Coding G Style_Blk_Ln_002 uidelines
For
AUTOSAR MCAL Development

REVISION HISTORY

Version. No.	Modified by	Reason for change
0.1	Hung Le	Initial

TABLE OF CONTENT

1. Introduction

1.1. Purpose

The goal of this document is to create a uniform coding habits among development team so that the code written by different developers are clean, easy to read, review and maintain. A mixed coding style is harder to maintain. So, it is important to apply a consistent coding style across project. When maintaining code, it is better to conform to the style of the existing code rather than blindly follow this document or your own coding style.

1.2. Scope

This document describes general software coding standard for code written in C language. Each chapter in the document specifies certain aspect of the coding guidelines, like

- **Chapter #2**: The naming convention to be used for naming the variables, Macros, typedefs, functions etc.
- **Chapter #3:** Coding style to be adhered, including usage of space, blank lines, indentation in the source code.
- **Chapter #4:** Coding rules for declarations of functions, structures, usage of pre-processing directives etc.
- **Chapter #5:** And some recommendations and best practices that are followed.

Each item has four sub-headings, which are explained below

Sub-Heading	Description
Rule	Describes the item that is to be followed
Example	Gives an example for the said item. In certain cases, gives both Compliant and Non-Compliant versions
Rationale	The rationale behind following this item
Verification Method	The method used to verify whether the item is followed

1.3. Reference

[1] AUTOSAR Specification of C Implementation Rules - v1.0.5: “AUTOSAR_TR_CImplementationRules.pdf”

[2] https://barrgroup.com/Embedded-Systems/Books/Embedded-C-Coding-Standard

[3] AUTOSAR_SWS_BSWGeneral.pdf – Part of ASR 4.2.2

[4] AUTOSAR_SWS_StandardTypes.pdf - Part of ASR 4.2.2

2. Naming Convention

This section specifies the naming conventions to be followed for file, functions, variables, constants, macros and types.

2.1. General

2.1.1. NAME_GENERAL_001

Rule:
Only below characters should be used in naming of a identifier in the source code.

- a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _

Example:
Not required

Rationale:
Not required

Verification method:
Manual review

2.1.2. NAME_GENERAL_002

Rule:

- Identifiers shall not contain the '_' character twice in succession.
- Digits and underscores are allowed but not at the start.

Example:
Not required

Rationale:
Not required

Verification method:
Manual review

2.1.3. NAME_GENERAL_003

Rule:
Avoid using the same variable/constant name in Inner scope. Names of Identifiers should be unique and should not be reused in different scope.

Example:
Not required

Rationale:
Reduce Complexity of code.

Verification method:
Manual review

2.2. File Naming

2.2.1. NAME_FILE_001

Rule:
The naming of the C source files shall be in below format:
1. Implementation file: <Msn>.c
2. Interface header file: <Msn>.h
3. Private header file (optional): <Msn>_<Name>.h
4. Configuration header file: <Msn>_Cfg.h
5. Configuration Data file: <Msn>_Cfg.c
6. Module Interrupt Routine file (optional): <Msn>_Irq.c

Implementation files shall always have a file name extension of '.c'.
Header files shall always have a file name extension of '.h'.

Example:
Compliant:
Adc.c
Adc.h
Adc_InternalFct.h

Not compliant:
adc.c
adc.c
adc_InternalFct.h
Adc_internalFct.h

Rationale:
The file belongs to which module is easily understandable.

Verification method:
Manual review

2.3. Variable/Constant Naming

2.3.1. NAME_VAR_001

Rule:
Use Camel case convention for all Variable Name. Avoid usage of '_' in between variable name.

Example:
Compliant: STATIC unit16 l_ResultBuffer_u16;

Not compliant: STATIC unit16 l_Result_Buffer_u16;

Rationale:
Avoid usage of multiple '_' in variable name.

Verification method:
Manual Review

2.3.2. **NAME_VAR_002**

Rule:
All MCAL Modules shall label Variables according to the following scheme:
Composition of name: <prefix >_<Variable name>_<suffix>

Spelling of name: The variable name should be chosen such that it should convey/describe the functionality/purpose of the variable.

Prefix of name: The variable name should be prefixed by single keyword that describes the scope of the variable, as per the table given below

Keyword	Scope
g_	Global/Extern type
l_	Static / File local variables
f_	Function local variables
p_	For parameters of the function (there can be exception for AUTOSAR APIs)
(No Prefix)	For Structure/Union members

Suffix of name: The variable name should be suffixed based on the data type of the variable. (Also applies for Structure/Union Members)

The following is a list of common data types used in AUTOSAR mcal which should be used in variable names:

Order	Keyword	Data Type
1	pp	pointer to pointer
2	ptr	Pointer
3	st	Structure
4	un	Union
5	aa	Array
6	en	enumerated type
7	u8	Unsigned int 8
8	u16	Unsigned int 16
9	u32	Unsigned int 32
10	c	Char
11	s	String
12	fl	Float
13	db	Double
14	bool	Boolean
15	s8	Signed int 8
16	s16	Signed int 16
17	s32	Signed int 32
18	dd	derived data type

Example:
Assume that ADC is the module name in below examples

a) Function local variable of type uint32
```
static uint32 f_Val_u32;
```

b) static variable of type uint8
```
static uint32 l_Var_u8;
```

c) static pointer to type uint32
```
static uint32 *l_PtrName_ptr;
```

d) global/extern structure
```
extern AdcConfigType g_adcConfig_st;
```

Rationale:
The variable owner and its type can be understood from its name.

Verification method:
Manual review

2.3.3. NAME_VAR_003

Rule:
All AUTOSAR MCAL Modules shall label global variables according to the following scheme:
Composition of name: <Module name>_<prefix >_<Variable name>_<suffix>

There should not be any underscore in the '*Variable name*'.

Spelling of name: First letter of each word upper case, consecutive letters lower case. The Prefix should be 'g' followed by a meaningful variable name. The list of common data types and the corresponding character string, which should be used in variable names mentioned in local variable section holds good for global variables also.

Example:
Global/exported variable of type unit16
Compliant:
```
extern unit16 Adc_g_Result_u16;
```

Not compliant:
```
extern unit16 AdcResult_u16;
```

Rationale:
The variable scope, its owner and its type can be understood from its name.

Verification method:
Manual review

2.3.4. NAME_VAR_004

Rule:
Reserved identifiers, macros and functions in the standard libraries, shall not be defined, redefined or undefined. The names of macros, objects and functions in the standard libraries shall not be reused.

Example:
Not required.

Rationale:
There are some specific reserved words and function names which are known to give rise to undefined behaviours if they are redefined or undefined.

Verification method:
Manual review

2.4. Function Naming

2.4.1. NAME_FUNC_001

Rule:
Global function names are structured with upper case/lower case and the first letter should be upper case. The Function name should follow Camel case convention
Composition of name: <Module name>_<Function name>

Only one underscore between module name and function name. This shall be applicable to Static functions which are not specified as part of AUTOSAR specifications.

Example:
Compliant:
Com_NormalReset

Not compliant:
com_NormalReset
Com_normalReset
Com_Normalreset

Rationale:
The module to which the function belongs can be understood from its name.

Verification method:
Manual review

2.4.2. NAME_FUNC_002

Rule:
Static function names are structured with upper case/lower case and the first letter should be upper case. The Function name should follow Camel case convention and be prefixed by '_' (underscore)
Composition of name: _<Module name>_<Function name>

One underscore at the start of the static function name, and one more underscore between module name and function name.

Example:
Compliant:
_Com_NormalReset

Not compliant:
com_NormalReset
Com_normalReset
Com_Normalreset

Rationale:
The module to which the function belongs can be understood from its name.

