking.	
Class of rule	A	
Verification	PC-Lint no. 960, 1916	
Example	Bad	Good
	`u16 badfunction(u8 fac, ...)`	`u16 goodfunction(u8 fac, u8 fac1)` `u16 simplefunction(u8 fac)`
	Whenever a function with variable argument list might seem useful it must be replaced by a set of functions with fixed argument list.	
Reference	MISRA guidelines, rule 69	

F.9 Functions must always be declared at file scope

Description	Functions should not be declared inside another function.	
Background	Declaring functions at block scope may be confusing.	
Class of rule	A	
Verification	PC-Lint no. 960	
Example	Bad	Good
	`u32 facu(u32 n)` `{` `  void donothing(void);` `  ...` `  donothing();` `  ...` `}`	`void donothing(void);` `u32 facu(u32 n)` `{` `  ...` `  donothing();` `  ...` `}`
	The normal practice is to put function declarations of exported functions in header files only.	
Reference	MISRA guidelines, rule 68 Rule F.2	

F.10 The type of the return expression must match the type declared in the function prototype

Description	The type of the value returned by a function via the return statement must match the definition of the function. This implies that - each exit point of a non void function must have a return statement - return statements of functions of type void must not have an expression at all.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 82, 533	
Example	Bad	Good
	`u8 calc_something(void)` `{` `  u32 result;` `  ...` `  return result;` `}`	`u8 calc_something(void)` `{` `  u32 result;` `  ...` `  return (u8)result;` `}`
	Even if the bad example will work in this case it is good practice to return the correct type in any case.	
Reference	MISRA guidelines, rule 83, 84 Rule F.4	

F.11 Unused return values of a function shall be casted explicitly to void		
Description	If a function returns a value (according to its prototype), but in the current context this value is not used, the return type has to be explicitly casted to void.	
Background	If a function error information, then that information should be at least tested (often the return value contains some error status). If it is really intented not to use this information this should be explictly stated by casting the return value to void. This increases also static checkability.	
Class of rule	A	
Verification	PC-Lint no. 534	
Example	Bad	Good
	`Declaration:` `void* LIB_MEM_Copy (void* dst, void* src, u32 len);` `…` `LIB_MEM_Copy ( &a, &b, 128);…` `…`	`Declaration:` `void* LIB_MEM_Copy (void* dst, void* src, u32 len);` `…` `(void)LIB_MEM_Copy ( &a, &b, 128);` `…`
	Since in this use case the return value is not needed (LIB_MEM_Copy returns the destination pointer), the return value must be casted to void.	
Reference	MISRA guidelines, rule 86	

2.7 Preprocessor Directives

P.1 Check definition of compiler switches before use		
Description	Expressions for *#if* shall always be given in a way, that they can be resolved by the preprocessor. Before using an imported preprocessor definition check if it is defined. The definition checks shall be centralized locally. The check for the existence of the preprocessor definitions used in the file shall be done at the end of the import section. If used before (to steer including), check before usage, because it may be defined in a subsequent include-file.	
Background	Undefined preprocessor symbols in an #if-construct may produce unexpected results, but no warnings. LINT will find those undefined preprocessor symbols, but you must lint each file of the project every time to find these problems since the definitions are imported.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 553	
Example	Bad	Good
	`mod.c` `------` `#if NC_MOD_X == NC_MOD_X__A` `…` `#endif`	`mod.c` `------` `#ifndef NC_MOD_X` `  #error "NC_MOD_X not defined!"` `#endif` `#ifndef NC_MOD_X__A` `  #error "NC_MOD_X__A not defined!"` `#endif` `…` `#if NC_MOD_X == NC_MOD_X__A` `…` `#endif`
	If `NC_MOD_X` or `NC_MOD_X__A` is not defined the preprocessor will take a branch which is not intended. In the good example the compiler will abort.	
Reference	MISRA guidelines, rule 97	

P.2 Use the form *#include <file.h>* for header inclusion		
Description	Use *#include <header.h>* instead of *#include "header.h".*	
Background	The include path is defined by the build environment. The PDS (= Private Development Space) should be searched first, even if the compiled file is in the PIS (= Project Integration Space).	
Note	Use only standard lowercase characters for headerfile names	
Class of rule	A	
Verification	Code review, PC-Lint no. 12, 960	
Example	Bad	Good
	`File x.h at PDS` `#define XXX 1` `File x.h at PIS` `#define XXX 0` `File y.c at PDS` `#include "x.h"` `File z.c at PIS` `#include "x.h"`	`File x.h at PDS` `#define XXX 1` `File x.h at PIS` `#define XXX 0` `File y.c at PDS` `#include <x.h>` `File z.c at PIS` `#include <x.h>`
	In the bad example in y.c XXX is defined as 1, but in the file z.c XXX is defined as 0, because the current directory (PIS for z.c) is searched first before using the search-path (PDS, then PIS). In the good example the include-files are only searched along the include-path (PDS, then PIS); thus XXX is defined as 1 in both files.	
Reference	MISRA guidelines, rule 88, 89	

