Coding Guidelines

Internship

Revision: 0.1

Contents

[Contents 2](#_Toc10617477)

[1. Introduction 3](#_Toc10617478)

[1.1. Purpose 3](#_Toc10617479)

[1.2. Scope 3](#_Toc10617480)

[1.3. Terminology, Abbreviations and Definitions 3](#_Toc10617481)

[2. Method 5](#_Toc10617482)

[2.1. Numbering of the rules 5](#_Toc10617483)

[2.2. Structure of the rules 5](#_Toc10617484)

[2.3. Summary of the rules 6](#_Toc10617485)

[2.4. Verification of the coding rules 7](#_Toc10617486)

[3. Internship Coding Rules 8](#_Toc10617487)

[4. Naming Convention 46](#_Toc10617488)

[5. Type definition 62](#_Toc10617489)

[6. APPENDIX A: MISRA C Rule set 63](#_Toc10617490)

[7. APPENDIX B: Mapping of Functional Safety coding rules to MISRA C and Internal Coding Guidelines 76](#_Toc10617491)

[8. APPENDIX C: Known and allowed deviations from MISRA 2004 rules 76](#_Toc10617492)

[9. APPENDIX D: Technical clarifications 77](#_Toc10617493)

[9.1. NC.8 Naming Conventions for Defines 77](#_Toc10617494)

[10. Document History 81](#_Toc10617495)

# Introduction

## Purpose

The Coding Guideline and Naming Convention plans to explain the internal coding standards and the naming rules taken in consideration in the Internship.

Precondition for that document is, **that each developer is aware of and understands the MISRA rules. These rules will not be repeated here again.**

Applicable C language standard:

ISO/IEC 9899:1990 (E) (known as **C90**)

Applicable MISRA standards:

* MISRA 2004: mandatory for C code (implicitely defines the C language version)
* MISRA 2008: mandatory for C++ code
* MISRA 2012: for information only, not yet applicable

## Scope

The scope of this document is to familiarize the development teams with the C coding rules that are applied in the Internship. This can be achieved by providing positive/negative coding examples regarding a certain rule.

## Terminology, Abbreviations and Definitions

Process relevant topics are part of the Glossary of the Process Portal.

Specific abbreviations are explained in this chapter.

|  |  |
| --- | --- |
| MISRA | Motor Industry Software Reliability Industry |
| Declaration | This is only an information statement which does not create core and does allocate memory.  extern int MyVariable; |
| Definition | This is a statement that allocates memory for that object and “creates code”.  int MyVariable; |
| shall | This means the same like “has to” |
| should | This means “if ever possible and useful” but exceptions are allowed. |

# Method

## Numbering of the rules

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

## Structure of the rules

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **<ID>** | **<rule>** | | | |
| **Description** | | <detailed description of the rule> | | |
| **Note** | | <additional hints for usage> | | |
| **Example** | | Bad | Good | |
|  | | <example> |  | |
| <explanation of example> |  | |
| **Reference** | | <references> | | [Go to summary](#bookmark=id.39kk8xu) |

The rule itself is given in the header of the table, together with the identifier. How-ever, 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 to apply the rule, but which may help to understand the purpose of the rule, is given in a “Note” field.

All rules are mandatory for the Internship.

## Summary of the rules

|  |  |
| --- | --- |
| **Id** | **Topic** |
| **Internship Coding Guidelines** | |
| AIC.1 | All code shall conform MISRA 2004 |
| AIC.2 | All code shall conform to ISO 9899:1990 standard C |
| AIC.3 | Variables or parameters that do not change their values shall be declared as const. |
| AIC.5 | Code shall be explained by comments |
| AIC.5a | Comment shall be consistent with the design |
| AIC.5b | Decision points (if/else, loops, switch case) are to be commented |
| AIC.5c | Macros are to be commented |
| AIC.5d | Variables are to be commented |
| AIC.5e | Assignments with complex formulas/expressions (even if encapsulated by macros) are to be commented |
| AIC.5f | Comments shall be reviewed for compliance with the code it documents. |
| AIC.5g | Functions shall have Doxygen like comments. |
| AIC.6 | A header file must not contain code |
| AIC.7 | Bitfield usage is restricted. |
| AIC.8 | Dynamic memory allocation is forbidden. |
| AIC.9 | No global variables should be used. |
| AIC.10 | Assembler code is forbidden to be embedded in C files. |
| AIC.11 | Compiler and Linker Warnings are not allowed. |
| AIC.12 | Function-like macros are forbidden. |
| AIC.13 | Constraints for structures |
| AIC.14 | Standard C-libraries shall not be used |
| AIC.15 | Only symbolic usage of ENUM (no type cast, no use as index) |
| AIC.16 | Usage of external algorithms, Code, OpenSource |
| AIC.17 | Switch/Case rules |
| AIC.18 | Interface: no explicit or implicit anonymous parameters allowed |
| AIC.18a | The usage of the keyword “this” even in C language is forbidden |
| AIC.19 | Initialization of static variables |
| AIC.19a | Usage of local static variables shall be avoided |
| AIC.20 | Bitfields shall not be used. |
| AIC.21 | Bitfields and ENUM are forbidden as part of NVM content |
| AIC.23 | Public interface functions shall check incoming call parameter before use |
| AIC.26 | Support of AUTOSAR compliant Memory Mapping Support Common rule |
| AIC.32 | C99 style structure/union element ( .elementName) initialization is forbidden |
| AIC.33 | When signed variable types are used, the values shall not exceed the specified symmetric range. |
| AIC.34 | Typedef are mandatory on Interfaces |
| **Naming Convention** | |
| NC.1 | Naming convention for Software Work products |
| NC.2 | Naming Convention for Source code and Header files |
| NC.2b | Naming convention for (SW) Interface files |
| NC.3 | Data type Naming Conventions of Variables and Constants(requires “c”) |
| NC.4 | Type abbreviation for Arrays and Pointers |
| NC.5 | Static/Global Variable Naming Conventions in C |
| NC.6 | Local Variable Naming Conventions |
| NC.7 | Typedefs for structures and unions. |
| NC.8 | Naming Conventions for Defines/macros (without parameter) |
| NC.9 | Function Naming Conventions |
| NC.12 | Include protection |
| NC.13 | ENUM Typedef definition |
| **Type definition** | |

## Verification of the coding rules

The verification of the rules is done by using static code analysis tools, like PC-LINT or Polyspace and manually in case the rule can’t be checked by a tool.