Verification method:
Manual review

2.5. Type Naming

2.5.1. NAME_TYPE_001

Rule:
Enumeration elements shall be uppercase, with words separated by the underscore character. All the Enumeration should be 'typedef'. The typedef name and the Enumeration name should be the same, except Enumeration element name will have suffix '_t'

Example:
Compliant:

```
typedef enum Spi_StatusType_t  {
  SPI_UNINIT,
  SPI_IDLE,
  SPI_BUSY
} Spi_StatusType;
```

Rationale:
Readability

Verification method:
Manual Review

2.5.2. NAME_TYPE_002

Rule:
All MCAL Modules shall label self-defined i.e. specific data types according to the following scheme
Composition of type: <Module name>_<Type name>Type

Only one underscore between module name and type name.
Spelling of type: First letter of each word upper case, consecutive letters lower case.

Example:
Eep_LengthType
Dio_SignalType
Nm_StatusType

Rationale:
Readability

Verification method:
Manual review

2.5.3. NAME_TYPE_003

Rule:
All the Enumeration and Structure should be type casted using typedef keyword.

Example:
Not required

Rationale:
Code consistency.

Verification method:
Manual review

2.5.4. NAME_TYPE_004

Rule:
Prefix not required for Structure/Union members

Example:
Not required

Rationale:
Code consistency.

Verification method:
Manual review

2.6. Macro Naming

2.6.1. NAME_MACRO_001

Rule:
Constants defined as macro shall be written in upper case. Digits and underscores are allowed but not at the start.
With Exceptions for:

- Macros defined in Compiler.h for identifying the type of Compiler used

Ex: __ICCARM__

- Macros defined in Device header file.

Ex: _HC32A4A0_

- Device identification MACRO used as part of Project Pre-processor setting.

Ex: __HC32A4A0__

Example:
Compliant: #define NR_OF_ELEMENTS 10

Not compliant: #define nr_of_elements 10

Rationale:
Readability

Verification method:
Manual review

3. Coding style

This section specifies the House Styling Guide that must be followed for MCAL source code. Styling guide includes usage of White spaces, Blank lines, indentation etc.

3.1. File Structure

3.1.1. STYLE_FILE_001

Rule:
All MCAL Modules shall provide at least the following files:
1. Module header file: <Module name>.h

2. Module source file: <Module name>.c

4. Module configuration file: <Module name>_Cfg.h
(customizable data for module configuration)

4. Module configuration parameters: <Module name>_Cfg.c
(for precompile-time configuration parameters are used)

5. Module callback header file: <Module name>_Cbk.h
(if callbacks are provided to other modules)

6. Module ISR source file: <Module name>_Irq.c

Example:
Gpt.h
Gpt.c
Gpt_Cfg.h
Gpt_Cfg.c
Gpt_Cbk.h
Gpt_Irq.c

Rationale:
Not required

Verification method:
Manual review

3.1.2. STYLE_FILE_002

Rule:
The header files should not be included back to back. One header file included in another header file which is included back in the first header file.

Example:
Not required

Rationale:
Not required

Verification method:
Manual review

3.2. Comments

3.2.1. STYLE_COMMENT_001

Rule:
Comments shall comply with the ANSI-C Standard. C++ comments are not permitted.

Example:
Compliant: /* Comment */

Non-Compliant: // Comment

Rationale:
Readability & consistency

Verification method:
Manual Review

3.2.2. STYLE_COMMENT_002

Rule:
The condition is repeated as a comment after the *'#endif'/'#else if'/'#else'* in the case of *'#if'* blocks that are longer than 10 lines.

Example:

```
#if (CONDITION == TRUE)
/*
  more than 10 lines
*/
#else /* (CONDITION == TRUE) */
/*
  more than 10 lines
*/
#endif /* (CONDITION == TRUE) */
```

Rationale:
Readability

Verification method:
Manual Review

3.2.3. STYLE_COMMENT_003

Rule:
Comments generally precede a block of code that performs the algorithm or technical detail defined in comment

Example:
Not Required

Rationale:
Readability

Verification method:

Manual Review

3.2.4. STYLE_COMMENT_004

Rule:
Use the following capitalized comment markers to highlight important issues:

- "WARNING:" alerts a maintainer there is risk in changing this code.
- "NOTE:" provides descriptive comments about the "why" of a chunk of code—as distinguished from the "how" usually placed in comments
- "TODO:" indicates an area of the code is still under construction and explains what remains to be done

Example:
Not Required

Rationale:
Readability

Verification method:
Manual Review

3.2.5. STYLE_COMMENT_005

Rule:
No code shall be commented out. use #if 0 #endif to disable code segments temporarily.

Example:
Not Required

Rationale:
Readability

Verification method:
Manual Review

3.3. Indentation

3.3.1. STYLE_INDENTATION_001

Rule:
Each level of Indentation should align at multiple of 2 space (no Tabs) from start of the line.

Example:
Compliant:

```
If(contd1) {
  /*
  Statements #1
  */
} else {
  If(contd2) {
    /*
    Statements #2
    */
  } else {
```

```
    /*
    Statements #2
    */
  }
}
```

Non-Compliant:

```
If(contd1) {
/*
Statements #1
*/
} else {
If(contd2) {
/*
Statements #2
*/
} else {
/*
Statements #2
*/
}
}
```

Rationale:
Style consistency across various text editors and IDE.

Verification method:
Manual review

3.3.2. STYLE_INDENTATION_002

Rule:
All the Tabs in the file should be converted to Spaces. No Tabs should be present in *.c and *.h files (source files). The tab character (ASCII 0x09) shall never appear within any source code file

Example:
Not Required

Rationale:
Readability across various IDE and during comparison in Beyond compare

Verification method:
Manual review

3.3.3. STYLE_INDENTATION_003

Rule:
Compound statements are statements that contain lists of statements enclosed in { } braces. Every Compound statement must

1. The enclosed list should be indented one more level in the compound statement itself.
2. The opening left brace should be at the end of the line beginning the compound statement and the closing right brace should be alone on a line, positioned under the beginning of the compound statement.

3. Single statement when it is part of a control structure, such as an if-else or for statement shall also always surrounded by braces.

Example:

```
if (condition) {
  if (other_condition) {
    /* Statements */
          }
}
```

Rationale:
Style consistency

Verification method:
Manual review

3.3.4. STYLE_INDENTATION_004

Rule:
Use proper Alignment/Indentation in case of

- Multiple Arguments in a function.
- Multiple members of Structure/Union

Use whitespaces in comments to improve Alignment of comments/variables and improve Readability.

Example:

```
{
  uint32   Hello1;          /* Variable #1 */
  float    HelloWorld2;  /* Variable #2 */
  Boolean HelloC3;       /* Variable #3 */
}
```

Rationale:
Readability

Verification method:
Manual review

3.3.5. STYLE_INDENTATION_005

Rule:
Within a switch statement, the case labels shall be aligned; the contents of each case block shall be indented once from there.

Example:

```
switch(hello) {
  case 0x1:
    /* Statements */
    break;
  case 0x1:
    /* Statements */
    break;
  default:
    /* Statements */
```

```
    break;
}
```

Rationale:
Readability

Verification method:
Manual review

3.3.6. STYLE_INDENTATION_006

Rule:
Whenever a line of code is too long to fit within the maximum line width, indent the second and any subsequent lines in the most readable manner possible

Example:

```
If ((ADC_READY == l_AdcCurrentStatus) && \
  (ADC_BUSY == l_AdcCurrentStatus)) {
    /*
      Statements
    */
}
```

Rationale:
Readability

Verification method:
Manual review

3.3.7. STYLE_INDENTATION_007

Rule:
The # in a pre-processor directive shall always be located at the start of a line, though the directives themselves may be indented within a #if or #ifdef sequence.