P.3 Macros containing expressions must be parenthesized		
Description	Parenthesize every parameter which is evaluated and the whole expression in a macro to avoid unwanted effects.	
Background	Macros are only a textual replacement. After that replacement the compiler applies the operator priority to the expression, not caring what was done by the macro-expansion.	
Class of rule	A+	
Verification	Code review, PC-Lint no. 665, 773	
Example 1	Bad	Good
	`#define D 0x3` `#define B(A) A*3` `#define C D+5` `#define M(a,b) a * b`	`#define D 0x3` `#define B(A) ((A)*3)` `#define C (D+5)` `#define M(a,b) ((a) * (b))`
	In the bad example ‘B(3+3)’ evaluates to 12 instead of 18; ‘2*C’ evaluates to 11 instead of 16; ‘M(1+2, 3+4)’ evaluates to 11 instead of 21.	
Example 2	Bad	Good
	`#define MACRO(a,b) a + b` `c = c * MACRO(x,y);` `is expanded to:` `c = c * x + y;`	`#define MACRO(a,b) ((a) + (b))` `c = c * MACRO(x,y);` `is expanded to:` `c = c * ((x) + (y));`
	In the "bad" example the macro's expression is not calculated before the surrounding multiplication.	
Reference	MISRA guidelines, rule 96	

P.4 Enclose macros which contain statements within *{}*		
Description	Enclose macros which contain statements within *{}*.	
Background	This is necessary, if the macro is called within a branch. The second statement would be unconditional.	
Note	A violation of this rule can cause problems only if also rule A.7 is violated. This rule exists only for backward compatibility with older code (where rule A.7 did not exist in the coding guidelines) and for increasing robustness.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`#define MY_MACRO()  lv_state_1 = 1;\` `                    lv_state_2 = 0` `if (lv_state_0) MY_MACRO();` `is expanded to:` `if (lv_state_0)  lv_state_1 = 1;` `lv_state_2 = 0;`	`#define MY_MACRO() {lv_state_1 = 1;\` `                     lv_state_2 = 0;}` `if (lv_state_0) MY_MACRO();` `is expanded to:` `if (lv_state_0) {lv_state_1 = 1;` `lv_state_2 = 0;};`
	In both examples rule A.7 is violated. In the bad example only the first statement (lv_state_1 = 1) is part of the if-structure, the second statement (lv_state_2 = 0) is executed unconditionally. Be aware that the "good" example is still not conforming to the coding guidelines. Nevertheless the problem does not occur anymore.	
Reference	Rule P.5, A.7	

P.5 Macros must not end with a “;”		
Description	Macros must not end with a semicolon.	
Background	If a macro ending with a ";" is used within code, a statement may be ended without the author's intention.	
Class of rule	A+	
Verification	Code review	
Example	Bad	Good
	`#define NC_FOO 10;` `...` `x = NC_FOO – 3;`	`#define NC_FOO 10` `...` `x = NC_FOO – 3;`
	In the bad example 10 will be assigned to x instead of 7. '- 3;' will be a valid statement without effect.	
Reference	Rule P.4	

P.6 Avoid unintended side effects in macros

Description

The standard way of implementing a macro should take into account the following rules:

- don't change global data
- use each argument exactly once
- expect variables or constant expressions as arguments,
  don't expect tokens or only constant expressions as arguments,
  don't expect, that you may pass complex expressions.

There might be good reasons for writing a macro, which does not follow these rules. In these cases, a detailed interface description of the macro is required (either in the design document or in the code). If macros are designed only to take constant expressions or even to take side-effect-expressions as a parameter, it has to be documented.

Background

Macros break the encapsulation functions provide. By textual reworking access to data not mentioned as parameters may be generated. The user is not able to judge what may happen with the parameters passed to the macro.

Class of rule

A+

Verification

Code review, PC-Lint no. 666

Examples

The examples below point out similar problems using macros. There is no division in "bad" and "good" examples as it depends on the circumstances what is "bad" and what is "good".