# Internship Coding Rules

The Internship uses the ISO 9899:1990 "Programming languages- C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC9899/AMD1:1995, and ISO/IEC 9899/COR2:1996 for developing source code**.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.1** | **All code shall conform MISRA 2004** | | | |
| **Description** | | Allowed extensions to MISRA 2004 rules are listed in Appendix C | | |
| **Note** | | The document can be found in Process  It implicit also defines the correct C standard. | | |
| **Example** | | Bad | Good | |
|  | |  |  | |
|  | |
| **Reference** | |  | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.2** | **All code shall conform to ISO 9899:1990 standard C** | | | |
| **Description** | | **The only allowed extensions** to the ISO standard 9899:1990 are the use of inline  functions (as defined in ISO/IEC 9899:1999) and the C++ comment style(//) usage as well as special “embedded” keywords like: “interrupt”. Such special keywords are to be abstracted in a GLB\_ style macro in order to be able to modify them for portability reasons centrally. (\_\_interrupt, \_interrupt, interrupt ) | | |
| **Note** | | Modern Compiler specific keywords like  \_\_attribute\_\_((noinline)) etc. are to be abstracted by a GLB\_xxx definition. Direct use is not permitted.(portability/compiler abstraction) | | |
| **Example** | | Bad | Good | |
|  | | //direct use of inline keyword  **inline** bool DLPprj\_b\_IsFault (**const** uint8\_t cub\_SquibIndex)  {  (void)cub\_SquibIndex;  return b\_TRUE;  } | //use of global inline function  **GLB\_INLINE** bool DLPrj\_b\_IsFault(**const** uint8\_t cub\_SquibIndex)  {  (void)cub\_SquibIndex;  return b\_TRUE;  } | |
|  | |
| **Reference** | | MISRA-C:2004, rules 1.1, 19.14, 19.16  ISO/IEC 9899:1990  MISRA C 2004 Permits (First Edition).pdf  Permit/MISRA/C:2004/1.1.C.1 | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.3** | **Variables or parameters that do not change their values shall be declared as const.** | | | |
| **Description** | | a) Variables that do not change their value shall be declared as const and shall be initialized on the declaration line.(static/global will be put to ROM then)  b) Pointer to data that shall not be changed shall be declared as pointer to const  c) The pointer itself should also be declared as const.  d) Call by value Parameter do not need to be declared as const. | | |
| **Note** | | Main idea is to avoid common coding errors around increment of the value or the pointer itself.  Example:  What does the developer intend here to do?  \*p++;  \*(p++) or (\*p)++ ? | | |
| **Example** | | Bad | Good | |
|  | | a1)  uint8\_t AUTest\_aub\_Array [] =  { 1, 2, 3 };  a2)  uint8\_t AUT\_ub\_MyConst; | a1)  **const** uint8\_t AUTest\_**c**aub\_Array[] =  { 1, 2, 3 };  a2)  **const** uint8\_t AUTest\_**c**ub\_MyConst = 123; | |
| a1)  The elements of the array AUTest\_aub\_Array must not be changed.  Because it is not declared as const the array elements may be changed.  a2)  Variables declared as const have to be initialized on the declaration line. | a1)  The elements of the array AUTest\_**c**aub\_Array must not be changed.  Because it is declared as const, the elements of the array cannot be changed.  a2)  The variables declared as const is initialized on the declaration line. | |
| b1)  uint8\_t ABFile\_ub\_MyFunc(S\_SquibData \* ps\_SqData, | b1)  uint8\_t ABFile\_ub\_MyFunc( **const** S\_ABFile\_SquibData \* **const** cpcs\_SqData);  Cannot change the target data  Cannot change the pointer (no pointer arithmetic)  b2)  uint8\_t ABFile\_ub\_MyFunc( S\_ ABFile\_SquibData \* **const** cps\_SqData);  Cannot change the pointer (no pointer arithmetic)  b3)  uint8\_t ABFile\_ub\_MyFunc( **const** S\_ ABFile\_SquibData \* ps\_SqData);  same like:  uint8\_t ABFile\_ub\_MyFunc(S\_ ABFile\_SquibData **const** \* pcds\_SqData);  Cannot change the target data | |
| The function:  - shall not change the content at address ps\_SqData but it can change the data as well as the pointer itself. | The function:  - shall not change the content at address ps\_SqData  - does not change the pointer itself | |
|  | |  |  | |
| **Reference** | | MISRA 2004 16.7  PcLint Message: Note 953: Variable 'xxx’ (line 280) could be declared as const | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5** | **Code shall be explained by comments** | | | |
| **Description** | | Comments shall be meaningful and describe the functional aspect of the code. | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | EDRFile\_as\_DeployEventInfo[ub\_BufferIdx].ul\_DeployList |= ul\_Deployments; | /\*Update list of deployed squibs during event\*/  EDRFile\_as\_DeployEventInfo[ub\_BufferIdx].ul\_DeployList |= ul\_Deployments; | |
| The comment in the “good” example shows the intended function of the code line. | | |
| **Reference** | | - | | [Go to summary](#bookmark=id.2s8eyo1) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5a** | **Comment shall be consistent with the design** | | | |
| **Description** | | If a comment explains a variable, function or any other symbol or expression, it shall be consistent with similar explanation (if present) in the detailed design document. | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | For macrodefinition  CDScApp\_ub\_SERV22\_MAX\_NO\_OF\_IDS  Design description:  Purpose:  Service $22 specific define;max length of  DID list in request  Value: ((uint8\_t)8)  Source Code:  /\* Length define\*/ | For macrodefinition  CDScApp\_ub\_SERV22\_MAX\_NO\_OF\_IDS  Design description:  Purpose:  Service $22 specific define;max length of  DID list in request  Value: ((uint8\_t)8)  Source Code:  /\* max length of DID list in request \*/ | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5b** | **Decision points (if/else, loops, switch case) are to be commented** | | | |
| **Description** | | An explanation should indicate the reason for this condition. Repeating just the condition content is not a valid explanation. | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | if( A > Limit)  {  …  }  for(int i = 0; i != LIMIT; i++)  {…} | //checks if A is exceeding the voltage limit  if( A > Limit)  {  //trigger fault monitoring  …  }  //loop protects the array boundaries and initializes each array element  for(int i = 0; i != LIMIT; i++)  { x[i]= 0;…} | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5c** | **Macros are to be commented** | | | |
| **Description** | | Each macro (normal and also the “unwanted” function like macro) is to be commented. It shall be explained the meaning, purpose, and in case of complex combination of macro also what the purpose of this combination or formula is. Especially in case of formula (hidden by a macro) it is important to explain the formula. | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | **#define** FHCfg\_ub\_SELF\_HEAL\_VALUE (uint8\_t)40u  **#define** OIH\_ul\_DID4052\_SCALING ((uint32\_t)((4096.0 \* 10.0) / 3.0)) | /\* Number of ignition cycles for a fault to be self healed\*/  **#define** FHCfg\_ub\_SELF\_HEAL\_VALUE (uint8\_t)40u  **/\* scaling factor used to preserve accuracy of division \*/**  **#define** OIH\_ul\_DID4052\_SCALING ((uint32\_t)((4096.0 \* 10.0) / 3.0)) | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5d** | **Variables are to be commented** | | | |
| **Description** | | Every Variable shall be commented in regard of its purpose and meaning as well as intended use. | | |
| **Note** | |  | | |
| **Example** | | Bad | Good | |
|  | | **static** bool INDWLamp\_b\_FaultFlag;  E\_AOS\_BoSeConfigState e\_BoSeConfigState; // BoSe substate in case of being configured | //flag used for faults qualified while startup is active.  **static** bool INDWLamp\_b\_FaultFlag;  /\* BoSe substate, if configured; range of values: *BoSeConfig\_INIT* = 0,  *BoSeConfig\_CHILD* = 1,  *BoSeConfig\_ADULT* = 2,  *BoSeConfig\_FAULT* = 3  \*/  E\_AOS\_BoSeConfigState e\_BoSeConfigState; | |
|  | | |
| **Reference** | | - | | [Go to summary](#bookmark=id.2s8eyo1) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5e** | **Assignments with complex formulas/expressions (even if encapsulated by macros) are to be commented** | | | |
| **Description** | | Complex expressions may not be easy to understand by the reader. If furthermore some terms are “hidden” by a macro definition the task is even harder and may lead to wrong understanding of the code. | | |
| **Note** | |  | | |
| **Example** | | Bad | Good | |
|  | | /\* Static Buffer Aging Timer : 1 min \*/  **#define** EDRAsm\_ul\_STATIC\_BUFFER\_TIMER\_RELOAD ((uint32\_t) (1000 \* 60))  …  EDRAsm\_ul\_StaticBufferAgingTimer = EDRAsm\_ul\_STATIC\_BUFFER\_TIMER\_RELOAD; | /\* Static Buffer Aging Timer : 1 min \*/  **#define** EDRAsm\_ul\_STATIC\_BUFFER\_TIMER\_RELOAD ((uint32\_t) (1000 \* 60))  …  /\* reinit the 1min counter \*/  EDRAsm\_ul\_StaticBufferAgingTimer = EDRAsm\_ul\_STATIC\_BUFFER\_TIMER\_RELOAD; | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5f** | **Comments shall be reviewed for compliance with the code it documents.** | | | |
| **Description** | | Each time a change is made in the code, when reviewing the code, comments shall be also checked in order to find out if they still correctly and accurately document the code. | | |
| **Note** | |  | | |
| **Example** | | Bad | Good | |
|  | | If for whatever reason the sequence  **if**(FHApp\_b\_FaultProcessingInhibitted)  {  /\* Do not record Fault to NVM \*/  …  }  **else**  {  /\* Record fault to NVM \*/  … code which attempts a write to NvM  }  is changed to:  **if**(FHApp\_b\_FaultProcessingInhibitted == FALSE)  {  /\* Do not record Fault to NVM \*/  … code which attempts a write to NvM  }  **else**  {  /\* Record fault to NVM \*/  …  }  BAD: comments were not changed according to the change in code! | **if**(FHApp\_b\_FaultProcessingInhibitted == FALSE)  {  /\* Record fault to NVM \*/  … code which attempts a write to NvM  }  **else**  {  /\* Do not record Fault to NVM \*/  …  } | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.5g** | **Functions shall have Doxygen like comments.** | | | |
| **Description** | | Each function shall have doxygen like comments. | | |
| **Note** | |  | | |
| **Example** | | Bad | Good | |
|  | | **void** Function (**void**) | /\*\* @brief Function that does nothing.  \*  \* @param none  \* @return none  \*/  **void** Function (**void**) | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.6** | **A header file must not contain code** | | | |
| **Description** | | Header files are only used to export/import items. Therefore header files must  not contain executable code and must not define variables.  Exported definitions ( constants, variables) shall be declared with the keyword *extern*.  For function prototypes the keyword “extern” is optional.  The only code allowed in the header file is the definition of the inline functions that are used in several SWC or SWC internal files. This is sometimes required if modern compiler techniques like WholeProgram/CrossModule Optimization are not possible due to specific aspects (e.g. MPU concepts)  Then the provided macro (encapsulating “**static inline**”) is to be used to avoid “multiple instances of same name” messages and ensure C99 compatibility.  See: “MISRA C 2004 Permits (First Edition)” *“Permit/MISRA/C:2004/8.5.A.1 An inline function is defined in a header file”*  Two general exceptions are allowed:   1. inline functions 2. a clean and safe encapsulation mechanism is used.  example: Algo Class design header files | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | My\_Header.h  uint8\_t AUTFile\_ub\_MyVar;  uint8\_t AUTFile\_ub\_GetToto(void) {...} | My\_Header.h  extern uint8\_t AUFile\_ub\_MyVar;  extern uint8\_t AUFile\_ub\_GetToto(void);  My\_Cfile.c  uint8\_t AUFile\_ub\_MyVar;  uint8\_t AUFile\_ub\_GetToto(void)  {...} | |