Example:

```
   If ((ADC_READY == l_AdcCurrentStatus) && \
     (ADC_BUSY == l_AdcCurrentStatus)) {
#if (ADC_STATUS_API == STD_ON)
    /*
      Statements
    */
#else /* if (ADC_STATUS_API == STD_ON)*/
    /*
      Statements
    */
#endif /* if (ADC_STATUS_API == STD_ON) */
   }
```

Rationale:
Readability

Verification method:
Manual review

3.4. White Spaces

3.4.1. STYLE_WHITE_SPACES_001

Rule:
There shall be usage of One space in below cases

1. Each of the keywords if, while, for, switch shall be followed by one space.
2. Each comma separating function parameters shall always be followed by one space.
3. Each semicolon separating the elements of a statement shall always be followed by one space

Example:

```
If (ADC_READY == l_AdcCurrentStatus) {
 /*
  Statements
 */
}

for (i = 0u, i < 5u; i++) {
 /*
  Statements
 */
}
```

Rationale:
Style Consistency

Verification method:
Manual review

3.4.2. STYLE_WHITE_SPACES_002

There shall be usage of one space before and After in below cases

Rule:

1. Each of the assignment operators =, +=, -=, *=, /=, %=, &=, |=, ^=, ~=, and !=shall always be preceded and followed by one space.
2. Each of the binary operators +, -, *, /, %, <, <=, >, >=, ==,!=, <<, >>, ^, &&, and || shall always be preceded and followed by one space.
3. The ? and : characters that comprise the ternary operator shall each always be preceded and followed by one space.

Example:

```
l_AdcCurrentStatus = ADC_READY;

If ((ADC_READY == l_AdcCurrentStatus) {
 /* Statements */
}

      l_AdcCount++;
```

Rationale:
Style Consistency

Verification method:
Manual review

3.4.3. STYLE_WHITE_SPACES_003

Rule:
There shall be no spaces in below cases

1. Each of the unary operators +, -, ++, --, ! , and ~, shall be written without a space on the operand side.
2. The structure pointer and structure member operators (-> and . respectively) shall always be without surrounding spaces.
3. The left and right brackets of the array subscript operator ([and]) shall be without surrounding spaces, except as required by another white space rule.
4. Expressions within parentheses shall always have no spaces adjacent to the left and right parenthesis characters.
5. There shall be no spaces, after the semicolon which ends the current statement.

Example:

```
l_AdcCurrentStatus = ADC_READY;

If ((ADC_READY == l_AdcCurrentStatus) {
  /* Statements */
}

      l_AdcCount++;
```

Rationale:
Style Consistency

Verification method:
Manual review

3.4.4. STYLE_WHITE_SPACES_004

Rule:

1. The pointer operators * and & shall be written without a space on the operand side.
2. The left and right parentheses of the function call operator shall always be without surrounding spaces, except that the function declaration shall feature one space between the function name and the left parenthesis to allow that one particular mention of the function name to be easily located.

Example:

```
uint8 *hello_ptr = &hello_u8;

void Adc_GetStatus (uint32 status)
{
  /* Statements */
}

main ()
{
  /* Statements */
       Adc_GetStatus(l_AdcStatus_u32);
  /* Statements */
```

```
}
```

Rationale:
Readability

Verification method:
Manual review

3.4.5. STYLE_WHITE_SPACES_005

Rule:
There shall be no spaces in below cases

1. Each semicolon shall follow the statement it terminates without a preceding space.
2. There shall be no spaces, after the semicolon which ends the current statement. All source code lines shall end only with the Carriage Return ('LF' (ASCII 0x0A), not with the pair 'CR'-'LF' (0x0D 0x0A).

Example:
Not required.

Rationale:
Style consistently

Verification method:
Manual review

3.5. Blank Lines

3.5.1. STYLE_BLK_LN_001

Rule:
There shall be a blank line before and after each natural block of code. Examples of natural blocks of code are loops, if…else and switch statements, and consecutive declarations.

Example:
Not required.

Rationale:
Readability

Verification method:
Manual review

3.6. Miscellaneous

3.6.1. STYLE_MISC_001

Rule:
The width of all lines in a program shall be limited to a maximum of 160 characters.

Example:
Not required

Rationale:
Easy for peer reviews and code examinations.

Verification method:
Manual review

3.6.2. STYLE_MISC_002

Rule:
Do not rely on C's operator precedence rules, as they may not be obvious to those who maintain the code. To aid clarity, use parentheses (and/or break long statements into multiple lines of code) to ensure proper execution order within a sequence of operations
Unless it is a single identifier or constant, each operand of the logical AND (&&) and logical OR (||) operators shall be surrounded by parentheses.

Example:

```
if ((Var_A > 0) && (Var_B < 10)) {
  /* Statements */
}
```

Rationale:
Readability.

Verification method:
Manual review

3.6.3. STYLE_MISC_003

Rule:
The braces (or curly brackets) must be used in the following styles.

- In function body, the braces (curly brackets) shall always be in a new line. Both Open and Close braces should be in a new line.
- In other cases (Loops, Conditions etc.), the open braces shall start on the same line as the loop or Condition and Close braces should be in a new line.

Example:

```
void Adc_GetStatus (uint32 status)
{
  /* Statements */
}
```

```
if ((Var_A > 0) && (Var_B < 10)) {
  /* Statements */
}
```

Rationale:
Code style consistency.

Verification method:
Manual review

4. Coding Rules

This section specifies the coding rules that must be strictly followed to maintain the standard of MCAL modules. This section includes guidelines from AUTOSAR Coding Standard, MISRA etc.

4.1. Conformance to External Rules

4.1.1. RULES_EXT_001

Rule:
If a MISRA warning has occurred, analyse the warning to know if that error can be avoided by modifications in the code. Or that violation of MISRA rule shall be commented and reasoned at the corresponding code line.

Example:
Not required

Rationale:
Not required

Verification method:
MISRA Checker Tool

4.1.2. RULES_EXT_002

Rule:
If a violation of "AUTOSAR C programming guideline" has occurred, analyse to know if that error can be avoided by modification in the source code or comment the reason at corresponding line(s) of code.

Example:
Not required

Rationale:
Not required

Verification method:
Manual review

4.1.3. RULES_EXT_003

Rule:
All approved MISRA Exceptions should be preceded (and followed) by MISRA Exception MACROs. The MACROs will be substituted with the appropriate Exception for the MISRA Checker Tool being used.

Compliant Naming:

Example:

```
/* For IAR PROJECT */
MISRA_EXCEPTION_17_4 /* Exception Enabled for MISRA. Rule 17.4  */
 MISRA_DEFAULT /* Back to Normal MISRA Checker context */
(Exceptions MACRO should be added in file 'Çompiler.h')
```

```
MISRA_EXCEPTION_17_4
/* Pointer Arithmetic */
l_hello_ptr++;
MISRA_DEFAULT
```

Rationale:
Not required

Verification method:
MISRA Checker Tool

4.2. Environment

4.2.1. **RULES_ENVR_001**

Rule:
The software modules must be developed to be compliable for all Compiler platforms without any changes. Any necessary compiler specific instructions (e.g. memory locators, pragmas, use of atomic bit manipulations etc.) must be exported to macros and include files.

Example:
Not required

Rationale:
MCAL Modules should be Compiler independent. To avoid major rework due to change in compiler and processor specific changes.

Verification method:
Manual review

4.2.2. **RULES_ENVR_002**

Rule:
All MCAL Modules shall not use compiler or platform specific keywords directly.