Example 1 (changing global data)

a) Wanted changing of global data:

```
#define A(a) {             \
    iga_av = ((a) + 0x80); \
}

A(iga_tmp)
```

b) More visible way:

```
#define A(a) ((a) + 0x80)


iga_av = A(iga_tmp);
```

In example 1a The global variable IGA_AV is changed in the macro. If possible use the "more visible way" (1b).

Example 2 (only constant expressions as argument)

Macro restricted to constants.

```
#define STATE(ch) \
((ch) < 10 ) ?    \
…                 \
: ((ch) < 20) ?   \
…                 \
: ((ch) < 30) ?   \
…                 \
:                 \
…
```

When passing a constant, the macro is expanded to one code line only; when passing a variable, each if branch will be expanded.

Example 3 (complex expressions as argument / use argument twice)

a) Standard way

```
#define STORE_MV(buf, mv) { \
    buf[1] = HI(mv);        \
    buf[2] = LO(mv);        \
}
```

b) Better if using complex expressions

```
#define STORE_MV(buf, mv) { \
    u16 tmp = (mv);         \
    buf[1] = HI(tmp);       \
    buf[2] = LO(tmp);       \
}
```

In the example 3a only variables can be passed as parameters. The example 3b can also handle complex expressions as parameters, only evaluating them once, but is using a local variable.

So if you use an argument twice in your macro, you have to document, whether you may only pass side-effect-free expressions (i.e. no ++ or no function call) or if the macro is designed to handle complex expressions.

Example 4 (tokens as argument)	Expecting a token	
	`#define NC_CH_A 1` `#define NC_CH_B 2` `#define CHANNEL(which) \` `  NC_CH_##which`	
	"CHANNEL(B)" will expand to NC_CH_B, then to 2. Passing a variable or a constant expression (e.g. 2+1) will not work.	

P.7 Do not use the *#pragma* directive

Description	Do not use the *#pragma* directive since it is compiler dependent.	
Background	There are other mechanisms installed to hide the *#pragma* directives in the compiler independent modules.	
Class of rule	A	The rule does not apply to modules, which are explicitly designed to be compiler dependent.
Verification	Code review	
Example	Bad	Good
	`#pragma section CODE ".A_CODE"` `  standard`	`#define MEM_CODE MEM_CODE_100MS` `#include <gmem.h>`
	The "bad" example uses a compiler specific way for memory allocation. In the "good" example the compiler specifics are hidden behind a header file. (The name and the behavior of the header file are generic, but its implementation is compiler specific). So the source file is not compiler dependent in this context.	
Reference	MISRA guidelines, rule 99 Rule G.2	

P.8 The ## preprocessor operator in macros must not be used if it is not required

Description	Macros support different ways of argument substitution. Using the double hash version is only suitable when concatenation of strings is required (e.g. for composing new identifier names).	
Background	Minimize problems with recursive substitution.	
Class of rule	A	
Verification	Code review, PC-Lint no. 960	
Example	Bad	Good
	`#define mul(a, b) ((##a) * (##b))`	`#define mul(a, b) ((a) * (b))`
	While the bad example works without problems in this case, the use of concatenation operators is useless and decreases readability.	
Reference	MISRA guidelines, rule 98	

P.9 Do not call a function-like macro without all of its arguments		
Description	Function like macros should always be called with the correct number (and intended type) of parameters. They have to be treated in this respect like real functions.	
Background	Function like macros are defined with a certain number of arguments (usually even a specific type is intended for each argument, though this is not checked). If you call the macro with some arguments missing, this will lead to undefined behavior. Comparing to real functions, the level of checks possible by the compiler are much more restricted. See F.1	
Class of rule	A+	
Verification	PC-Lint no. 123, 131	
Example	Bad	Good
	`#define mul(a, b) ((a) * (b))` `…` `x= mul(x1);` `…`	`#define mul(a, b) ((a) * (b))` `…` `x= mul(x1, x2);` `…`
	In the bad example the behavior is undefined.	
Reference	MISRA guidelines, rule 94 Rule F.1	

P.10 Use #include only before any C-statement		
Description	#include statements in a file shall only be preceded by other preprocessor directives or comments	
Background	All the #include statements in a particular code file should be grouped together near the head of the file. It is important to prevent the situation of executable code coming before a #include directive, otherwise there is a danger that the code may try to use items which would be defined in the header. To this end, the rule states that the only items which may precede an #include in a file are other pre-processor directives or comments.	
Class of rule	A	The rule does not apply to modules, which are using automatic configuration. This rule does not apply to includes due to the Memory allocation rules
Verification	Code review, PC-Lint no. 960	
Reference	MISRA guidelines, rule 87 Rule G.1	