| In the “bad” example the header file generates code. The “good” example exports the declared variable and function. | | |
| **Reference** | | MISRA-C:2004, rule 8.5  MISRA C 2004 Permits (First Edition).pdf  Permit/MISRA/C:2004/1.1.C.1  Permit/MISRA/C:2004/8.5.A.1 | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.7** | **Bitfield usage is restricted.** | | | |
| **Description** | | If bitfields are absolutely required to be used in it is forbidden to mix bitfield style access with bitwise operation on the same variable. | | |
| **Note** | | MISRA rule 3.5 gives some general hints for bitfield usage  AIC.20 states “Bitfield shall not be used”  Here this is just a clarification for exceptional cases where still bitfield needs to be used (e.g. µC Abstraction) | | |
| **Example** | | Bad | Good | |
|  | | typedef struct BitField  {  uint8\_t bf1: 1;  uint8\_t bf2: 1;  uint8\_t bf3: 1;  } BitField;  BitField var;  //forbidden bitwise direct access  var = 0x03 ; /\*read var.bf1\*/ | typedef struct TAG\_S\_EDRFile\_BITFIELD  {  uint8\_t bf1: 1;  uint8\_t bf2: 1;  uint8\_t bf3: 1;  } S\_EDRFile\_BITFIELD;    BitField var;    //allowed access by element name var.bf1 = 1; var.bf2 = 1; var.bf3 = 0; | |
| Because the order of the bitfields in the variable is implementation defined, there is no guarantee that var.bf1 is on the least significant 2 bits! | Variables declared as bitfields may be safely used as bitfields (access to members’ operator). | |
| **Reference** | | MISRA 2004 Rule 3.5, | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.8** | **Dynamic memory allocation is forbidden.** | | | |
| **Description** | | Do not use dynamic variables, e.g. do not use the heap functions malloc(). calloc(), free(),… | | |
| **Note** | | 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 could lead to an undefined behavior. | | |
| **Example** | | Bad | Good | |
|  | | {  uint8\_t \*pub\_ptr;  pub\_ptr = (uint8\_t \*) **malloc**(sizeof(uint8\_t));  …  } | {  uint8\_t ub\_var;  uint8\_t \*pub\_ptr;  pub\_ptr = &(ub\_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** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.9** | **No global variables should be used.** | | | |
| **Description** | | Passing of information between SWC must be realized using get/set functions (interface functions).  If the user deviates from this rule, it must be explained in a comment why.  Two general exceptions are allowed:   1. inline functions (implemented in header files) require sometimes variable access 2. Allowed if a clean and safe encapsulation mechanism is used.  example: Algo Class design header files <additional hints for usage> 3. A concept like in C++ (public, protected, private) for differentiation between SW-C internal vs. SW-C external communication support the possibility to stay encapsulation compliant without “full SW-global” variables. | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | **/\* in the CFG.h file\*/**  **extern** uint32\_t CFFile\_ul\_FL\_Configuration;  /\*--------------------------\*/  /\*in the CFG.c file \*/  ………  CFFile\_ul\_FL\_Configuration = 0xFE34;  **…………**  /\* in some other .c file \*/  #include “CFG.h”  …………  ABCFile\_ul\_cfg = CFFile\_ul\_FL\_Configuration**;**  …………. | **/\*in the CFG.h file \*/**  **extern** uint32\_t **CFFile\_ul\_GetFL** (void);  /\*--------------------------\*/  **/\* in the CFG.c file\*/**  **static** uint32\_t CFFILE\_ul\_FL\_Configuration;  uint32\_t **CFFILE\_ul\_GetFL**(void)  {  **return** CFFILE\_ul\_FL\_Configuration;  }  /\*--------------------------\*/  **/\* in some other .c file \*/**  #include “CFG.h”  FL\_ul\_cfg = **CFFILE\_ul\_GetFL();** | |
| The value of the global variable is CFFile\_ul\_FL\_Configuration is exported visible in the entire project | The static global variable CFFile\_ul\_FL\_Configuration is hidden inside the CFG file. The value is passed using interface functions | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.10** | **Assembler code is forbidden to be embedded in C files.** | | | |
| **Description** | | No assembler code is allowed in C code.  If assembler code must be used, it has to be encapsulated in .asm source files. | | |
| **Note** | | Background is, that modern compiler do automatic inline, reordering of instructions and other optimizations. This would affect inline assembler code too and could lead to unexpected side effects. Additionally this makes code “non portable”.  Known potential exceptions are interrupt locking routines with a single line asm instruction. Here the use must proof the “compiler/optimizer safe” implementation and document the need for this line in an appropriate comment.  Bad side effect: usually this requires often additional #pragma or other special keywords. | | |
| **Example** | | Bad | Good | |
|  | | If(ub\_Timer< MAX\_TIME)  (  asm("wait");  asm (" NOP "); //1  asm (" NOP "); //2  asm (" NOP "); //3  asm (" NOP "); //4  ) | Implementation in a separate Assembler file e.g. \*.s | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.11** | **Compiler and Linker Warnings are not allowed.** | | | |
| **Description** | | Compiler and Linker warnings caused by own code must be removed before the source file is checked in the Version Management tool. Maximum warning level must be enabled. | | |
| **Note** | | If there are warnings in own code caused by wrong code from external SWC (header files), the provider of that file is to be informed immediately. | | |
| **Example** | | Bad | Good | |
|  | | **OccupantStatus.c**  OIHStat\_v\_InitAllOcsData**();**  **During compilation the following message is received:**  05\_Source/50\_Occupant\_Information\_Handling/OIH\_OccupantStatus.c", line 235: warning (dcc:1500): function OIHStat\_v\_InitAllOcsData has no prototype | **OIH\_**Occupant**Status.c**  **static** void OIHStat\_v\_InitAllOcsData( void );  …………..  OIHStat\_v\_InitAllOcsData();  //no warnings anymore | |
| The message is due to the absence of the prototype of OCS\_v\_InitAllOcsData() before the function is called. | The prototype must precede the first call of a function in the source file. | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.12** | **Function-like macros are forbidden.** | | | |
| **Description** | | A function (inline) shall be used in preference of a function-like macro | | |
| **Note** | | Function like macros are not type safe; often lead to mistakes around braces. This can be avoided by using inline functions  GLB\_INLINE shall be used to abstract from the compiler specific version of the “inline” keyword.  All user specific INLINE definitions e.g. ISA\_INLINE must finally expand to GLB\_INLINE to avoid any trouble with the compiler or static code check tools like PCLint.  It is forbidden to directly use the pure compiler specific keywords “\_inline, inline, @inline...” inside the code to support portability. | | |
| Remark:  GLB\_INLINE has to expand to “static inline” in order to ensure that the inline “function” name does not get visible multiple times (identifyer clash/doublicate symbols....). In C90 inline is an extension (not well defined), inC99 it has to be mandatory static inline.  Be careful to avoid such constructs:  **static** GLB\_INLINE ABCFile\_b\_MyFunc( );  this expands to invalid code  **static static** inline ABCFile\_b\_MyFunc( ); | | |
| **Example** | | Bad | Good | |
|  | | #define ABCFile\_ub\_MAX(a,b) (if(a<b)  {  return b;  }  else  {  return a:  }) | GLB\_INLINE uint8\_t ABCFile\_ub\_MAX(uint8\_t a, uint8\_t b)  { if(a<b)  {  return b;  }  else  {  return a:  }  } | |
| No type check performed on parameters. | Type dependent definition (actual arguments are checked against formal arguments). | |
| **Reference** | | MISRA 2004: 19.7 A function should be used in preference to a function like macro  MISRA 2012: DIR 4.9 A function should be used in preference to a function-like macro where they are interchangeable  Supports CERT-C PRE31-C | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.13** | **Constraints for structures** | | | |
| **Description** | | For structures a type name has to be defined using typedef at the declaration point.  A structure must be declared with the same name in both locations  A structure construct shall :  Have data sorted according to size in order to avoid gaps;  Have dummy bytes inserted when sorting does not solve alignment ( this includes the case of members declared under compiler switches)  Have data sorted in descending order according to size in order to avoid gaps. This will help to safe RAM by reducing the structure size onto reasonable content  Structures of structures should be sorted according to their alignment too in descending order. (Example)  Structures located in NVM or their RAM representatives must always be filled on each gap with dummy/fill bytes  The structure “tag” in typedef struct TAG\_S {...}S\_TYPE;  should be avoided. If it is anyway used then it has to use the following syntax:  typedef struct **TAG\_S\_**TYPE {...}S\_TYPE | | |
| **Note** | | **MISRA 2004 5.4 (required) and 5.6 (Advisory) state:**  A tag name shall be a unique identifier.  ... No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names. Typedef name “S\_TYPE” should not be reused. | | |
| **Example** | | Bad | Good | |
|  | | **struct** {  uint16\_t uw\_CurrentState;  uint8\_t ub\_TransCounter;  uint32\_t ul\_LastState;  } SWFile\_s\_Var1;  **struct** {  uint16\_t uw\_CurrentState;  uint8\_t ub\_TransCounter;  uint32\_t ul\_LastState;  } SWFile\_s\_Var2;  -------------------------  **struct** S\_SW\_STCB {  uint16\_t uw\_CurrentState;  uint8\_t ub\_TransCounter;  uint32\_t ul\_LastState;  };  struct SWFile\_s\_var; | **typedef** **struct** {  uint32\_t ul\_LastState;  uint16\_t uw\_CurrentState;  uint8\_t ub\_TransCounter;  uint8\_t ub\_Fill;//Fillbyte  } S\_SW\_STCB;  S\_SWFile\_STCB SWFile\_STCB\_s\_Var;  -------------------------  **typedef struct** **TAG\_**S\_SWFile\_STCB{  uint32\_t ul\_LastState;  uint16\_t uw\_CurrentState;  uint8\_t ub\_TransCounter;  uint8\_t ub\_Fill;//Fillbyte  }S\_SWFile\_STCB;  S\_SWFile\_STCB SWFile\_STCB\_s\_Var; | |
| In the “bad” example the type-name is missing and the alignment of the structure elements is not assured. The data that retrieved from the structure can be corrupted because of the missing alignment. The “good” example defines a type-name and also assures that the date retrieved from the stricter is correct by long (32 bits) aligning the structure elements.  Declaring a variable using the “bad” example will use more storage space then the “good” example. | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.14** | **Standard C-libraries shall not be used** | | | |