Example:
If specific keywords are needed, they shall be redefined (mapped) as follows:

```
#define STATIC static
```

Rationale:
To avoid major rework due to change in compiler and processor specific changes.

Verification method:
Manual review

4.2.3. **RULES_ENVR_003**

Rule:
The pre-compile time configuration data shall be captured either as #defines or as constants. It is fixed before compilation starts. The configuration of the SW element is done at source code level.

Example:
Not required

Rationale:
AUTOSAR requirement

Verification method:
Manual review

4.2.4. **RULES_ENVR_004**

Rule:
While using object/library files (from third party, customer, etc.) it shall be confirmed that it was produced with same compilers and version which is used in the project.

Example:
Not required

Rationale:
Correct integration

Verification method:
Manual review

4.2.5. **RULES_ENVR_005**

Rule:
Compiler specific header files, libraries and intrinsic functions shall not be used. However, generic Compiler functionalities can be made available as MACRO in Compiler.h. These MACROs can be used in source code and when using another Compiler, the MACRO will be substituted with appropriate keyword of the new Compiler being used (for the same functionality)

Example:
For IAR Compiler: `#define WEAK __weak`
For GHS Compiler: `#define WEAK PRAGMA(weak)`

Rationale:
Portability and understandability

Verification method:
Manual review

4.2.6. **RULES_ENVR_006**

Rule:
ANSI-C does not support inline assembler and there is no general keyword for the compilers (GHS, NEC compiler). Therefore a macro shall be added in the compiler abstraction to simplify porting to different compilers.

Example:
```
/* Not compliant */
__asm(“hault”);

/* Compliant */
ASM_HAULT();
```

Rationale:
To have a compiler independent code.

Verification method:
Manual review

4.2.7. **RULES_ENVR_007**

Rule:

- Avoid extensive usage of Assembly language in source code.
- All usage of assembler shall be documented
- Assembler instructions shall only be introduced using the asm declaration.
- Assembly language shall be encapsulated and isolated.

Example:
Not required.

Rationale:
Inline assembly code restricts the portability of the code.

Verification method:
Manual review

4.2.8. **RULES_ENVR_008**

Rule:
Assumption about atomic data access shall be documented in “Design” document.

* On 32 bits CPU architectures access to naturally aligned 8, 16 and 32-bits values is atomic. Therefore, no lock for direct simple accesses is needed (but read-modify-write instructions, e.g. ++, --, +=, -=, etc. are not atomic and still have to be protected).

Example:
Not required.

Rationale:
To have a compiler independent code.

Verification method:
Manual review

4.3. File Inclusion

4.3.1. **RULES_INCL_001**

Rule:
Header files which are part of predefined program libraries shall be included using <>.

Header files which are a part of the source code generated in the software project shall be included with “”.

Example:
Compliant:

```
#include <string.h>
#include "Eep_Read.h”
```

Not compliant:

```
#include “string.h”
#include <Eep_Read.h>
```

Rationale:
Library and project files shall be easily identified.

Verification method:
Manual review

4.3.2. RULES_INCL_002

Rule:

- Each header file shall protect itself against multiple inclusion.
- A “.c” file shall not be included in another file.

Example:
Compliant:

```
#ifndef FILENAME_H
#define FILENAME_H
.....
#endif /* FILENAME_H */
```

Rationale:
Avoid multiple re-definitions. Inclusion of own header file is the only way of allowing the compiler to check for consistency between declaration and definition of global variables and functions.

Verification method:
Manual review

4.3.3. RULES_INCL_003

Rule:
Each module shall include its own header file. Each module shall also perform Software Major & Minor version check between *.c to *.h file. If the values are not identical to values expected, an error shall be reported.

Example:

```
/*** Eth.h ***/
#define ETH_SW_MAJOR_VERSION (1u)
#define ETH_SW_MINOR_VERSION  (0u)

/*** Eth.c ***/
#include "Eth.h"

#define ETH_SW_MAJOR_VERSION_C (1u)
#define ETH_SW_MINOR_VERSION_C (0u)

#ifndef HC32A4A0_MCAL_VERSION_NOCHECK
#if ((ETH_SW_MAJOR_VERSION != ETH_SW_MAJOR_VERSION_C) || \
   (ETH_SW_MAJOR_VERSION != ETH_SW_MINOR_VERSION_C))
 #error "Inconsistent Software Versions of Eth.c and Eth.h"
#endif
```

Rationale:

Inter Module Checks is necessary to avoid integration of *.h and *.c of different software version.

Verification method:
Manual review

4.3.4. RULES_INCL_004

Rule:
Each Module shall perform Inter Module Checks through pre-processor checks. If the values are not identical to values expected, an error shall be reported.

Example:
```
#include "Det.h"

#if ((DEM_AR_RELEASE_MAJOR_VERSION != ETH_AR_RELEASE_MAJOR_VERSION_C) || \
   (DEM_AR_RELEASE_MINOR_VERSION != ETH_AR_RELEASE_MINOR_VERSION_C))
 #error "Inconsistent AUTOSAR Version Numbers of Dem.h and Eth.c"
#endif
```

Rationale:
Inter Module Checks is necessary to avoid integration of incompatible modules.

Verification method:
Manual review

4.4. Comments/Documentation

4.4.1. RULES_COMMENT_001

Rule:
Comments in header files should only describe the externally visible behaviour of functions being documented.

Example:
Not required

Rationale:
Exposing the internals may lead to creating unnecessary dependencies.

Verification method:
Manual review

4.4.2. RULES_COMMENT_002

Rule:
The SRS ID should be mentioned/included as comments for below items.

- Function Definition and Declaration
- Typedef definitions
- MACRO definitions
- Variable declaration
- Loops, Condition etc
- Code blocks/snippets.

Example:

Not required

Rationale:
Required for establish Bi-directional traceability between requirement and code block.

Verification method:
Manual review

4.5. Types

4.5.1. RULES_TYPES_001

Rule:
Enumeration types must be used instead of integer types and constants to select from a limited series of choices.

Example:
Compliant:

```
typedef enum Can_HwFaultStatus_t {
  CAN_HW_NO_FAULT,
  CAN_HW_SHORT_TO_GND,
  CAN_HW_SHORT_TO_VCC,
} Can_HwFaultStatus;

Can_HwFaultStatus Can_FaultStatus;
```

Not compliant:

```
#define CAN_HW_NO_FAULT          (uint8)0u
#define CAN_HW_SHORT_TO_GND      (uint8)1u
#define CAN_HW_SHORT_TO_VCC      (uint8)2u
uint8 Can_HwFaultStatus;
```

Rationale:
Enhances debugging, readability and maintenance. Compiler or static check tool can be set to generate a warning when the *'enum'* type variable is used in *'switch'* statement and all enumerators are not used as case.

Verification method:
Manual review

4.5.2. RULES_TYPES_002

Rule:
Initialization of Enumeration elements shall be by either:

- Not specifying any constants
- Specifying all the constants (or)
- Specifying only the first Member

Example:
Compliant:

```
typedef enum Spi_StatusType_t  {
  SPI_UNINIT = 0u,
  SPI_IDLE,
```

```
  SPI_BUSY
} Spi_StatusType;
```

Rationale:
Readability

Verification method:
Manual Review

4.5.3. RULES_TYPES_003

Rule:
Integer values of the enumeration elements must not be used in calculations.

Example:

```
typedef enum Element_t {
  ELEMENT_1,
  ELEMENT_2,
  ELEMENT_3
} Element;

Element Variable;
```

Not compliant: `Variable = ELEMENT_1 + 3;`

Rationale:
The integer value of an enumeration element can be changed when another element is added at later point of time during development.