P.11 Do not use *#define* within a block		
Description	Do not use *#define* within a block.	
Background	While it is legal C to place *#define* anywhere in a code file placing them inside blocks is misleading as it implies a scope restricted to that block which is not the case. In the SV templates *#define* directives are placed near the start of a file before the first definition.	
Class of rule	A	The rule does not apply to modules, which are using automatic configuration. This rule does not apply to defines due to the Memory allocation rules.
Verification	PC-Lint no. 960	
Example	Bad	Good
	`void fct(void) {` `  u8 foo;` `  #define NC_FOO 3` `  foo = NC_FOO;` `}`	`#define NC_FOO 3` `void fct(void) {` `  u8 foo;` `  foo = NC_FOO;` `}`
	Remember that preprocessor definitions are independent from the blocks. While NC_FOO is defined from the point of definition to the end of the file, foo is only defined till the end of the block. In the good example this is clearer.	
Reference	MISRA guidelines, rule 91	

P.12 #undef shall not be used		
Description	#undef shall only be used if it cannot be avoided and should be well documented.	
Background	#undef should not normally be needed. Its use can lead to confusion with respect to the existence or meaning of a macro when it is used in code.	
Class of rule	B	The rule does not apply to modules, which are using automatic configuration.
Verification	PC-Lint no. 961	
Example	Bad	Good
	`#define COMPARE(a,b) ((a) > (b))` `…` `/* use COMPARE(a,b) */` `…` `#undef COMPARE` `#define COMPARE(a,b) ((a) == (b))` `…` `/* use COMPARE(a,b) */` `…`	`#define COMP_GT(a,b) ((a) > (b))` `#define COMP_EQ(a,b) ((a) == (b))` `…` `/* use COMP_GT(a,b) */` `…` `/* use COMP_EQ(a,b) */` `…`
	In the bad example it is confusing that the definition of "COMPARE()" differs at different locations.	
Reference	MISRA guidelines, rule 92	

2.8 Form and Appearance

A.1 Use a proper representation of expressions		
Description	Binary operators except for "." (period) and "->" must be separated by a space from their operands, unary operators must not. Two operators should never be written one directly after the other, they must be separated by parenthesis. Redundant parentheses often are recommended to illustrate priority.	
Background	Best readability	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`x=a-b*c.re+d*c. im;` `*x=z--;` `y = -a + --b;` `u = &--v;` `*z = &x++ == &y;`	`x = a – (b * c.re) + (d * c.im);` `*x = z--;` `y = (-a) + (--b);` `u = (&(--v));` `*z = ((&(x++)) == (&y));`
	The “bad” example is difficult and unpleasant to read.	
Reference	Rule O.2	

A.2 Use common formatting rules, where applicable

Description	As a rule, at most one of the following programming units should appear in one line: – a statement – a declaration – "{" – "}" – a label – comment line Where readability is improved, two or more of the listed elements (except "}") may appear in one line. In any case, matching "{" and "}" must be placed as the first non-blank character in a line, and in the same column. Statements and declarations of the same block nesting level must begin in the same column. Nesting should be indicated by indentation with a uniform number of spaces (if possible no less than two) or indentations after each: – "{" – label (including case and default labels).	
Background	Best readability	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`{ a++; /* increment a */ if (b>a) b=c; else b=d;}`	`{` `  a++; /* increment a */` `  if ( b > a )` `  b = c;` `  else` `  b = d;` `}`
	The “bad” example is difficult and unpleasant to read.	

A.3 Limit identifier lengths to 31 characters		
Description	Use no identifier that is longer than 31 characters.	
Background	This is necessary to avoid a conflict with members of the tool chain, which can not handle more than 31 characters, e.g. ISO-C guarantees only 31 significant characters.	
Class of rule	A+	
Verification	PC-Lint no. 621	
Example	Bad	Good
	`u8 a_very_long_identifier_for_a_variable_1; u8 a_very_long_identifier_for_a_variable_2;`	`u8 a_very_long_id_for_a_var_1; u8 a_very_long_id_for_a_var_2;`
	Depending on the tool chain the variable "a_very_long_identifier_for_a_variable_1" and "a_very_long_identifier_for_a_variable_2" might not be distinguished (especially if external linkage is required)	
Reference	Kernighan / Ritchie topic "identifiers" ISO/IEC 9899 section 5.2.4.1 MISRA guidelines, rule 11	