| **Description** | | Standard C-libraries usually provide with the compiler shall not be used.  Example:  Function usage like e.g. memset(), memcpy() is forbidden. This should be covered by self implemented and tested “library” functions. | | |
| **Note** | | SW can use the services from C\_Util.  Background is that MISRA rule 3.6 advices to proof the MISRA compliance of the library implementation. In lot cases the compiler libraries are not available as source code and also the effort to proof the correctness would be too much effort.  Remark:  Libraries provided from 3rd party or the compiler are mostly provided as “QM” SW. This means they do not out of the box fulfill the Internship process requirements. Therefore they can not be used in our ASIL-B (or higher) compliant SW without additional discussion and further protecting mechanisms or a specific qualitifaction. | | |
| **Example** | | Bad | Good | |
|  | | /\* Initialize the RAM with data stored in ROM \*/  memcpy(pub\_target, pub\_source, ul\_CopyByteCount); | /\* Initialize the RAM with data stored in ROM \*/  CUtil\_v\_MemCpy(pub\_target, pub\_source, ul\_CopyByteCount); | |
| **Reference** | | MISRA-C:2004, rule 3.6, 20.9,20.10, 20.11, 20.12  ISO26262 | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.15** | **Only symbolic usage of ENUM (no type cast, no use as index)** | | | |
| **Description** | | ENUM types are to be used only for logical checks. Any usage as index to tables, or less than/greater than comparison is forbidden. An enum is only a symbolic value. The assigned value is implementation specific and any assumptions of its value are dangerous.  Any kind of calculations with ENUM are forbidden too. | | |
| **Note** | | ENUM are “implementation defined” in C.  The size depends on the µC and the Compiler  The type (signed/unsigned) depends on the Compiler  The size depends on the Compiler options/optimization e.g. ENUM compression features use smaller types than defined by C | | |
| **Example** | | Bad | Good | |
|  | | //assumptions on a value  **if** (e\_StartupType > *ABC\_e\_COLDSTART*)  {  ...  }  //-------------------  //forbidden usage as index  MyArray[e\_MyEnum] = OtherArray[e\_MyEnum];  //----------------  //calculations with ENUM are strictly forbidden e\_MyEnum++;  //type casts and assignments to different types are forbidden  uint32\_t MyInt = (uint32\_t)e\_MyEnum; | //only logical check  **if** (e\_StartupType == *ABC\_e\_COLDSTART*)  {  ...  } | |
| **Reference** | | MISRA-C:2004, rule 9.3 (only related, differences in content)  MISRA-C:2012, 10.1 An operand of *essentially enum type* should not be used in...:  MISRA-C 2012, rule 8.12, rule 10.4  Supporting: ISSUE 91285, **54700**  ISO/IEC 9899 : 1990 (E)  First edition 1990-12-15  6.5.2.2 Enumeration specifiers  ISO/IEC 9899:1999 (E)  6.7.2.2 Enumeration specifiers | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.16** | **Usage of external algorithms, Code, OpenSource** | | | |
| **Description** | | It is strictly forbidden to introduce any external code/code parts, that are not explicitly licensed for that project. | | |
| **Note** | | The intellectual property is usually protected by laws, licenses like GPL (open Source SW) or proprietary licenses. Also “algorithms” can be protected by patents. Even though sometimes code is published on code sharing/programming example web sites, it is not guaranteed that such code is legally free usable. Therefore it is forbidden to introduce code into Internship that was not explicitly licensed from e.g. 3rd party suppliers (Vector, ETAS etc.), provided by the OEM or explicitly legalized for the appropriate projects. | | |
| **Example** | | Bad | Good | |
|  | | n/a | n/a | |
| **Reference** | | Explicit customer requests to avoid it. | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.17** | **Switch/Case rules** | | | |
| **Description** | | Switch/case rules are well described in MISRA 2004 and 2012 but require some extensions.   1. A “default:” label definitely every time requires a “break;” 2. (Style) The body of every “case” label is to be encapsulated in { } independent whether this are single or multi line “case” | | |
| **Note** | | Rational:  Multiple tools struggle if there is no final “break” on a “default” or a default is completely missing.  PcLint settings do not detect all deviating variants currently.  See also  MISRA/PERMIT/C:2004/14.1.B.1 | | |
| **Example** | | Bad | Good | |
|  | | switch(ub\_value)  {  case 1:  //some code  break;  case 2: //no { }  //some code  break;  default: //missing break  } | switch(ub\_value)  {  case 1:  {  //some code  break;  }  case 2:  {  //some code  break;  }  default:  {  **break;**  }  } | |
| **Reference** | | BNF form of C language for switch/case | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.18** | **Interface: no explicit or implicit anonymous (void/void\*) parameters allowed** | | | |
| **Description** | | On Interfaces it is strictly forbidden to remove the type information.  This is supporting the compiler to recognize at compile time misuse of interfaces (wrong source data types) as well as function internal implementation errors of applying in the function wrong types to the parameter. Static code check tools (PcLint) and the compiler would not be able to check whether the access (size ranges) to the provided data is correct (according to design intention) because the size information is lost if this rule is not considered. Usage of asserts is impossible or useless.  It is forbidden to use “void” types as parameter.  It is forbidden to use them implicit e.g. as part of a structure that is a function parameter.  This rule is valid for SWC internal as well as SWC public interfaces.  Only **very rare exceptional cases** for a **few single C functions** e.g. accessing NVM data deviations are allowed.  Such exceptions are to be documented in Modeling tool as well as code and accepted by generic SW architect for generic changes, and project SW architect for project specific exceptions. | | |
| **Note** | | Exceptional cases where void<\*> parameter are allowed are explicitly to be judged by Generic SW architects case by case. e.g. for byte wise NVM operations.  Use the technique of opaque data types instead. Detailed explanation see MISRA 2012 DIR 4.8. **“void \* usage is not allowed for such cases. This is syntactical wrong** | | |
| **Example** | | Bad | Good | |
|  | | MyFunc(**void**\* vPointer)  {  MySpecificType\* MySpecificPointer = (MySpecificType\*)vPointer;  }  typedef struct  {  void\* vPointer;  }MyStruct;  MyFunc(MyStruct s\_Pointer)  {  MySpecificType\* MySpecificPointer = (MySpecificType\*) s\_Pointer. vPointer;  } | ABCFile\_v\_MyFunc(S**\_ABCFile\_MySpecificType**\* ps\_PointerToStructure)  {  ...  } | |
| **Reference** | | **MISRA 2004 Rule 8.2 (required)** “whenever an object or function is declared, its type shall be explicitely stated.”  **Rule 8.3,8.4, 8.5**  **Rule 11.1** Conversion shall not be performed between a pointer to function or any other type than integral type  **Rule 11.2** Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to **object type or a pointer to void.**  **Rule 11.4**  A cast should not be performed between a pointer to object type and a different pointer to object type.  **MISRA 2012 DIR 4.8**  lf a pointer to a structure or union is never dereferenced within a  translation unit, then the implementation of the object should be  hidden.  ...  Hiding the implementation details creates **an opaque type** which may be referenced via a pointer but  whose contents may not be accessed. | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.18a** | **The usage of the keyword “this” even in C language is forbidden** | | | |
| **Description** | |  | | |
| **Note** | | Rational:  In combined projects with C and C++ code this will cause conflicts as the keyword is predefined in C++ language.   * Polyspace CP gets blocked * Compiler can get disturbed (through header files exporting it to C++ code) | | |
| **Example** | | Bad | Good | |
|  | | void STM\_v\_Ctor (void \*this ,const T\_STATE pf\_InitialState); | void STM\_v\_Ctor (void \***ptr\_MySelf** ,const T\_STATE pf\_InitialState); | |
|  | |  | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.19** | **Initialization of static variables** | | | |
| **Description** | | It is forbidden to initialize a “SWC/file global” static or extern variable at declaration.  Every SWC shall have an own “Init( )” function where **for every** declared such (SWC global) static variable a clear assignment takes place. | | |
| **Note** | | Rational:  Any such variable that gets no assignment at declaration is assumed to be “0” initialized, independently whether this was the users intention or simply a mistake (forgotten to do a real assignment)  Static code analysis tools can not differentiate between “**intentionally nothing assigned**” vs. “**by mistake no value assigned**” therefore they assume “all memory is zero initialized”(like defined by C language). Such an assumption can be switched off and then the tool can assume such variables as “not initialized at all” and can discover such mistakes. | | |
| **Example** | | Bad | Good | |
|  | | My.c  Static uint32\_t MyStatic = (uint32\_t)0815; | My.c  Static uint32\_t ABCFile\_ul\_MyStatic;  My\_Init( void)  {  MyStatic = (uint32\_t)0815;  } | |
| **Reference** | | n/a | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.19a** | **Usage of local static variables shall be avoided** | | | |
| **Description** | | Declaration and definition of local static variables shall be avoided.  Only in exceptional cases this might be allowed if a rational is explained and commented in code. | | |
| **Note** | | MISRA advisory Rule (2004 8.7, 2012 8.9) tells, if an object needs to be visible on block scope, it should get declared and defined also only on block scope. This leads to the understanding that “file global” static variables shall be avoided too. This approach has multiple disadvantages.   1. Local static variables behave in regards of initialization differently than normal “auto (local) variables”. An auto variable gets each time again created and initialized when the function gets called. A “local static” gets initialized only once (in startup). This can confuse developers that are not aware of this behavior.   int MyLocal = 0; behaves different than static int MyLocal = 0;   1. Local static variables require special consideration in startup code of the SW as well as in linker file 2. AIC.19 requires own init routines for initialization of variables that have a life time that goes beyond the lifetime of a function call. 3. Maintenance of SW causes more effort if “local static” are used e.g. if other functions require access then code needs to be changed. This has significant impact on existing Unit tests even though the previous function requires no functional change. | | |
| **Example** | | Bad | Good | |
|  | | My.c  void ABCFile MyFunction(void)  {  static uint32\_t ul\_MyStatic = 0;  …  if(MyStatic == 0)  {…}  } | My.c  static uint32\_t ABCFile\_ul \_MyStatic = 0;  void ABCFile MyFunction(void)  {  …  if(MyStatic == 0)  {…}  } | |
| **Reference** | | MISRA 2004 Rule 8.7, MISRA 2012 Rule 8.9, AIC.19 | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.20** | **Bitfields shall not be used.** | | | |