Verification method:
Manual review

4.5.4. RULES_TYPES_004

Rule:
All Modules shall use the following data types instead of native C data types:

1. Fixed size guaranteed
Data type: Representation
uint8: 8 bits
uint16: 16 bits
uint32: 32 bits
sint8: 7 bits + 1 bit sign
sint16: 15 bits + 1 bit sign
sint32: 31 bits + 1 bit sign

2. Minimum size guaranteed, best type is chosen for specific platform (only allowed for module internal use, not for API parameters)
Data type: Representation
uint8_least: At least 8 bits
uint16_least: At least 16 bits
uint32_least: At least 32 bits
sint8_least: At least 7 bits + 1 bit sign
sint16_least: At least 15 bits + 1 bit sign
sint32_least: At least 31 bits + 1 bit sign

Above integer types will be placed in the central AUTOSAR type header ('Platform_Types.h') which is defined individually for each supported platform. Avoid using classic types with '_t'

Example:
Not required

Rationale:
Not required

Verification method:
Manual review

4.5.5. RULES_TYPES_005

Rule:

- *Std_ReturnType:* This type can be used as standard API return type which is shared between the RTE and the BSW modules. The Std_ReturnType shall normally be used with value E_OK or E_NOT_OK. If those return values are not sufficient user specific values can be defined by using the 6 least specific bits.

- Because E_OK is already defined within OSEK, the symbol E_OK has to be shared.

Enumeration	
E_OK	0x00u
E_NOT_OK	0x01u

- The symbols STD_HIGH and STD_LOW are used to represent physical state HIGH and LOW

Enumeration	
STD_HIGH	0x01u
STD_LOW	0x00u

- The symbols STD_ACTIVE and STD_IDLE shall be used to define Logical state ACTIVE and IDLE.

Enumeration	
STD_ACTIVE	0x01u
STD_IDLE	0x00u

- The symbols STD_ON and STD_OFF are used along with COMPILER Macro.

Enumeration	
STD_ON	0x01u
STD_OFF	0x00u

Example:
Not required.

Rationale:
Many symbols (like OK, NOT_OK, ON, OFF) are already defined and used within legacy software. These conflicts ('redefinition of existing symbol') are expected.

Verification method:
Manual review

4.5.6. **RULES_TYPES_006**

Rule:
For simple logical values, for their checks and for API return values the AUTOSAR type boolean, defined in Platform_Types.h, can be used. The following values are also defined
FALSE = 0
TRUE = 1

Example:
Not required.

Rationale:
Compiler independency

Verification method:
Manual review

4.5.7. **RULES_TYPES_007**

Rule:
Variables for loop counters must be declared in the generic type (i.e. *'uint'*).

Example:
Not required

Rationale:
Compiler independency

Verification method:
Manual review

4.6. Declarations and Definitions

4.6.1. **RULES_DEFN_DECL_001**

Rule:

- The *'register'* storage class specifier must not be used.
- The keyword *'auto'* shall not be used (*'auto'* is the default storage class for local variables).

Example:
Not compliant: register uint8 f_Adc_Data_u8 = 0u;

Compliant: static uint8 f_Adc_Data_u8 = 0u;

Rationale:
Compiler technology is now capable of optimal register placement and some compilers ignore the *'register'* storage specifier.
The *'auto'* keyword is an unnecessary historical feature of the language.

Verification method:
Manual review

4.6.2. RULES_DEFN_DECL_002

Rule:
A structure or enumeration type variable shall not be defined in the type definition itself. It means each self-defined type must have an explicit type declaration even if there is only one variable of this type. All new structures, unions, and enumerations shall be named via a typedef

Example:
Not compliant:

```
Enum {
  DIR_RIGHT,
  DIR_LEFT
} Ctl_DriverSide;
```

Compliant:

```
typedef enum Ctl_DriverSideType_t {
  DIR_RIGHT,
  DIR_LEFT
} Ctl_DriverSideType;

Ctl_DriverSideType          Ctl_DriverSide;
```

Rationale:
Readability

Verification method:
Manual review

4.6.3. RULES_DEFN_DECL_003

Rule:
Multiple variable declarations shall not be allowed on the same line.

Example:
Compliant:

```
uint8 l_CanRxData_u8;
uint8 l_CanTxData_u8;
```

Not compliant:

```
uint8 l_CanRxData_u8, l_CanTxData_u8;
```

Rationale:
Readability

Verification method:
Manual review

4.6.4. RULES_DEFN_DECL_004

Rule:
Care should be taken to initialize all the variables before it is used. (Applies for static and function local variables too)

Example:

Compliant:

```
uint8 l_CanRxData_u8 = 0u;
```

Not compliant:

```
uint8 l_CanRxData_u8;
```

Rationale:
Avoid usage of variable before initialization (avoid garbage value)

Verification method:
Manual review

4.6.5. RULES_DEFN_DECL_005

Rule:
A global/exported object or function of a module shall be declared with *'extern'* only in that module header file.

Example:
Using the CAN module object/function in "Apl_Process.c" like below is not compliant

Not compliant:

```
extern uint8 Can_RxData;

Apl_ProcessCan(void) {
  if (0 == Can_RxData) {
    /* do some action */
  }
}
```

Compliant:

```
#include "Can_Receive.h"

Apl_ProcessCan(void) {
  if (0 == Can_RxData) {
    /* do some action */
  }
}
```

Rationale:
Allow access to global functions and variables by including the header file of the defining module.

Verification method:
Manual review

4.6.6. RULES_DEFN_DECL_006

Rule:
Each implementation file shall have the corresponding header file and an optional private header file.

Example:
Not required

Rationale:
Private functions and variables shall not be sharable to other modules.

Verification method:
Manual review

4.6.7. **RULES_DEFN_DECL_007**

Rule:
Variables and functions (other than macros) shall not be defined within in an '.h' file. They shall be defined within the module's C file.

* Exception: Definitions of inline functions in the header file are permitted.

Example:
In 'Icu.h', we declare "uint8 Icu_IntCounter". Then in 'Icu.c' and 'Icu_Cfg.c', both of them include 'Icu.h'. So, when compilling, there will be error. To avoid, we shall declare "uint8 Icu_IntCounter" in 'Icu.c' and 'extern' that variable in 'Icu.h' or 'Icu_Cfg.c'.

Rationale:
Prevent multiple declarations (i.e. linker problem).

Verification method:
Manual review

4.6.8. **RULES_DEFN_DECL_008**

Rule:
Direct use of compiler and platform specific inline keywords like *'__inline__'* or *'_inline'* are not allowed. To define an inline function macro 'LOCAL_INLINE' shall be used.

Example:
Not required

Rationale:
To encapsulate all needed keywords and properties to define an inline function.

Verification method:
Manual review

4.6.9. **RULES_DEFN_DECL_009**

Rule:
Declarations of functions shall always be stated with detailed parameter list, i.e. the type and a practical designation of the relevant parameters. Designator names in '.c' and '.h' file shall be identical.

Example:
```
extern Std_ReturnType Eep_Erase (
  Eep_AddressType EepromAddress,
  Eep_LengthType Length
);
```

Rationale:
Readability and correct use of APIs

Verification method:
Manual review

4.6.10. RULES_DEFN_DECL_010

Rule:
If a function parameter is not going to be changed inside the function, then it should be made a constant parameter. There can be deviation if the function is defined by AUTOSAR requirement.