A.4 Do not use identifiers with leading underscores		
Description	Never define variables / functions / preprocessor symbols / … with a leading underscore.	
Background	The compiler tool chain needs to define internal data for compile / link / debug purposes. These internal data are defined with a leading underscore. Normal code should never define variables / functions / preprocessor symbols / … with a leading underscore to avoid problems with the tool chain.	
Class of rule	A	
Verification	Code review	
Example	Bad	Good
	`u16 _foo;`	`u16 foo;`
	The "bad" example defines a variable with a leading underscore, which shall be avoided.	
Reference	Kernighan / Ritchie topic "identifiers" ISO/IEC 9899 section 5.2.4.1	

A.5 Keep the complexity of expressions small		
Description	If an expression is too complex, split it up in sub-expressions with temporary results.	
Background	Increases readability	
Note	Be especially careful to limit the number of pointer indirection levels. A large number of pointer indirections decreases readability very fast and leads to errors. This should be taken into account already in design phase!	
Class of rule	B	
Verification	Code review	
Example	Bad	Good
	`if (config_ptr->` `  device_config[device]->` `  channel_config[idx]->buffer->` `  content[0] != 0) {` `  …` `}`	`struct t_channel *my_channel;` `my_channel = config_ptr->` `  device_config[device]->` `  channel_config[idx];` `if (my_channel->buffer->content[0]` `  !=0 ) {` `  …` `}`
	In the "bad" example it is hard to check whether the correct channel is used. In the "good" example 'my_channel' can be used multiply.	
Reference	MISRA guidelines, rule 102	

A.6 Prefer decimal representation for integer constants. Do not use hexadecimal representation for negative constants. Do not use octal representation.		
Description	Any integer constants must be given in decimals, not in hexadecimals, except if the hexadecimal representation improves the readability and the value is positive (e.g. 0xffff or 0x8000). Octal constants (except zero) must not be used.	
Background	Excessive use of hexadecimal constants decreases the readability of the code and increases the effort for code reviews. Octal constants are normally not intended but used accidentally.	
Class of rule	A	
Verification	Code review, PC-Lint no. 960	
Example 1	Bad	Good
	`foo = 0xc7;` `bar = 012;`	`foo = 199;` `bar = 12;` `bla = 0xff;` `bla = 0;`
	The "bad" example assigns an octal constant 012 (decimal 10) to bar which is probably not what was intended.	
Example 2 (problems with negative hexadecimal constants)	Bad	
	`s32 x;` `x = (s32)-0x8000;` `if (x == (s32)-32768) {` `  /* int is of size 32bit */` `}` `else {` `  /* int is of size 16bit */` `}`	
	The evaluation of the "bad" example depends on the used integer promotion. On 16bit machines the difference between -0x8000 (which is unsigned) and -32768 (which is signed) occures.	
Reference	Kernighan / Ritchie topic "constants" ISO/IEC 9899 section 6.4.4.1 MISRA guidelines, rule 19	

A.7 The statements forming the body of an *if, else if, else, while, do while* or *for* statement must always be enclosed in braces		
Description	If the braces are omitted in case of single statements there is the danger of adding code which is intended to be part of the block but is actually not, because adding the braces has been forgotten. If X32 structures are used, this is automatically handled in a safe way.	
Background	Defensive programming.	
Class of rule	A	X32 generated syntax elements may not follow this rule
Verification	PC-Lint no. 960	
Example	Bad	Good
	`if (lv_do_it)` `  a = b;` `else` `  b = a;`	`if (lv_do_it)` `{` `  a = b;` `}` `else` `{` `  b = a;` `}`
	Both examples are equivalent, but when code is added to the else branch, adding the braces might be forgotten, resulting in incorrect code.	
Reference	MISRA guidelines, rule 59 Rule P.4	

2.9 Namespaces

N.1 Use an own name space for not specified items		
Description	For internal data separate namespaces shall be used, if these data are not encapsulated by standard-C-mechanism: Insert the name of the producing instance to any SW-internal data which is not encapsulated by standard-C-mechanism. An Instance may be an aggregate, an element-file, a C-function or any other scope, the data is belonging to. The naming rule for this instance may be instance-specific. Prefix symbols for meanings with an environmental description like: `<environmental description>__<meaning>`	
Background	By the distributed development of SW the risk of conflicting symbols grows. SW parts running correctly alone, crash when compiled together. Some keywords like NONE, WAIT, ON, OFF, OK, ERROR were used multiply for different purposes.	
Note	Those items may be: - SW internal, but (linker-) global data, but not “static” or function local data - macro definitions, but not explicitly specified NC-values	
Class of rule	A+	
Verification	Code review	
Example 1	Bad	Good
Linker name space	`mod_a.c` `-------` `flag lv_old_state; /*global data */` `mod_b.c` `-------` `flag lv_old_state; /*global data */`	`mod_a.c` `-------` `static flag lv_old_state;` `mod_b.c` `-------` `static flag lv_old_state;` ——————————— `OR:` `mod_a.c` `-------` `flag lv_mod_a_old_state;` `mod_b.c` `-------` `flag lv_mod_b_old_state;`
	To store $STATE_{n-1}$ the local data LV_OLD_STATE is used conflicting in two modules. To avoid conflicts the name space can be separated by standard-C-mechanism (static) or if not possible by creating own namespaces manually.	