| **Description** | | Bitfields shall not be used because they are not portable due to multiple reasons:   1. Little/big endianess depends on the CPU architecture 2. Size of bitfields is depending on the compiler and the used options e.g. Compilation optimization can compress bitfields to a smaller type than the C standard type int if the data still fits in 3. The order of the bitfields in the variable is implementation defined, there is no guarantee that var.bf1 is on the least significant 2 bits!   Allowed Exception:  For explicit access to elements of µC registers usually bitfields are defined as “Overlay” e.g. in AUTOSAR MCAL driver. | | |
| **Note** | | MISRA rule 3.5 gives some general hints for bitfield usage  See also Rule AIC.7 | | |
| **Example** | | Bad | Good | |
|  | | typedef struct BitField  {  uint8\_t bf1: 1;  uint8\_t bf2: 1;  uint8\_t bf3: 1;  } BitField;  BitField var;    //allowed access by element name var.bf1 = 1; var.bf2 = 1; var.bf3 = 0; | #define SWFile\_BIT1 (uint8\_t)0x1  #define SWFile\_BIT2 (uint8\_t)0x2  ...  uint8\_t ub\_var;    //allowed access by element name ub\_var = SWFile\_ub\_BIT1 ; ub\_var |= SWFile\_ub\_BIT2 ; ub\_var &= SWFile\_ub\_BIT3 ; | |
|  |  | |
| **Reference** | | MISRA 2004: Rule 3.5 (required) | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.21** | **Bitfields and ENUM are forbidden as part of NVM content.** | | | |
| **Description** | | Bitfields and ENUM are forbidden to be used in NVM data (EEPROM/DFLASH) because they are not portable due to multiple reasons and often changing in sizes:   1. Little/big endianess depends on the CPU architecture 2. Size is depending on the compiler and the used options e.g. Compilation optimization can compress bitfields to a smaller type than the C standard type int if the data still fits in 3. The order of the bitfields in the variable is implementation defined, there is no guarantee that var.bf1 is on the least significant 2 bits!   Allowed Exception:  For explicit access to elements of µC registers usually bitfields are defined as “Overlay” e.g. in MCAL driver.  Rational:  Register Access is µC HW specific and there is low need to be portable with such explicit expressions. | | |
| **Note** | | MISRA rule 3.5 gives some general hints for bitfield usage  Changed size due to “Description” will lead to different “filly byte” behavior and therefore time consuming bug hunting could be caused. | | |
| **Example** | | Bad | Good | |
|  | | typedef struct BitField  {  uint8\_t bf1: 1;  uint8\_t bf2: 1;  uint8\_t bf3: 1;  } BitField;  BitField var;    //allowed access by element name var.bf1 = 1; var.bf2 = 1; var.bf3 = 0; | #define SWC\_BIT1 (uint8\_t)0x1  #define SWC\_BIT2 (uint8\_t)0x2  ...  uint8\_t ub\_var;    //allowed access by element name ub\_var = SWC\_BIT1 ; ub\_var |= SWC\_BIT2 ; ub\_var &= SWC\_BIT3 ; | |
|  |  | |
| **Reference** | | MISRA 2004: Rule 3.5 (required) | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.23** | **Public interface functions shall check incoming call parameter before use** | | | |
| **Description** | | SW Component interfaces that are public available to other SW Components  have to ensure robustness by explicit parameter checks.  Any incoming Data from another SW-C or directly from HW  (e.g. SPI data, BUS signals)  that are directly or indirectly used as:   * Array index * Pointer to data * Pointer to function   are to be checked against:   * array bounds/overflow * not NULL pointer/illegal dereferencing | | |
| **Note** | | Defensive programming style is important on at least public interfaces as a balanced use between resource usage restrictions and robustness.  Such checks can be covered also by assert usage but assert are not active in series production SW anymore.  *Example here explicitely without naming convention in order to support readability*  The check against not NULL pointer helps to support static code analysis tools like Polyspace. If no such check is visible to them they will not analyze the code at all (RED error) | | |
| **Example** | | Bad | Good | |
|  | | void My\_v\_If(uint8\_t ub\_Idx)  {  //array write access unprotected  MyArray[ub\_idx] = ub\_Idx+5;  } | **//option A (not preferred)**  void SWC\_v\_If(uint8\_t ub\_Idx)  {  //array write access protected  if(ub\_Idx < C\_ub\_ArraySize(SWCFile MyArray))  {  SWCFile MyArray[ub\_idx] = ub\_Idx+5;  }  else  {  //invalid parameter  //required fault reaction??  }  };  **//option B (not preferred)**  **//use assert: //limitation ASSERT is not active in series SW**  **//therefore UNIT test behavior might change (test will fail) with “series production” setting**  void SWC\_v\_If(uint8\_t ub\_Idx)  {  //array write access protected !!  //at development time only !!  C\_v\_ASSERT(ub\_Idx < (DH\_SCNT)C\_ul\_ARRAY\_SIZE(MyArray));  MyArray[ub\_idx] = ub\_Idx+5;    };  **//option C**  **//signal by return success/no success**  bool SWC \_b\_If(uint8\_t ub\_Idx)  {  bool b\_RetVal = b\_FALSE;  //array write access protected  If(ub\_Idx < C\_ub\_ArraySize(MyArray))  {  SWCFile\_MyArray[ub\_idx] = ub\_Idx+5;  }  else  {  //invalid parameter  b\_RetVal = INVALID\_PARAMETER;  }  return b\_RetVal;  };  **//option D**  //mask away “bits” that can lead to out of bounds accesses  //are all 32Bits "agreed" for the project?  //easy and "cheap" protection:  //bits limited by a mask (fitting to the array size)  ul\_FireFLags = (uint32\_t)(MODAR\_ul\_GetFireFlagsFront( ) **&   SWC2\_MAX\_AXXEPTED\_BITS));** | |
|  |  | |
| **Reference** | | ISO 26262-6  Table 4 — Mechanisms for error detection at the software architectural level  1a Range checks of input and output data  1b Plausibility check Plausibility checks can include using a reference model of the desired behavior, assertion checks, or comparing signals from  different sources. | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AIC.26** | **Support of AUTOSAR compliant Memory Mapping Support**  **Common rule** | | | |
| **Description** | | <SWC> keyword shall be replaced by the Software Component short name defined in the SW Architecture. | | |
| **Note** | | Mandatory options are described within <> brackets .  Optional options are described within [] brackets | | |
| **Example** | | Bad | Good | |
|  | | n/a | SW Component Deployment Handler:  ***#define*** **DH**\_...  SW Component D-Loops Diagnosis:  ***#define*** **DLDIA**\_... | |
| **Reference** | |  | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **AIC.32** | **C99 style structure/union element ( .elementName) initialization is forbidden** | | |
| **Description** | C99 language standard allows “named” element initializations when assigning values to structure or union elements of e.g. constant structures.  This is forbidden | | |
| **Note** | Rational:  language version is C89/90 and not each compiler supports this C99 feature already as C90 extension (in all variants) so portability of SW is not ensured. | | |
| **Example** | Bad | Good | |
|  | *const DH\_MyType DHApp\_MyStruct*  *{*  .FirstElement = (uint8\_t)5,  .SecondElement = (uint8\_t)7  } | *//use C90 compliant style*  *const DH\_MyType DHApp\_MyStruct*  *{*  (uint8\_t)5,//.FirstElement  (uint8\_t)7 //.SecondElement  }  Remark:  Additional „,“ after the last assigned value (here 7)are syntax errors and therefore to be avoided | |
| **Reference** |  | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **AIC.33** | **When signed variable types are used, the values shall not exceed the specified symmetric range.** | | |
| **Description** | **The following symmetric ranges shall be used:**  //! Signed byte. Range: **-127..127**  typedef signed char sint8\_t;  //! Signed integer. Range: **-32767..32767**  typedef short sint16\_t;  //! Signed long integer. Range: **-2147483647..2147483647**  typedef long int sint32\_t;  //! Signed long long integer. Range:  **-9223372036854775807..9223372036854775807**  typedef long long int sint64\_t; | | |
| **Note** | For Internship the following macros have been defined specifying the symmetric ranges:  //! Maximum of sint8\_t  #define **MAX\_SINT8** (sint8\_t) 127  //! Minimum of sint8\_t  #define **MIN\_ SINT8** (sint8\_t) **(-127)**  //! Maximum of sint16\_t  #define **MAX\_SINT16** (sint16\_t) 32767  //! Minimum of sint16\_t  #define **MIN\_SINT16** (sint16\_t) **(-32767)**  //! Maximum of sint32\_t  #define M**AX\_SINT32** (sint32\_t) 2147483647  //! Minimum of sint32\_t  #define **MIN\_SINT32** (sint32\_t) **(-2147483647)**  //! Maximum of sint64\_t  #define **MAX\_SINT64** (sint32\_t) 9223372036854775807  //! Minimum of sint64\_t  #define **MIN\_ SINT64** (sint32\_t) **(-9223372036854775807)**  **Rationale:**   * This was done in order to have symmetric signals * This was done to avoid an overflow when calculating the absolute value from the sum of two minimum signed values and storing the absolute value in an unsigned value of the same type.   This will apply to every test method that is performed on the source code:   * Code Inspection * Unit Testing , etc. | | |
| **Example** | Bad | Good | |
|  | sint16\_t var1;  If var1 <= -32768  This will be considered an underflow. | sint16\_t var1;  If var1 <= MIN\_SWORD (-32767)  This will not be considered an underflow | |
|  | |
| **Reference** |  | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **AIC.34** | **Typedef are mandatory on Interfaces** | | |
| **Descript.** | For each exported (external visible) data/variables/enum etc. on Interfaces a typedef according to the described naming conventions is mandatory. | | |
| **Note** | **Rationale:**   * SPICE asessement requested to describe in SW architecture precisely the ranges of communicated data. It is better maintainable to describe the range at the type definition than on each instance/ or typle less declaration * Instantiation of such an element (e.g. as temporary variable because of return data from function calls) is not easy possible and this causes MISRA related conflicts * For explicit usage  of variables with defines, it can be ensured then by type cast’s in the define that the correct data and types are used and compared (supports a more type safe programming style and static code checking)   e.g.  **typedef** uint32\_t T\_MY\_TYPE  **#define** ABC\_t\_START\_VALUE ((T\_MY\_TYPE)0x0)  //..........................    T\_MY\_TYPE t\_Value = ABC\_t\_START\_VALUE; | | |
| **Example** | Bad | Good | |
|  | //MyHeader.h  //no datatype so can not get instanciated  enum MY\_WEEK  {  MONDAY,  TUESDAY  };  //extern accessable  extern Enum MYWEEK MyFunction(void);  //ExternalUser.c  void foo( )  {  //leads to MISRA violation   //possible “side effect…”  if( (MyFunction( ) == MONDAY) ||  (MyFunction( ) == TUESDAY) )  {  //do something  }  } | //MyHeader.h  typedef enum  {  MONDAY,  TUESDAY  } MY\_WEEK;  //extern accessable  extern Enum MYWEEK MyFunction(void);  //ExternalUser.c  void foo( )  {  const MY\_WEEK e\_Tmp = MyFunction( );  //MISRA compliant  if( (e\_Tmp == MONDAY) ||  (e\_Tmp == TUESDAY) )  {  //do something  }  } | |
| //MyHeader.h  Enum MY\_WEEK  {  MONDAY,  TUESDAY  };  //extern accessable  extern Enum MYWEEK MyFunction(void);  //ExternalUser.c  void foo( )  {  //leads to MISRA violation “side effect…”  if( (MyFunction( ) == MONDAY) ||  (MyFunction( ) == TUESDAY) )  {  //do something  }  } | |
| **Reference** |  | |  |