Example:
```
void func(const int p_FirstNumber, const int p_SecondNumber)
```

Rationale:
Readability

Verification method:
Manual review

4.6.11. RULES_DEFN_DECL_011

Rule:
All MCAL Modules shall apply the following naming rule for enabling/disabling the detection and reporting of development errors:

'<MODULENAME>_DEV_ERROR_DETECT'

Example:
```
In 'Eep_Cfg.h':
#define EEP_DEV_ERROR_DETECT ON /* detection module wide enabled */
…

In source 'Eep.c':
#include "Eep_Cfg.h"
…

#if (EEP_DEV_ERROR_DETECT == ON)
 /*
 development errors to be detected
 */
#endif /* (EEP_DEV_ERROR_DETECT == ON) */
```

Rationale:
Uniformity

Verification method:
Manual review

4.6.12. RULES_DEFN_DECL_012

Rule:
The *'volatile'* keyword should be used to indicate variables that may be modified outside of the program's control (i.e., any register values that are set or cleared by hardware, or memory that is dual-port and may be modified by an external device).

Example:
Not required

Rationale:
If this is not done, the optimizer may make optimizations that will result in incorrect operation.

Verification method:
Manual review

4.6.13. **RULES_DEFN_DECL_013**

Rule:
Type casting of the variables should be done correctly.

Example:
Not compliant:
(uint16)0x10000

Compliant:
(uint16)0xFFFF
(uint32)0x10000

Rationale:
To avoid bugs in implementation

Verification method:
Manual review

4.6.14. **RULES_DEFN_DECL_014**

Rule:
Bit-fields shall not be defined within signed integer types.

Example:
Not required

Rationale:
Compiler independency

Verification method:
Manual review

4.7. Control Statement Expressions

4.7.1. **RULES_EXPR_001**

Rule:
While performing masking operation, make sure of the mask value to be used and its impact for other functionalities.

Example:
Not required

Rationale:

To avoid bugs in implementation

Verification method:
Manual review

4.7.2. **RULES_EXPR_002**

Rule:
Check for wrong usage of “&&", "||" while performing mask operations.

Example:
Not required

Rationale:
To avoid bugs in implementation

Verification method:
Manual review

4.7.3. **RULES_EXPR_003**

Rule:
During updating for similar part of code for timer unit/channel interrupt handlers, check the unit/channel numbers used carefully, because there can be typo mistakes in updating correct values of those numbers.

Example:
Not required

Rationale:
To avoid bug in implementation

Verification method:
Manual review

4.7.4. **RULES_EXPR_004**

Rule:
When evaluating the equality of a variable against a constant, the constant data shall always be placed to the left of the equal-to (==) operator.

Example:
Not required

Rationale:
To avoid logical error.

Verification method:
Manual review

4.7.5. **RULES_EXPR_005**

Rule:
Considering C integer boundaries (overflows & underflows) while handling arithmetic operation.

Example:

- Overflow of unsigned integers:
32 bits integer types can hold certain ranges of values.

So if we have two unsigned integer types each with the value of 2147483648 (a & b):
a + b = 4294967296
which is larger than the maximum value that can be represented in an unsigned integer type. This is called an integer overflow.

- Underflow of unsigned integers:
unsigned int a, b;
a = 0
b = a - 1

The value of b is -1, which is below than the minimum possible value that can be stored this is called an integer underflow.

- Overflow/Underflow of Signed Integers:
The signed integer two's compliment representation in binary will have value, padding & sign bits. The sign bit represents the sign of the integer 0 for positive and 1 if the number is negative. When an overflow or underflow condition occurs on signed integers the result will wrap around the sign and causes a change in sign.

For example:
A 32 bits number 2147483647 = 0x7FFFFFFF in hex.
If we add 1 to this number, it will be 0x80000000 which is equivalent to -2147483648 decimal.

With signed addition or subtraction, you can overflow the sign boundary by causing a positive number to wrap around 0x80000000 and become a negative number. You can also underflow the sign boundary by causing a negative number to wrap below 0x80000000 and become a positive number.

Note: When result of arithmetic operation is stored in HW registers, the maximum value which register can hold to be considered before loading result of arithmetic operation into a register. If result value exceeds the limit which register can hold, work around to be carried out before loading result value into a register.

Rational:
A numeric overflow or underflow that occurs early in a block of code can lead to a subtle series of cascading faults; not only is the result of a single arithmetic operation tainted, but every subsequent operation using that tainted result introduces a point where an attacker might have unexpected influence.

Verification method:
Manual review

4.7.6. **RULES_EXPR_006**

Rule:
Avoid using dynamic objects or variables.

Example:
Don't use function *'malloc()'* to allocate memory in program.

Rationale:
It's helpful for easily control memory.

Verification method:
Manual review

4.7.7. **RULES_EXPR_007**

Rule:
No implicit type conversions.

Example:

```
extern void foo(uint8 ucVal);
uint8  Value_u8;
uint16 Value_u16;
sint16 ssValue_u16;
void func(void) {
/* Compilant */
  foo(Value_u8);
  foo((uint8)Value_u16);

/* Not compliant */
  foo(Value_u16);
  foo(ssValue_u16);
}
```

Rationale:
Implicit type conversion may lead to data loss.

Verification method:
Manual review

4.7.8. **RULES_EXPR_008**

Rule:
None of the bit-wise operators (i.e. '*&*, |, ~, ^, <<, >>') shall be used to manipulate signed integer data.

Example:
Not required

Rationale:
Compiler independency

Verification method:
Manual review

4.7.9. **RULES_EXPR_009**

Rule:
Signed integers shall not be combined with unsigned integers in comparisons or expressions.

Example:
Not required

Rationale:
Compiler independency

Verification method:
Manual review

4.8. Control Flow

4.8.1. RULES_CTRLFLOW_001

Rule:
Any *'if'* statement shall end with an *'else'* clause.
If-else if-else statements should have the below form.

Example:

```
if (condition) {
  statements;
} else if (condition) {
  statements;
} else if (condition) {
  statements;
} else {
  statements;
}
```

Rationale:
Understandability

Verification method:
Manual Review

4.8.2. RULES_CTRLFLOW_002

Rule:
Switch statement should have below form. Verify for *'break'* between each *'case'* statement, unless the *'case'* statements need to be executed in sequence in switch statements.

* All switch statements shall have the default case implemented, even if it is left empty apart from the break statement.

Example:

```
switch (condition) {
  case ABC:
  case DEF:
    statements;
    break;
  case XYZ:
    statements;
    break;
  default:
    statements;
    break;
}
```

Rationale:
Unnecessary code execution may lead to wrong functionality.

Verification method:
Manual review

4.8.3. **RULES_CTRLFLOW_003**

Rule:
For Loop should have below form. Avoid using comma operator in For Loop Initialization or Update clauses.

Example:

```
for (initialization; condition; update) {
  statements;
}
```

Rationale:
Unnecessary code execution may lead to wrong functionality.

Verification method:
Manual review

4.9. Functions

4.9.1. **RULES_FUNC_001**

Rule:
All services and APIs of a module shall be multicore re-entrant.

* Re-entrant definition (single core): A function is re-entrant if it can be interrupted somewhere in its execution and then safely called again before its previous invocation completes executing, produce correct results for both invocations.

* Re-entrant definition (multiple cores): A function is re-entrant if it can be called on one core while it is already executing on any other core and still produce correct results for all calls.

Consequence:
- Must not use any static/global variable without resource protection (lock) if data consistency is required. Accesses from services (APIs) or processes or especially interrupt routines have to be considered.
- No assumptions about priorities and scheduling possible (e.g. for interrupt code).
- Must not modify its owned code.
- Must not call other non-re-entrant function(s).

Example:
Not required

Rationale:
To avoid errors in execution

Verification method:
Manual review

4.9.2. **RULES_FUNC_002**

Rule:
There should be only one return statement in function. i.e. the function should have only one exit point.