Example 2

Meaning is specified

from AID / specification:

Switch	Main	Values	Meaning
NC_INJ_CONF	System flag to distinguish between MPI and HPDI system	0 1	MPI HPDI

Bad

```
config.h
--------
#define MPI    0
#define HPDI   1
```

Good

```
config.h
--------
#define NC_INJ_CONF__MPI     0
#define NC_INJ_CONF__HPDI    1
```

Although MPI and HPDI may be specified without any prefix as possible values for a compiler switch or an online data and are displayed in this form, the symbolic constant shall be prefixed to separate name spaces.

N.2 Do not define the same identifier in inner and outer scope

Description

Identifiers in an inner scope shall not use the same name as an identifier in an outer scope and therefore hide that.

Background

Hiding identifiers with an identifier of the same name in a nested scope leads to code which is very confusing.

Class of rule

A+

Verification

PC-Lint no. 578

Example

Bad

```
u16 n;

void fct(void) {
   int n;

   for (n = 0; n < 10; n++)
      …
}
```

Good

```
u16 n;

void fct(void) {
   int i;

   for (i = 0; i < 10; i++)
      …
}
```

In the "bad" example an already used identifier is hidden by a local one. In the "good" example an unused identifier is used.

Reference

MISRA guidelines, rule 21

Rule N.1

N.3 Do not use the same identifiers in different name spaces		
Description	No identifier in one name space shall have the same spelling as an identifier in another name space.	
Background	Although the compiler distinguishes between different name spaces the programmer may be confused.	
Note	Member names can be reused in different structures.	
Class of rule	B	
Verification	PC-Lint no. 578, 580	
Example	Bad	Good
	`typedef struct foo {` `  u8 foo;` `} foo;` `foo foo;`	`typedef struct s_foo {` `  u8 foo;` `} t_foo;` `t_foo foo_str;`
	In the "bad" example "foo" is used in four different name spaces. In the good examples prefixes indicate the different name spaces.	
Reference	MISRA guidelines, rule 12	

N.4 The declaration must exactly match the definition		
Description	The declaration must match the definition (including the name of the given identifiers).	
Class of rule	A+	
Verification	Code review, PC-Lint no. 15, 18, 516, 532, 937	
Example 1 (Function)	Bad	Good
	`extern s8 fct(u8 a, u8);` `u8 fct(u8 par1, u8 par2) {` `  …` `}`	`extern u8 fct(u8 par1, u8 par2);` `u8 fct(u8 par1, u8 par2) {` `  …` `}`
	In the "bad" example the return type and the naming of the parameters differ. In the good example even the spellings of the parameters are identical between declaration and definition, what is not required in the C standard, but by this rule.	
Example 2 (Data)	Bad	Good
	`typedef u16 my_type;` `extern u16 my_data;` `my_type my_data;`	`typedef u16 my_type;` `extern my_type my_data;` `my_type my_data;`
	In the "bad" example the written type of the variable differs between declaration and definition. Changes will not affect all occurrences and will easily lead to incompatibilities.	
Reference	MISRA guidelines, rule 26, 72, 74 Rule F.2, F.5	

N.5 Use declarations with external linkage only on file level		
Description	Declarations with external linkage are only allowed at file level and not in function or block level.	
Background	Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	
Class of rule	A+	
Verification	PC-Lint no. 512, 960	
Example	Bad	Good
	`static u8 data;` `void fct(void) {` `  extern u16 data;` `  data = 3;` `}`	`extern u16 data;` `static u8 my_data;` `void fct(void) {` `  data = 3;` `}`
	In the "bad" example the inner declaration of data hides the outer variable. In the "good" example the variables can be distinguished clearly.	
Reference	MISRA guidelines, rule 24, 68 Rule F.9, N.2	