# Naming Convention

**Rational for naming conventions:**

**ISO 26262:6 Table 1 — Topics to be covered by modelling and coding guidelines**

1h Use of naming conventions ASIL-A++, ASIL-B ++, ASIL-C ++, ASIL-D ++

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| <SWC> | This is an abbreviation for the SWC where the function is defined. This is **all upper case** and 1..n characters in length. |
| <File> | This is an abbreviation of the file name for the SWC where the function is defined. This is m**ixed case Starting with a Capital letter** and additional 1..n characters in length. |
| <return type> | refers to the type definition of the return value (see table) and this is lower case of 1...4 characters in length  According to type definition chapter. |
| <storage type> | "e" if the variable is in NVM (EEPROM/dFlash)  or "c" if it is a static variable declared as “const”(but not for function arguments) |
| <data type> | refers to the prefixes used in the data type definition  e.g. "ub" for unsigned byte |
| <action> | is a verb describing the action that the function will perform.(e.g. Set/Get) This is mixed case and of any length, starting with a capital letter. |
| <TYPE-ID> | For ENUM it is important to mention also the TYPE-NAME in the ENUM element.  Here it is the TYPE name (the <DESCRIPTIVE\_NAME> part) |
| <Descriptive\_name> | File:  one or more concatenated words describing the main feature implemented by the file. This is **mixed case** and of any length  Unit/Function:  is a noun optionally followed by modifiers used to describe the action the function will perform. This is **mixed case** and of any length.  Macro:  All Capital Letter.  Typedef: (struct, ENUM)  All Capital Letter. |
| <ELEM\_DESCRIPTIVE\_NAME> | This is a descriptive name of the ENUM element. e.g. MONDAY |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.1** | **Naming convention for Software Work products** | | | |
| **Description** | | All work products created during the software life cycle shall follow the naming  defined in the configuration management plan.  Work products not mentioned in the CM plan should be self explaining.  Work products not owned by Internship do not have to follow the naming  conventions.  File extensions mostly are dependent from the choice of tools and shall be established at the beginning of a project. | | |
| **Note** | | SW work products are for example:   * detailed designs * source files (incl. the implemented Language elements like functions, variables, consts etc.) | | |
| **Example** | | Bad | Good | |
|  | | N/A | N/A | |
| - | - | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.2** | **Naming convention for Source code and Header files** (except SW Interface files) | | | |
| **Description** | | <SWC>\_< Descriptive\_name>.c/.h  File extension exception:  \*.inc is allowed for special algorithm files only because of simulation and cross compilation restrictions.  Developer specific file extensions like \*.con or \*.appl shall not be used anymore. | | |
| **Note** | |  | | |
| **Example** | | Bad | Good | |
|  | | DeployDecision.c  DeployDecision.h  Atic155HsLsTest.c | DH\_DeployDecision.c  DH\_DeployDecision.h  HATIC\_155HsLsTest or HAL\_ Atic155HsLsTest.c | |
| In the “bad” example the reference to the functionality is missing. | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.2b** | **Naming convention for (SW) Interface files** | | | |
| **Description** | | All **Interface files** shall be named according to the following format:  SWC = Software Component  IF = Interface abbrev.  CDT = Constants Defines Types  <SWC>\_<Descriptive\_name>\_**g**<IF, CDT > (**g**eneric)  <SWC>\_<Descriptive\_name>\_**p**<IF, CDT > (**p**roject specific)  **There exist the following classes:**  <SWC>\_Control\_gIF Init function and cyclic called functions  <SWC>\_Public\_gCDT Public provided Constants, Defines/Macro, Types and  <SWC>\_User\_gIF Public provided functions like GetStatus/SetStatus ...  <SWC>\_User\_**p**IF Public provided functions like GetStatus/SetStatus ...  Valid for the Software domain: Functionality = **SWComponent short name (2-5 characters)**  <Feature\_Descriptive\_name> - one or more concatenated words describing the main feature implemented by the file.  File extension \*.h | | |
| **Note** | | Using this rule is **optional for Domain Algo** | | |
| **Example** | | Bad | Good | |
|  | |  | DH\_Control\_gIF.h  DH\_Public\_gCDT.h  DH\_User\_gIF.h | |
|  | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.3** | **Data type Naming Conventions of Variables and Constants(requires “c”)** | | | |
| **Description** | | The Hungarian system for naming conventions of variables is mainly used. This  system let programmers identify the type of a variable from its name. It uses a set of lower case letter prefixes added [to normal variable names.](#_heading=h.upglbi) | | |
| **Note** | | - | | |
| **Example** | | Bad | Good | |
|  | | //type prefix is missing  AllowedFLs | //local variable **ul**\_AllowedFLs  //global/static variable  DHApp\_**ul**\_AllowedFLs  //global/static **constant** array  DHApp\_**c**aub\_StaggerTable[...] | |
| The “good” example shows the type of the variable. | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.4** | **Type abbreviation for Arrays and Pointers** | | | |
| **Description** | | "a" precedes the type of array elements.  "p" precedes the type that the pointer points to. | | |
| **Note** | | The type is describe in chapter [5.Type definition](#bookmark=id.1egqt2p) REMARK Avoid too deep nested names like “pointer to array of pointer to array of ….” | | |
| **Example** | | Bad | Good | |
|  | | - | //pointer to array with uint8\_t paub  //array ofpointers to  apub uint8\_t  uint8\_t aub\_Test[20];  //pointer to array of unsigned byte  uint8\_t \* paub\_Ptr; | |
| - |  | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.5** | **Static/Global Variable Naming Conventions in C** | | | |
| **Description** | | The static variables shall be named according to the following format:  <SWC><File>\_<storage type><data type>\_<Descriptive\_name> where: | | |
| **Note** | | *Static* ***global*** *variables:*  variables declared as static at the top level of a source file (outside any function definitions) are only visible throughout that file ("file scope", also known as "internal linkage")  *Static* ***local*** *variables: (not meant with this rule here)* variables declared as static inside a function are statically allocated .Hence whatever values the function puts into its static local variables during one call will still be present when the function is called again. | | |
| **Example** | | Bad | Good | |
|  | | RefValueCode | PM\_ub\_RefValueCode  PM: Variable is defined in SWC "PMODE"  ub: Data type: unsigned byte  RefValueCode: Description of the memory content | |
| S\_DHB\_DeploymentHandlerConfig DHB\_**e**s\_DHConfig | |
| The “bad” example does not provide the information about the owner and type of the variable | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.6** | **Local variable Naming Conventions** | | | |
| **Description** | | Local variables don’t contain the name of the SWC in which they are defined.  <data type>\_<Descriptive\_name> | | |
| **Note** | | A variable declared as *local* is one that is visible only within the C function that declares and defines it. It has local scope. | | |
| **Example** | | Bad | Good | |
|  | |  | ub\_Test  Data type: unsigned byte  Test - Description of the memory | |
| The examples shows the correct way of declaring local variables | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.7** | **Typedef for structures and unions.** | | | |
| **Description** | | According to rule AIC12, **structures should only be used based on a typedef**. The type name should be like this:  typedef struct **TAG\_S\_**<SWC><File>\_<DESCRIPTIVE\_NAME>  {  ...  } **S\_**<SWC><File>\_<DESCRIPTIVE\_NAME>;  The structure “tag” in typedef struct TAG\_S\_... {...}S\_TYPE;  is not mandatory.  To ensure that they are named in a consistent format, the following  standard has been established:  S\_<SWC><File>\_<DESCRIPTIVE\_NAME> U\_<SWC><File>\_<DESCRIPTIVE\_NAME> | | |
| M**embers** are following the naming conventions for local variables.  <data type>\_<Descriptive\_name> | | |
| **Note** | | **Former NC.7 and NC.11 combined**  **see also AIC.13**  The structure “tag” in typedef struct TAG\_S {...}S\_TYPE;  should be avoided. | | |
| **Example** | | Bad | Good | |
|  | | typedef struct s\_SomeStruct  {  uint8\_t ub\_SomeValue;  sint32\_t SomeOtherValue;  }s\_SomeStruct;  typedef union  {  uint8\_t SomeValue;  sint32\_t sl\_SomeOtherValue;  }u\_SomeUnion; | **typedef** struct **TAG**\_**S**\_DHApp\_SOMESTRUCT  {  uint8\_t ub\_SomeValue;  sint32\_t sl\_SomeOtherValue;  }S\_DHApp\_SOMESTRUCT;  **typedef** union  {  uint8\_t ub\_SomeValue;  sint32\_t sl\_SomeOtherValue;  }U\_DHApp\_SOMEUNION; | |
| In the “bad” example the type of the component is missing, camel case for typename are used.... The ”good” example provides the complete information. | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.8** | **Naming Conventions for Defines/macros (without parameters)** | | | |
| **Description** | | Some general versions of macro definitions are allowed.  For macros used in **preprocessor conditional** inclusion directives and/or in preprocessor expressions the following naming convention applies:  <SWC><File>\_<DESCRIPTIVE\_NAME> where  For macros used in source **code expressions** (as opposed to preprocessor expressions) the type indication is mandatory as described below:  <SWC><File>\_<**data type**>\_<DESCRIPTIVE\_NAME> where  Defines are not “consts” in the sense of “storage class specifier”.  So the “c” in the type descriptor is not allowed. | | |
| **Note** | | The following exceptions are allowed:  TRUE  FALSE  Please refer to the technical notes in Appendix D  Important: **No type cast** is required anymore.  For use as code expression b\_TRUE, b\_FALSE shall be used because they contain already the type cast. (e.g. “return b\_TRUE” ) | | |
| **Example** | | Bad | Good | |
|  | | //no SWC indication  #define USE\_STAGGERING 16  /\*type indication in a macro used only in preprocessor expressions\*/  #define DHT\_MMD\_TIMER 1u  #if (DHT\_MMD\_TIMER == 1u)  …  /\*typecast used in definition of a macro used in preprocessor expressions \*/  #define DHT\_MMD\_TIMER (uint8\_t)1  #if (DHT\_MMD\_TIMER == 1u) | //used as preprocessor switch #if(…) #define **DHApp\_**USE\_STAGGERING 16  //used as operator //provider guaranties  //that the value will stay in range of byte #define **DHApp\_**ub\_USE\_STAGGERING (uint8\_t)16 | |
| //used as preprocessor switch #if(…) #define DH\_USE\_STAGGERING 16**U**  //used as operator //provider guaranties  //that the value will stay in range of byte #define DH\_ub\_USE\_STAGGERING DH\_USE\_STAGGERING | |
| The “bad” example doesn’t specify the owner of the macro and the type. | | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.9** | **Function Naming Conventions** | | | |
| **Description** | | To ensure that functions are named in a consistent format, the following  standard has been established:  **PUBLIC:(exported on SWC-Interface)**  <SWC>\_<return value>\_<Action>< Descriptive\_name>  **SW-C internal: (non public “protected” or “private”)**  <SWC>< File>\_<return value>\_<Action><Descriptive\_name> | | |
| **Note** | | To keep public interface names stable, and to hide SW component internal implementation details, an approach is to define **public** interfaces just with a **Component abbreviation like DH** | | |
| **Example** | | Bad | Good | |
|  | | PM\_ReadMemory()  PM: Function is defined in SWC "PMODE"  Read: Action "Read"  Memory: Descriptive function name  ub\_GetID()  v: No return value (void)  Get: Action "Get"  ID: Descriptive function name | //public available  PM\_ub\_ReadMemory()  PM: Function is defined in SWC "PMODE"  ub: Returned value is unsigned byte  Read: Action "Read"  Memory: Descriptive function name  //SW-C internal  DHApp\_b\_IsTimerFinished()  Explains clearly by Name what it does and returns. | |
| For the first function the return type is missing and for the second function the SWC that owns the function is not mentioned | The name of the functions provides all the information necessary to form an idea about the intended usage of the function. | |
| **Reference** | | - | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.12** | **Include protection** | | | |
| **Description** | | According to MISRA each header file shall contain an include protection against “Multiple including”.  To ensure that functions are named in a consistent format, the following  standard has been established:  <File>**\_H**  **ALGO**  <File>**\_h**  Remark:  File contains here already the SW-C name.  The include protection should be **identical to the file name** but **in capital letter** | | |
| **Note** | | **Leading underscore(s) is/are forbidden, because this is reserved by C language already** | | |
| **Example** | | Bad | Good | |
|  | | **#ifndef** **\_\_**ISC\_CALLOUT\_GIF\_H\_\_  **#define** **\_\_**ISC\_CALLOUT\_GIF\_H\_\_  …  **#endif** | **#ifndef** HAL\_IF\_HAL\_COMBOSYNC\_gIF\_H  **#define** HAL\_IF\_HAL\_COMBOSYNC\_gIF\_H  …  **#endif** | |
| **Reference** | | MISRA 2004 20.1 | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NC.13** | **ENUM type definition** | | | |
| **Description** | | typedef enum **TAG\_E\_**<SWC><File>\_<DESCRIPTIVE\_NAME>  {  <SWC><File>\_e\_<TYPE-ID>\_<ELEMENT\_DESCRIPTIVE\_NAME>  ...  } **E\_**<SWC><File>\_<DESCRIPTIVE\_NAME>;//This is the “TYPE-ID”  The **“tag”** in typedef enum TAG\_E {...}E\_...;  **is not mandatory**. If it is used then it has to use the described syntax: | | |
| **Note** | | **The prefix on the ENUM values are mandatory**(other than with structs) because on assignment of such a value it would be easy to mix incompatible ENUM types with each other otherwise due to identical value names.  typedef enum  {  Monday, //prefix missing  ...  Sunday  } E\_ABC\_FullWeek;  E\_FullWeek MyWeek;  typedef enum  {  Monday, //prefix missing  ...  Friday  } E\_ABC\_WORKWEEK;  E\_ABC\_WORKWEEK NextWeek;  MyWeek = Monday; //Which Monday is taken, from which ENUM ???   1. from E\_ABC\_FULLWEEK 2. from E\_ABC\_WORKWEEK   **Better:**  typedef enum  {  ABC\_e\_**FULLWEEK\_**MONDAY,//show type name AND SW-C prefix  ...  ABC\_e\_**FULLWEEK \_**SUNDAY  } E\_ABC\_**FULLWEEK**;  MyWeek = ABC\_e\_**FULLWEEK\_**MONDAY; //no confusion anymore | | |
| **Example** | | Bad | Good | |
|  | | //prefixes missing  //Type not capital letter  //file name prefix missing  **typedef** **enum** E\_DH\_StartupType  {  *e\_COLDSTART* = 0,  *e\_WARMSTART* = 1  } E\_DH\_StartupType; | **typedef** **enum** TAG\_E\_DHApp\_STARTUP  {  ***DHApp\_****e\_*STARTUP\_*COLDSTART* = 0,  ***DHApp\_****e\_*STARTUP\_*WARMSTART* = 1  } E\_DHApp\_STARTUP;  //an instance of the type def  E\_DHApp\_STARTUP DHApp\_e\_Startup; | |
| **Reference** | |  | |  |

# [Type definition](#bookmark=id.3rdcrjn)

The following type definition applies to all work products of the Internship.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Definition** | **Prefix** | **Size(in bits)** |
| bool | Boolean | b | 8 |
| uint8\_t | unsigned char | ub | 8 |
| sint8\_t | signed char | sb | 8 |
| uint16\_t | unsigned short | uw | 16 |
| sint16\_t | signed short | sw | 16 |
| uint32\_t | unsigned long | ul | 32 |
| sint32\_t | signed long | sl | 32 |
| uint64\_t | unsigned long long | ull | 64 |
| uint64\_t | signed long long | sll | 64 |
| void | void | v | - |
| S\_<name> | Struct | s | any |
| U\_<name> | Union | u | any |
| E\_<name> | Enumeration | e | 8\*sizeof(int) |
| T\_<name> | Type definitions/re-definition | t | any |
| FCT\_<name> | pointer to a function | fct | CPU specific |
|  | pointer | p | CPU specific |