Example:
Not required

Rationale:
MISRA C requirement

Verification method:
MISRA Checker

4.9.3. **RULES_FUNC_003**

Rule:
Functions (other than macros) shall not be defined within in a ".h" file. Exception: Definitions of inline functions in the header file are permitted.

Example:
Not required

Rationale:
Not required

Verification method:
Manual review.

4.9.4. **RULES_FUNC_004**

Rule:
Declaration and definition of local functions shall have the storage-class specifier "static". Local functions are visible only inside the module.

Example:
Not required

Rationale:
Not required

Verification method:
Manual review.

4.10. Pointers and Arrays

4.10.1. **RULES_PTR_001**

Rule:
Avoid extensively usage of of pointers

Example:
Just use one global pointer to point to value of configuration in configuration files (*'<Msn>_Cfg.c', '<Msn>_PBcfg.c'*). Don't or limit using local pointer that refers to structures in global configuration pointer. E.g.
- ICU module just uses pointer 'Icu_pCfgPtr'
- ADC module just uses pointer 'Adc_pCfgPtr'

Rationale:
To avoid errors in implementation

Verification method:
Manual review

4.10.2. **RULES_PTR_002**

Rule:
Pointers shall be checked before being de-referenced.

Example:

```
void Mod_FunctionX(uint8 *p_Data1, uint8 *p_Data2) {
  /* Compliant */
  if (p_Data1 != NULL) {
    *p_Data1 = 0u;
  }

  /* Not compliant */
  *p_Data2 = 0u;
}
```

Rationale:
Avoid undefined behaviour if a pointer is NULL.

Verification method:
Manual review

4.10.3. **RULES_PTR_003**

Rule:
A *'typedef'* shall be used to simplify program syntax when declaring function pointers.

Example:

```
typedef uint8 (*Mod_FuncX_t)(void);
Mod_FuncX_t  Mod_MyFunc;
```

Rationale:
Improved readability

Verification method:
Manual review

4.10.4. **RULES_PTR_004**

Rule:
To address a structure's fields via a pointer to the structure, use the notation:
'Ptr->Field' rather than '(*ptr).field'

Example:
Not compliant: `if (0 == (*DataPtr).Field)`

Rationale:
Improved readability

Verification method:
Manual review

4.10.5. RULES_PTR_005

Rule:
Functions shall not call themselves, either directly or indirectly.

Example:
Not required.

Rationale:
As the stack space is limited resource, use of recursion may lead to stack overflow at run-time. It also may limit the scalability and portability of the program. Recursion can be replaced with loops, iterative algorithms or worklists.

Verification method:
Manual review

4.11. Structures and Unions

4.11.1. RULES_STRUCT_001

Rule:
All members of a structure shall be named and shall only be accessed via their names. The members shall have suffix (as per Naming conventionName_Var_002) and no prefix.

Example:

```
typedef struct Mod_Param_t {
  uint8   ucIndex_u8;
  uint32 ulValue_u32;
} Mod_Param;

Mod_Param Param_st;

Not compliant:  *((uint8 *)&Mod_Param + 1) = 0u;
```

Rationale:
Avoid reading/writing to unnamed locations in memory.

Verification method:
Manual review

4.11.2. RULES_STRUCT_002

Rule:
Appropriate care shall be taken to prevent the compiler from inserting padding bytes within struct or union types.

Example:
Not required.

Rationale:
Minimize the size of structures or union.

Verification method:
Manual review

4.11.3. **RULES_STRUCT_003**

Rule:
Appropriate care shall be taken to prevent the compiler from altering the intended order of the bits within bit-fields.

Example:
Not required.

Rationale:
To avoid reading/writing to wrong bit field.

Verification method:
Manual review

4.12. Pre-processing Directives

4.12.1. **RULES_PREPROCESS_001**

Rule:
Direct values should be replaced by macros (*'#define'*) or constants (*'const'*). *'#define'* statements should be used preferably.

Example:
```
#define TIMEOUT  4
...
if (timerValue >= TIMEOUT)
```

Rationale:
The values of the constants can be changed if required at one place, instead of changing everywhere where referred and avoid making errors.

Verification method:
To be automated

4.12.2. **RULES_PREPROCESS_002**

Rule:
Only the following pre-processor directives shall be used: '#if', '#elif', '#else', '#endif', '#define', '#include', '#error'. Compiler MACRO version of #pragma' can be used. '#undef', '#ifdef', '#ifndef' shall not be used.

However, there are few exceptions for this like

- Using #ifndef for avoiding multiple time file Inclusion.

Example:
Not required.

Rationale:
To avoid using compiler specific pre-processor directives.

Verification method:
Manual review

4.12.3. RULES_PREPROCESS_003

Rule:
Compiler switches shall be compared with defined values. Simple checks if a compiler switch is defined shall not be used. In general, 'STD_ON' and 'STD_OFF' shall be used to switch functionality on or off. These symbols and their values are defined in 'Std_Types.h'.

Example:
Compliant: #if (EEP_DEV_ERROR_DETECT == STD_ON)

Not compliant: #ifdef EEP_DEV_ERROR_DETECT

Rationale:
To avoid errors in implementation, if the defined value for a compiler switch is not intended.

Verification method:
Manual Review

4.12.4. RULES_PREPROCESS_004

Rule:
Macros used in place of functions should have the same notation as a function.

Example:
#define Wdg_Trigger() (WD_TIM = 0xff)

Rationale:
Not required

Verification method:
Manual Review

4.12.5. RULES_PREPROCESS_005

Rule:
Macros shall not use global names, since the global names may be hidden by a local declaration.

Example:
Not compliant:

```
/* Adc_GucStatus is global variable holds current status of ADC driver */
#define CHANGE_STATUS(x) {Adc_GucStatus = (x)}
```

Rationale:
To avoid hidden data flow

Verification method:
Manual Review

4.12.6. RULES_PREPROCESS_006

Rule:
If a macro definition represents several statements, the entire statement shall be encapsulated within curly brackets. For multi-lines macro definitions use the '\' symbol at the end of line. E.g.

```
{ \
  Statement 1; \
```

```
  Statement 2; \
}
```

Example:

```
#define Swap(x,y,h) { \
 h = x; \
 x = y; \
 y = h; \
}
```

Rationale:
Readability

Verification method:
To be automated

4.12.7. RULES_PREPROCESS_007

Rule:
In the pre-compile check for entire function, only published macro as specified in SWS should be used.

Example:
Not required

Rationale:
If the function is removed because of pre-compile check defined by design, the user is not aware and may use the function in applications, which leads to error.

Verification method:
Manual Review

4.12.8. RULES_PREPROCESS_008

Rule:
In order to avoid hidden unintended type conversions, the explicit casting in macros for macro parameters shall not be allowed. If type casting is needed, it shall be done explicitly by the macro caller. Exceptions shall be documented.

Examples:

Not compliant:

```
/* A 32 bits value parameter will be implicitly converted to ‘uint16’ */
#define WRITE_REGISTER_16BIT(address, value) \
                                ((*(volatile uint16*)(address)) = (uint16)(value))
```

Compliant:

```
/* It will detect value parameters which are not ‘uint16’ */
/* Address casting is tolerated as an exception for the memory mapped I/O access */
#define WRITE_REGISTER_16BIT(address, value) \
                                                ((*(volatile uint16*)(address)) = (value))
```

Rationale:
The type of parameter need be checked correctly by user. Avoid the incorrect value in parameter input.

Verification method:

Manual Review

4.12.9. **RULES_PREPROCESS_009**

Rule:
Parameterized Macros shall not be used if a function can be written to accomplish the same behaviour. In case parameterized macros are used for some reason, below items should be followed.