N.6 Define an identifier with external linkage exactly once		
Description	An identifier with external linkage shall have exactly one external definition.	
Background	This precludes the situation where there is no definition of the identifier, and where there is more than one definition of the identifier (perhaps in different files).	
Class of rule	A+	
Verification	PC-Lint no. 14, Linker & Locater	
Example	Bad	Good
	`u16 data;` `u16 data = 3;`	`u16 data = 3;`
	In the "bad" example there are two definitions of data. Although this complies to ISO/IEC 9899 it is better to have one definition to avoid confusion. In the "good" example it is clear, that data is initialized.	
Reference	MISRA guidelines, rule 25 Rule D.10, N.1	

3. Further explanations

3.1 Views on the Rules

Rule	Page	Class of rule	function critical	Relevant also for non-portable code	Not checked by PC-Lint	Relevant for S4-Review	PC-Lint messages
G.0	9	A		X		X	950
G.1	9	A		X	X	X	
G.2	10	A		X	X	X	
G.3	10	A		X	X	X	
G.4	11	B		X	X	X	
G.5	12	A		X		X	506,527,528,529,550,563,681,750,751,752,753, 754,774,827,944
G.6	13	A	X	X		X	14,15,547,683
G.7	13	B	X	X	X	X	
C.1	14	B		X	X	X	
C.2	14	B		X	X	X	
C.3	15	B		X	X	X	
C.4	15	A	X	X			602
C.5	16	A		X	X	X	
C.6	16	A		X			950
C.7	17	A	X	X	X	X	
C.8	17	A		X	X	X	
D.1	18	A	X			X	970,971
D.2	19	A		X	X	X	
D.3	20	A	X	X	X	X	
D.4	21	A	X	X	X	X	
D.5	21	A	X	X	X	X	
D.6	22	A		X	X	X	
D.7	23	B		X	X	X	
D.8	24	A	X	X			530,603,644,645,771,772
D.9	25	B	X	X	X	X	
D.10	26	A		X		X	759,765,766
D.11	27	B		X	X	X	
D.12	27	A		X			961
D.13	28	A				X	950
D.14	29	A	X	X	X	X	
D.15	30	B		X	X	X	
D.16	31	A	X	X			421
D.17	31	B		X	X	X	
D.18	32	B		X			818,952,953,954
D.19	33	A	X	X		X	122,570,573,574,620,648,650,732,734,737,915, 960,961
D.20	34	A	X	X		X	524,569,570,573,574,653,712,713,731,732,734, 735,736,737,776,779,912,915,916,919,961
D.21	*35*						*-> Rule is removed*
D.22	36	A	X	X		X	519,549,609,610,611,740,923

Rule	Page	Class of rule	function critical	Relevant also for non-portable code	Not checked by PC-Lint	Relevant for S4-Review	PC-Lint messages
D.23	37	A	X	X			507,511,923
D.24	38	A		X		X	641
D.25	39	B		X	X	X	
D.26	39	A		X		X	46,806
D.27	40	A	X	X		X	531
D.28	41	A	X	X	X	X	
D.29	41	A	X	X		X	545,663
D.30	*42*						*-> Rule is removed*
D.31	42	A		X			940
D.32	42	A	X	X	X	X	
D.33	43	A		X		X	777,960
D.34	44	A		X			43
D.35	45	A	X	X	X	X	
O.1	46	B	X	X	X	X	
O.2	46	B		X		X	834,961
O.3	47	A	X	X		X	564,960
O.4	49	B		X	X	X	
O.5	50	A	X	X		X	413,613,674,733,789,794,809
O.6	51	A	X	X		X	572,701,702
O.7	*52*						*-> Rule is removed*
O.8	52	A			X	X	
O.9	52	A	X	X		X	701,702
O.10	53	A		X		X	720
O.11	53	A	X	X		X	648
O.12	54	A	X	X			421,829
O.13	55	A	X	X		X	414,795
O.14	56	A	X	X		X	946
O.15	*56*						*-> Rule is removed*
O.16	*56*						*-> Rule is removed*
O.17	57	A	X	X		X	413,613,794
O.18	58	A		X	X	X	
O.19	58	A		X		X	147,960(partlycovered)
O.20	59	A		X			720,820
O.21	59	A		X			720
O.22	60	A		X		X	720
S.1	61	A			X	X	
S.2	61	A			X	X	
S.3	61	A			X	X	
S.4	62	A			X	X	
S.5	62	A		X			801,961
S.6	63	A		X			421
S.7	63	B		X			960
S.8	64	B		X			960
S.9	64	A		X			505,506,522,527,681,827
S.10	65	B	X	X	X	X	
S.11	66	A	X	X		X	415,416,428,661,676,796,797
S.12	67	A		X	X	X	