# APPENDIX A: MISRA C Rule set

R = Required

A = Advisory

M = Mandatory

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nr | Type | Rule Number | Rule Type | Rule description | Rule Verification |
|  |  | **Environment** |  |  |  |
| **MISRA 2012** | | **MISRA 2004** | |  |  |
| 1.1,  1.2 | R,  A | 1.1 | R | All code shall conform to ISO 9899:1990 "Programming languages- C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. | PCLINT #950 |
| 1.3 | R | 1.2 | R | No Reliance shall be placed on undefined or unspecified behavior. | Code Review |
| Dir 1.1 | R | 1.3 | R | Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform. | Code Review |
| Dir 1.1 | R | 1.4 | R | The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers. | PCLINT #621 |
| Dir 1.1 | R | 1.5 | A | Floating-point implementations should comply with a defined floating-point standard. | Not relevant; no floating point arithmetic in use |
|  |  | **Language extensions** |  |  |  |
| Dir 4.2, Dir 4.3 | A,  R | 2.1 | R | Assembly language shall be encapsulated and isolated. | Code Review |
| 1.2 | A | 2.2 | R | Source code shall only use /\* ... \*/ style comments. | PCLINT #950 |
| 3.1 | R | 2.3 | R | The character sequence /\* shall not be used within a comment. | PCLINT #602 |
| Dir 4.4 | A | 2.4 | A | Sections of code should not be "commented out". | Code Review |
|  |  | **Documentation** |  |  |  |
| Dir 1.1 | R | 3.1 | R | All usage of implementation-defined behavior shall be documented. | Code Review |
| Dir 1.1 | R | 3.2 | R | The character set and the corresponding encoding shall be documented. | Code Review |
| Dir 1.1 | R | 3.3 | A | The implementation of integer division in the chosen compiler should be determined, documented and taken into account. | Code Review |
| Dir 1.1 | R | 3.4 | R | All uses of the #pragma directive shall be documented and explained. | Code Review |
| Dir 1.1 | R | 3.5 | R | If it is being relied upon, the implementation-defined behavior and packing of bitfields shall be documented. | Code Review |
| -- | -- | 3.6 | R | All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation. | Code Review |
|  |  | **Character sets** |  |  |  |
| 4.1 | R | 4.1 | R | Only those escape sequences that are defined in the ISO C standard shall be used. | PCLINT #606 |
| 4.2 | R | 4.2 | R | Trigraphs shall not be used. | PCLINT #739 |
|  |  | **Identifiers** |  |  |  |
| 5.1, 5.2, 5.3, 5.4, 5.5 | R | 5.1 | R | Identifiers (internal and external) shall not rely on the significance of more than 31 characters. | PCLINT #621 |
| 5.3 | R | 5.2 | R | Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. | PCLINT #578 |
| 5.6 | R | 5.3 | R | A typedef name shall be a unique identifier. | PCLINT #623,#578 |
| 5.7 | R | 5.4 | R | A tag name shall be a unique identifier. | PCLINT #14,#15,  #64,#578 |
| 5.8, 5.9 | R,  A | 5.5 | A | No object or function identifier with static storage duration should be reused. | PCLINT #578,#580 |
| -- | -- | 5.6 | A | No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names. | PCLINT #578,#580 |
| -- | -- | 5.7 | A | No identifier name should be reused. | PCLINT #578,#580 |
|  |  | **Types** |  |  |  |
| 10.1,  10.2,  10.3,  10.4 | R,  R,  R,  R | 6.1 | R | The plain char type shall be used only for the storage and use of character values. | PCLINT #970,#971 |
| 10.1,  10.3,  10.4 | R,  R,  R | 6.2 | R | signed and unsigned char type shall be used only for the storage and use of numeric values. | Code Review |
| Dir 4.6 | A | 6.3 | A | Typedefs that indicate size and signedness should be used in place of the basic types. | PCLINT #970 |
| 6.1 | R | 6.4 | R | Bit fields shall only be defined to be of type unsigned int or signed int. | PCLINT #46 |
| 6.2 | R | 6.5 | R | Bit fields of type signed int shall be at least 2 bits long. | PCLINT #806 |
|  |  | **Constants** |  |  |  |
| 4.1,  7.1 | R,  R | 7.1 | R | Octal constants (other than zero) and octal escape sequences shall not be used. | PCLINT #960 |
|  |  | **Declarations and definitions** |  | |  |
| 8.2,  8.4,  17.3 | R,  R,  M | 8.1 | R | Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call. | PCLINT #718,#746,  #937,#957 |
| 8.1 | R | 8.2 | R | Whenever an object or function is declared or defined, its type shall be explicitly stated. | PCLINT #745,#939 |
| 8.3 | R | 8.3 | R | For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical. | PCLINT #18,#516,  #532 |
| 8.3 | R | 8.4 | R | If objects or functions are declared more than once their types shall be compatible. | PCLINT #15 |
| -- | -- | 8.5 | R | There shall be no definitions of objects or functions in a header file. | Code Review |
| -- | -- | 8.6 | R | Functions shall be declared at file scope. | PCLINT #960 |
| 8.9 | A | 8.7 | R | Objects shall be defined at block scope if they are only accessed from within a single function. | Code Review |
| 8.5 | R | 8.8 | R | An external object or function shall be declared in one and only one file. | PCLINT #578 |
| 8.6 | R | 8.9 | R | An identifier with external linkage shall have exactly one external definition. | Code Review |
| 8.7 | A | 8.10 | R | All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. | PCLINT #765 |
| 8.8 | R | 8.11 | R | The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage. | PCLINT #512 |
| 8.11 | A | 8.12 | R | When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization. | Code Review |
|  |  | **Initialization** |  |  |  |
| 9.1 | M | 9.1 | R | All automatic variables shall have been assigned a value before being used. | PCLINT #644,#771,  #530 |
| 9.2,  9.3 | R,  R | 9.2 | R | Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures. | PCLINT #940 |
| 8.12 | R | 9.3 | R | In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized. | PCLINT #960 |
|  |  | **Arithmetic type conversions** |  | |  |
| 10.3,  10.4,  10.6,  10.7 | R,  R,  R,  R, | 10.1 | R | The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant | PCLINT #524,#653 |
| 10.3,  10.4,  10.6,  10.7 | R,  R,  R,  R, | 10.2 | R | The value of an expression of floating type shall not be implicitly converted to a different type if: a) it is not a conversion to a wider floating type, or b) the expression is complex, or c) the expression is a function argument, or d) the expression is | PCLINT #747,#917,  #918 |
| 10.8 | R | 10.3 | R | The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression. | Code Review |
| 10.8 | R | 10.4 | R | The value of a complex expression of floating type may only be cast to a narrower floating type. | PCLINT #912 |
| -- | -- | 10.5 | R | If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand. | PCLINT #701,#702 |
| 7.2 | R | 10.6 | R | A "U" suffix shall be applied to all constants of unsigned type. | Code Review |
|  |  | **Pointer type conversions** |  | |  |
| 11.1 | R | 11.1 | R | Conversions shall not be performed between a pointer to a function and any type other than an integral type. | Code Review |
| 11.1.  11.2,  11.5,  11.7 | R,  R,  A,  R | 11.2 | R | Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void. | Code Review |
| 11.1,  11.2,  11.4,  11.6 | R,  R,  A,  R | 11.3 | A | A cast should not be performed between a pointer type and an integral type. | PCLINT #923 |
| 11.3 | R | 11.4 | A | A cast should not be performed between a pointer to object type and a different pointer to object type. | Code Review |
| 11.8 | R | 11.5 | R | A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer. | Code Review |
|  |  | **Expressions** |  |  |  |
| 12.1 | A | 12.1 | A | Limited dependence should be placed on C’s operator precedence rules in expressions. | PCLINT #834,#961 |
| 13.2 | R | 12.2 | R | The value of an expression shall be the same under any order of evaluation that the standard permits. | PCLINT #564 |
| 13.6 | M | 12.3 | R | The sizeof operator shall not be used on expressions that contain side effects. | PCLINT #960 |
| 13.5 | R | 12.4 | R | The right-hand operand of a logical && or || operator shall not contain side effects. | PCLINT #960 |
| 12.1 | A | 12.5 | R | The operands of a logical && or || shall be primary-expressions. | Code Review |
| 10.1 | R | 12.6 | A | The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, || and !). | Code Review |
| 10.1 | R | 12.7 | R | Bitwise operators shall not be applied to operands whose underlying type is signed. | PCLINT #960 |
| 12.2 | R | 12.8 | R | The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. | PCLINT #572 |
| 10.1 | R | 12.9 | R | The unary minus operator shall not be applied to an expression whose underlying type is unsigned. | PCLINT #501 |
| 12.3 | A | 12.10 | R | The comma operator shall not be used. | PCLINT #960 |
| 12.4 | A | 12.11 | A | Evaluation of constant unsigned integer expressions should not lead to wrap-around. | PCLINT #648 |
| Dir 1.1 | R | 12.12 | R | The underlying bit representations of floating-point values shall not be used. | Not relevant; no floating point arithmetic in use |
| 13.3 | A | 12.13 | A | The increment (++) and decrement (--) operators should not be mixed with other operators in an expression. | Code Review |
|  |  | **Control statement expressions** |  | |  |
| 13.4 | A | 13.1 | R | Assignment operators shall not be used in expressions that yield a Boolean value. | PCLINT #720,#820 |
| 14.4 | R | 13.2 | A | Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. | PCLINT #720 |
| Dir 1.1 | R | 13.3 | R | Floating-point expressions shall not be tested for equality or inequality. | PCLINT #777 |
| 14.1 | R | 13.4 | R | The controlling expression of a for statement shall not contain any objects of floating type. | PCLINT #960 |
| 14.2 | R | 13.5 | R | The three expressions of a for statement shall be concerned only with loop control. | Code Review |
| 14.2 | R | 13.6 | R | Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop. | Code Review |
| 14.3 | R | 13.7 | R | Boolean operations whose results are invariant shall not be permitted. | PCLINT #506 |
|  |  | **Control flow** |  |  |  |
| 2.1 | R | 14.1 | R | There shall be no unreachable code. | PCLINT #506,#527,  #681,#827 |
| 2.2 | R | 14.2 | R | All non-null statements shall either: a) have at least one side-effect however executed, or b) cause control flow to change. | PCLINT #505,#522 |
| 15.6 | R | 14.3 | R | Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character. | PCLINT #960 |
| 15.1,  15.2,  15.3 | A,  R,  R | 14.4 | R | The goto statement shall not be used. | PCLINT #801 |
| -- | -- | 14.5 | R | The continue statement shall not be used. | PCLINT #960 |
| 15.4 | A | 14.6 | R | For any iteration statement there shall be at most one break statement used for loop termination. | Code Review |
| 15.5 | A | 14.7 | R | A function shall have a single point of exit at the end of the function. | Code Review |
| 15.6 | R | 14.8 | R | The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement. | PCLINT #960 |
| 15.6 | R | 14.9 | R | An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement. | PCLINT #960 |
| 15.7 | R | 14.10 | R | All if ... else if constructs shall be terminated with an else clause. | PCLINT #960 |
|  |  | **Switch statements** |  |  |  |
| 16.1 | R | 15.1 | R | A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement. | Code Review |
| 16.2 | R | 15.2 | R | An unconditional break statement shall terminate every non-empty switch clause. | PCLINT #616,#825 |
| 16.3 | R | 15.3 | R | The final clause of a switch statement shall be the default clause. | PCLINT #744 |
| 16.4 | R | 15.4 | R | A switch expression shall not represent a value that is effectively Boolean. | PCLINT #960 |
| 16.6 | R | 15.5 | R | Every switch statement shall have at least one case clause. | PCLINT #764 |
|  |  | **Functions** |  |  |  |
| 17.1 | R | 16.1 | R | Functions shall not be defined with a variable number of arguments. | PCLINT #960 |
| 17.2 | R | 16.2 | R | Functions shall not call themselves, either directly or indirectly. | Code Review |
| 8.2 | R | 16.3 | R | Identifiers shall be given for all of the parameters in a function prototype declaration. | PCLINT #960 |
| 8.3 | R | 16.4 | R | The identifiers used in the declaration and definition of a function shall be identical. | Code Review |
| 8.2 | R | 16.5 | R | Functions with no parameters shall be declared with parameter type void. | PCLINT #937 |
| 8.2  17.3 | R,  M | 16.6 | R | The number of arguments passed to a function shall match the number of parameters. | PCLINT #118,#119 |
| 8.13 | A | 16.7 | A | A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object. | PCLINT #818 |
| 17.4 | M | 16.8 | R | All exit paths from a function with non-void return type shall have an explicit return statement with an expression. | PCLINT #533 |
| -- | -- | 16.9 | R | A function identifier shall only be used with either a proceeding &, or with a parenthesized parameter list, which may be empty. | Code Review |
| Dir 4.7 | R | 16.10 | R | If a function returns error information, then that error information shall be tested. | Code Review #121103 |
|  |  | **Pointers and arrays** |  |  |  |
| 18.1 | R | 17.1 | R | Pointer arithmetic shall only be applied to pointers that address an array or array element. | Code Review |
| 18.2 | R | 17.2 | R | Pointer subtraction shall only be applied to pointers that address elements of the same array. | Code Review |
| 18.3 | R | 17.3 | R | >, >=, <, <= shall not be applied to pointer types except where they point to the same array. | PCLINT #946 |
| 18.4 | A | 17.4 | R | Array indexing shall be the only allowed form of pointer arithmetic. | Code Review |
| 18.5 | A | 17.5 | A | The declaration of objects should contain no more than 2 levels of pointer indirection. | Code Review |
| 18.6 | R | 17.6 | R | The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. | PCLINT #733,#789 |
|  |  | **Structures and unions** |  |  |  |
| 1.3 | R | 18.1 | R | All structure and union types shall be complete at the end of a translation unit. | PCLINT #43 |
| 19.1 | M | 18.2 | R | An object shall not be assigned to an overlapping object. | Code Review |
| -- | -- | 18.3 | R | An area of memory shall not be reused for unrelated purposes. | Code Review |
| 19.2 | A | 18.4 | R | Unions shall not be used. | PCLINT #960 |
|  |  | **Preprocessing directives** |  | |  |
| 20.1 | A | 19.1 | A | #include statements in a file should only be preceded by other preprocessor directives or comments. | PCLINT #961 |
| 20.2 | R | 19.2 | A | Non-standard characters should not occur in header file names in #include directives. | PCLINT #961 |
| 20.3 | R | 19.3 | R | The #include directive shall be followed by either a <filename> or "filename" sequence. | PCLINT #12 |
| 20.4 | R | 19.4 | R | C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct. | Code Review |
| -- | -- | 19.5 | R | Macros shall not be #define’d or #undef’d within a block. | PCLINT #960 |
| 20.5 | A | 19.6 | R | #undef shall not be used. | PCLINT #961 |
| Dir 4.9 | A | 19.7 | A | A function should be used in preference to a function-like macro. | Code Review |
| 1.3 | R | 19.8 | R | A function-like macro shall not be invoked without all of its arguments. | PCLINT #131 |
| 20.6 | R | 19.9 | R | Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. | PCLINT #436 |
| 20.7 | R | 19.10 | R | In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##. | PCLINT #773 |
| 20.9 | R | 19.11 | R | All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator. | PCLINT #553 |
| 20.11 | R | 19.12 | R | There shall be at most one occurrence of the # or # preprocessor operators in a single macro definition. | PCLINT #960 |
| 20.10 | A | 19.13 | A | The # and # preprocessor operators should not be used. | PCLINT #961 |
| 1.3 | R | 19.14 | R | The defined preprocessor operator shall only be used in one of the two standard forms. | PCLINT #960 |
| 4.10 | R | 19.15 | R | Precautions shall be taken in order to prevent the contents of a header file being included twice. | Standard Coding Templates |
| 20.13 | R | 19.16 | R | Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor. | Code Review |
| 20.14 | R | 19.17 | R | All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related. | Code Review |
|  |  | **Standard libraries** |  |  |  |
| 21.1 | R | 20.1 | R | Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined. | PCLINT #683 |
| 21.2 | R | 20.2 | R | The names of standard library macros, objects and functions shall not be reused. | Code Review |
| 4.11 | R | 20.3 | R | The validity of values passed to library functions shall be checked. | Code Review |
| 21.3 | R | 20.4 | R | Dynamic heap memory allocation shall not be used. | PCLINT #586 |
| Dir 1.1 | R | 20.5 | R | The error indicator errno shall not be used. | PCLINT #586 |
| 1.3 | R | 20.6 | R | The macro offsetof, in library <stddef.h>, shall not be used. | PCLINT #586 |
| 21.4 | R | 20.7 | R | The setjmp macro and the longjmp function shall not be used. | PCLINT #586 |
| 21.5 | R | 20.8 | R | The signal handling facilities of <signal.h> shall not be used. | PCLINT #586 |
| 21.6 | R | 20.9 | R | The input/output library <stdio.h> shall not be used in production code. | PCLINT #829 |
| 21.7 | R | 20.10 | R | The library functions atof, atoi and atol from library <stdlib.h> shall not be used. | PCLINT #586 |
| 21.8 | R | 20.11 | R | The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used. | PCLINT #586 |
| 21.10 | R | 20.12 | R | The time handling functions of library <time.h> shall not be used. | PCLINT #586 |
|  |  | **Run-time failures** |  |  |  |
| Dir 4.1 | R | 21.1 | R | Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults. | (a)PCLINT/Code Review (b) SW Profiling on EMU (c) Exception/ASSERTS in SW |

# APPENDIX B: Mapping of Functional Safety coding rules to MISRA C and Internal Coding Guidelines

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FS Rule Nr  ISO26262  Part 6 | FS Rule | MISRA C  2004 Rule | MISRA C  2012 Rule | Rule |
| 8.1a | One entry and one exit point in subprograms and functions. | 14.7 | 15.5 |  |
| 8.1b | No dynamic objects or variables, or else online test during their creation | 20.4 | 21.3 | AIC8 |
| 8.1c | Initialization of variables | 9.1 | 9.1 |  |
| 8.1d | No multiple use of variable names | 5.2, 5.7 | 5.3 |  |
| 8.1e | Avoid global variables or else justify their usage | 8.7 | 8.9 | AIC9 |
| 8.1f | Limited use of pointers | 11.1, 11.2, 11.3, 11.4, 11.5 | 11.1 – 11.8 |  |
| 8.1g | No implicit type conversions | 10.1, 10.2 | 10.3, 10.4, 10.6, 10.7 |  |
| 8.1h | No hidden data flow or control flow | 8.7,13.5, 13.6 | 8.9, 14.2 |  |
| 8.1i | No unconditional jumps | 14.4 | 15.1, 15.2, 15.3 |  |
| 8.1j | No recursions | 16.2 | 17.2 |  |

# APPENDIX C: Known and allowed deviations from MISRA 2004 rules

The following exceptions from the MISRA C 2004 Rule Set are allowed:

Usage of inline keyword (MISRA Rule 8.5)

Usage of interrupt keyword

Usage of C++ comments

# APPENDIX D: Technical clarifications

## NC.8 Naming Conventions for Defines

This is a difficult topic that requires some more detailed explanations. Main problem is, that excessive typecasting can potentially block static code check tools from discovering code issues.

Below two test cases try to show whether type casting, suffixing etc. blocks or supports tools like PCLint in discovering code weaknesses.

The value 65920 is used here to show if a tool would discover macro value changes in external modules (exceeding an expected type range) that affect the user’s code in other source files. It intentionally exceeds the size of a uint8\_t so that the static code check tool should complain here.

Defines, i.e. macro definitions without parameters, are basically used:

as names for constants used as operands, therefore an indication of the type using the type-prefix part of the name is important/useful

as names for constants used in preprocessor constant expressions  
the evaluation of these constant expressions are done according to the usual rules except that type int/unsigned int act as if implemented as type long/unsigned long, respectively (because of that the actual types of the operands have little or no importance at all!)

as switches used in conditional inclusion preprocessor directives (#ifdef, #if defined(name), and such) where the value range is not used but the mere fact that the macro is defined or not

Test case pre-conditions:

a) the "complete" definition, including the type indication

b) no type indication

c) no type indication

If a macro shall be defined to be used both for case c) usage and for case a) or b) then the following procedure should be followed:

/\* define a compiler switch

no type indication, just be sure to indicate signedness by using the U suffix if that's the case \*/

#define MOD\_CASE\_C\_CONSTANT 1u

/\* define a macro to be used as an operand by using the compiler switch already defined and specify the type according to the value of the substitution string in the definition of the compiler switch \*/

#define MOD\_ub\_CASE\_A\_CONSTANT MOD\_CASE\_C\_CONSTANT

1) No typecast in the definition,

If no typecast is used in the definition, then each time the macro is involved in an expression with operands of a different type, the appropriate typecast shall be used!

MOD\_ul\_SomeLong + (uint32\_t)MOD\_ub\_CASE\_A\_CONSTANT

If later on the value of the compiler switch is modified to some value outside the range of the type indicated in the case a) constants name, then:

the result of expressions (other than assignment) involving the constant will be correct because the type of an integer constant (the one specified in the substitution string of the macro) is the first one which can represent that value and this type is at least **int!** That means no truncation takes place as it would have happened if in the definition a typecast to a smaller type would have been specified!

- the assignment of the constant to a variable of the same type as the one indicated by the <data type> part of its name will trigger a lint message  
 *(Warning 569) of the type: Loss of information in initialization(m bits to n bits)*

the type indication in the macro name will no longer be consistent with its value (this has no harmful effect on the result of evaluation!)

//preprocessor switch  
//this big value illustrates the “out of range” issue   
//when using it for uint8\_t assignment on user’s code location

#define AOI\_DEFINE 65920u

//arithmetic operands:

//The provider "guaranties" that this value will stay in range of uint8\_t

#define AOI\_ub\_DEFINE AOI\_DEFINE

uint8\_t AOI\_v\_F(void);

#ifdef AOI\_DEFINE

#if (AOI\_DEFINE == 65920)

uint8\_t AOI\_v\_F(void)

{

uint8\_t AOI\_ub\_var3 = AOI\_DEFINE;

uint8\_t AOI\_ub\_var1 = AOI\_ub\_DEFINE;

uint32\_t AOI\_ul\_var2 = AOI\_ub\_DEFINE;

//...

AOI\_ub\_var3 = (uint8\_t)( (uint32\_t)AOI\_ub\_var1 + AOI\_ul\_var2);

//...

return AOI\_ub\_var3;

}

#endif

#endif

2) typecast used in the definition:

**This example shows that typecasts can block static code check tools**

#define MOD\_ub\_CASE\_A\_CONSTANT (uint8\_t)MOD\_CASE\_C\_CONSTANT

If a typecast used in the definition then each time the macro is involved in an expression with operands of a different type, the appropriate typecast shall still be used

MOD\_ul\_SomeLong + (uint32\_t)MOD\_ub\_CASE\_A\_CONSTANT

If later on the value of the compiler switch is modified to some value outside the range of the type indicated int e case a constants name then:

the result of expressions (other than assignment) involving the constant will be NOT correct because of the type cast in the definition

the assignment of the constant to a variable of the same type will NOT trigger a lint message

the type indication in the macro name will no longer be consistent with its value (this has no harmful effect on the result of evaluation!)

//preprocessor switch  
//this big value illustrates the “out of range” issue   
//when using it for uint8\_t assignment on user’s code location

#define AOI\_DEFINE 65920u

//arithmetic operands

#define AOI\_ub\_DEFINE (uint8\_t)AOI\_DEFINE

uint8\_t AOI\_v\_F(void);

#ifdef AOI\_DEFINE

#if AOI\_DEFINE == 65920

uint8\_t AOI\_v\_F(void)

{

uint8\_t AOI\_ub\_var3 = AOI\_DEFINE;

uint8\_t AOI\_ub\_var1 = AOI\_ub\_DEFINE;

uint32\_t AOI\_ul\_var2 = AOI\_ub\_DEFINE;

//...

AOI\_ub\_var3 = (uint8\_t)( (uint32\_t)AOI\_ub\_var1 + AOI\_ul\_var2);

//...

return AOI\_ub\_var3;

}

#endif

#endif

Conclusion:

Solution 1) (no typecast in the definition) shall be used!

# Document History

|  |  |  |
| --- | --- | --- |
| **Rev. #** | **Change Description** | **Author** |
| 0.1 | Initial Revision | C. Voina |
| 0.2 | Added comments for functions | C. Voina |