Rule	Page	Class of rule	function critical	Relevant also for non-portable code	Not checked by PC-Lint	Relevant for S4-Review	PC-Lint messages
S.13	68	A		X	X	X	
S.14	68	A		X			744
S.15	69	B	X	X			616,825
S.16	71	B		X	X	X	
S.17	71	A		X		X	961
S.18	72	A		X			764
F.1	73	A	X	X			118,119,418,422,515,516,579,605,668,671,747, 802,807,818,917,918
F.2	74	A	X	X		X	402,718,746,937
F.3	75	A		X		X	765
F.4	75	B		X		X	533
F.5	76	A		X		X	745,747,917,918,937,939,960
F.6	77	B		X	X	X	
F.7	78	A	X	X	X	X	
F.8	78	A		X			960,1916
F.9	79	A		X			960
F.10	79	A	X	X		X	82,533
F.11	80	A		X			534
P.1	81	A	X	X		X	553
P.2	82	A		X		X	12,960
P.3	83	A	X	X		X	665,773
P.4	84	A	X	X	X	X	
P.5	84	A	X	X	X	X	
P.6	85	A	X	X		X	666
P.7	86	A			X	X	
P.8	86	A		X		X	960
P.9	87	A	X	X			123,131
P.10	87	A		X		X	960
P.11	88	A		X			960
P.12	88	B		X			961
A.1	89	B		X	X	X	
A.2	90	B		X	X	X	
A.3	91	A	X	X			621
A.4	91	A		X	X	X	
A.5	92	B		X	X	X	
A.6	93	A		X		X	960
A.7	94	A		X			960
N.1	95	A	X	X	X	X	
N.2	96	A	X	X			578
N.3	97	B		X			578,580
N.4	97	A	X	X		X	15,18,516,532,937
N.5	98	A	X	X			512,960
N.6	98	A	X	X			14,Linker&Locater

3.2 Index

3.3 Related Documents

Item	Description	Location
ISO/IEC 9899	Programming languages – C (international standard) first edition 1990-12-15	Intranet
MISRA guide-lines	Guidelines for the use of the C language in vehicle based software, April 1998	Intranet
Kernighan / Ritchie	The C Programming Language, Second Edition, ANSI C	Book
gdatatypes.doc, gtypes.h	Documentation/Definition of base data types	PVCS: 0_GEN PI_ARCH 0_GEN PI LIB_PPC for PPC target 0_GEN PI LIB_TC17XX for TriCore target
PC-Lint	PC-Lint reference manual, Gimpel Software 2001	Intranet
Templates	Powertrain Software Architecture Templates	Intranet
LPRC	Functions which are used to ease bit operations within the Basic SW	PVCS: 0_GEN PI LIB_PPC for PPC target, 0_GEN PI LIB_TC17XX for TriCore target
BITOP	Functions which are used to ease bit operations within the Applicative SW	PVCS: 0 GEN STD PPC for PPC target, 0 GEN STD TRI for Tricore target
ECMATH	Math saturated functions incl. filter and type conver-sions.	PVCS: 0 GEN STD PPC for PPC target, 0 GEN STD TRI for Tricore target
Naming conven-tion rules (M730006)	Naming Convention (for the Functional Data Names as well as for Aggregate Short Names, for all Aggregate Elements, Data Interfaces and Action Calls) This is not part of the M730006	Intranet
Memory alloca-tion rules (M730061)	Guideline for the Usage of the Memory Allocation Mechanism for Application Software	Intranet

4. Related upstream Documents

- P04903 “Development Guideline for Product Software”.

5. Team

5.1 Core team

Name	Department	Location
Khosrau Heidary	SV P ED T SW DA	Regensburg
Patricia Giron	SV P ED SW DP 1	Toulouse
Dr. Bärbel Steininger	SV P ED T SW CC2	Regensburg
Georges Esteves	SV P ED DS SW2	Toulouse
Andreas Franke	SV P ED GS SW1	Regensburg
Juergen Lindinger	SV P ED T SW IO2	Regensburg
Dr. Michael Niemetz	SV P ED DS SW1	Regensburg
Konrad Schmid	SV P ED T SW EE2	Regensburg
Gerhard Wirrer	SV P ED DT T SW61	Regensburg

5.2 Review team

Name	Department	Location
Deniz Culha-Booher	SV P ED DET	Auburn Hills
Razvan Jigorea	SV P ED T SW EE	Timisoara
JinSeong Kim	SV P ED SW (4)	Ichon
Joseph John Pattara	SISL SEC AT EE	Bangalore
Michael Warmuth	SV P ED DS SW1	Regensburg
Ulrich Schweickl	SV P ED Q SW	Regensburg
Dr. Martin Hurich	SV P ED T SW DA	Regensburg