- Surround the entire macro body with parentheses.
- Surround each use of a parameter with parentheses.
- Use each parameter no more than once, to avoid unintended side effects.
- Never include a transfer of control (e.g., return keyword).

Example:
Not Complaint:
```
#define MAX(A, B)  (A > B ? A : B)
```

Complaint:
```
#define MAX(A, B)  ((A) > (B) ? (A) : (B))
```

Rationale:
There are a lot of risks associated with the use of pre-processor defines, and many of them relate to the creation of parameterized macros. The extensive use of parentheses (as shown in the example) is important, but does not eliminate the unintended double increment possibility of a call such as MAX(i++, j++). Other risks of macro misuse include comparison of signed and unsigned data or any test of floating-point data. Making matters worse, macros are invisible at run-time and thus impossible to step into within the debugger.

Verification method:
Manual review.

4.12.10. **RULES_PREPROCESS_010**

Rule:
Published information of Module shall be provided within all header files and protect against multiple definition.

Published information elements		
Information element	***Type / Range***	***Information element description***
<MIP>_VENDOR_ID	#define/uint16	Vendor ID (vendorId) of the dedicated implementation of this module according to the AUTOSAR vendor list. The ID is the same as in HIS Software Supplier Identifications [20].
<MIP>_MODULE_ID	#define/uint16	Module ID of this module, as defined in the BSW Module List [1].
<MIP>_AR_RELEASE_MAJOR_VERSION	#define/uint8	Major version number of AUTOSAR release on which the appropriate implementation is based on.
<MIP>_AR_RELEASE_MINOR_VERSION	#define/uint8	Minor version number of AUTOSAR release on which the appropriate implementation is based on.
<MIP>_AR_RELEASE_REVISION_VERSION	#define/uint8	Revision version number of AUTOSAR release on which the appropriate implementation is based on.
<MIP>_SW_MAJOR_VERSION	#define/uint8	Major version number of the vendor specific implementation of the module. The numbering is vendor specific.
<MIP>_SW_MINOR_VERSION	#define/uint8	Minor version number of the vendor specific implementation of the module. The numbering is vendor specific.
<MIP>_SW_PATCH_VERSION	#define/uint8	Patch level version number of the vendor specific implementation of the module. The numbering is vendor specific.

Example:

```
/* File: CanIf.h */
...
/* Published information */

#define CANIF_MODULE_ID_CFG                    0x003Cu
#define CANIF_VENDOR_ID_CFG                    0x002Bu
#define CANIF_AR_RELEASE_MAJOR_VERSION_CFG     0x04u
#define CANIF_AR_RELEASE_MINOR_VERSION_CFG     0x00u
#define CANIF_AR_RELEASE_PATCH_VERSION_CFG     0x03u
#define CANIF_SW_MAJOR_VERSION_CFG             0x01u
#define CANIF_SW_MINOR_VERSION_CFG             0x02u
#define CANIF_SW_PATCH_VERSION_CFG             0x03u
```

Rationale:

This is necessary to provide unambiguous version identification for each Module and enable version cross check as well as basic version retrieval facilities.

Verification method:

Manual review.

4.12.11. **RULES_PREPROCESS_011**

Rule:
Token pasting operator (##) can be used in cases, where it will increase the readability of the source code.

Examples:
```
#define _Spi_irq_uninit(n)       _Spi_Reset(SERCOM##n);
```

Rationale:
In cases, it is known to improve the readability of the source code.

Verification method:
Manual Review

4.13. Optimization

4.13.1. **RULES_OPT_001**

Rule:
Implementations which are weak against compiler optimization shall be avoided.

Example:
A delay loop with a local variable - can be treated as some useless lines of code from the perspective of a compiler - so that loop may potentially be removed by compiler.

Rationale:
Compiler independency

Verification method:
Manual review

4.13.2. **RULES_OPT_002**

Rule:
The time for the execution of interrupt service routine should be minimal. The delay in case of nested interrupts is to be documented.

Example:
Not required

Rationale:
Performance

Verification method:
Manual review

4.13.3. **RULES_OPT_003**

Rule:
Check the possibility to optimize the configuration structure so as not to configure ‘NULL POINTER’ for unused items (e.g. channels/blocks/etc.)

Example:
Not required

Rationale:
Memory optimization

Verification method:
Manual review

4.13.4. **RULES_OPT_004**

Rule:
Pre-processor macros/configuration data shall be memory mapped to the correct section.

Example:
Not required

Rationale:
To avoid problems when memory protection is used

Verification method:
Manual review

4.13.5. **RULES_OPT_005**

Rule:
Verify if all data are located in RAM.

Example:
Not required

Rationale:
To avoid problems when memory protection is used

Verification method:
Manual review

4.13.6. **RULES_OPT_006**

Rule:
Verify if all code is located in FLASH.

Example:
Not required

Rationale:
To avoid problems when memory protection is used

Verification method:
Manual review

4.13.7. **RULES_OPT_007**

Rule:

Be sure that public inline functions can be executable from RAM, i.e. such functions should not be under START_SEC_CODE area.

Example:
Not required

Rationale:
Performance

Verification method:
Manual review

4.13.8. **RULES_OPT_008**

Rule:
Memory mapping shall not be used for variables defined within a function. Memory mapping header files shall not be included inside the body of a function.

Example:
Not required

Rationale:
It is impossible to map the local variables in a function to a memory section.

Verification method:
Manual review

4.13.9. **RULES_OPT_009**

Rule:
All interrupt locks (Critical section) have to be < 10µs (target should be <5us, max <10us). While interrupts are locked:
- No function calls
- No loops
- No branches in the program flow

* Interrupt lock time is the duration for which the interrupts are disabled (or it is the time taken between Entry and Exit of a critical section).

Example:
Not required

Rationale:
To avoid the risk that some critical interrupt service routines may miss their deadlines.

Verification method:
Manual review

4.13.10. **RULES_OPT_010**

Rule:
Verify if only the configured resources are initialized and used.

Example:
Not required

Rationale:
To avoid unexpected behaviours

Verification method:
Manual review

4.13.11. **RULES_OPT_011**

Rule:
Avoid wastage of bytes by aligning variables in memory:

- 16 bits variables have to be located at an address divisible by 2.
- 32 bits variables have to be located at an address divisible by 4.

It is recommended to sort by size (e.g. first all pointer(s), then all 32 bits variable(s), then all 16 bits variable(s) and then all 8 bits variable(s)).

* Note: Pointers are 32 bits size (also uint8*)

Example:
Not compliant:

```
uint8 XZY_xTestval1_u8;
/* Gap 3 bytes */
uint16* XYZ_adrData_pu16;
uint8 XYZ_xTestval2_u8;
/* Gap 1 byte */
uint16 XYZ_stMachine1_u16;
```

Compliant:

```
uint16* XYZ_adrData_pu16;
uint16 XYZ_stMachine1_u16;
uint8 XYZ_xTestval1_u8;
uint8 XYZ_xTestval2_u8;
```

Rationale:
Memory optimization

Verification method:
Manual review

4.14. Miscellaneous

4.14.1. **RULES_MISC_001**

Rule:
Numeric values (Magic Numbers) shall not be used in code; symbolic values shall be used instead.

Exception:
1. 0 in initialization of variables.
2. 1 in bit operations.
3. Constants used in generic math expressions like 2 in calculating the average value of two numbers.

Example:
Not required

Rationale:
Readability and maintenance

Verification method:
Manual review

4.14.2. **RULES_MISC_002**

Rule:
Do not use any Customer name in *.c or *.h file (as part of Comments or during Bug Fix)

Example:
Not required

Rationale:
Not required

Verification method:
Manual review

4.14.3. **RULES_MISC_003**

Rule:
It is better to initialize all the STATIC variable during initialization. This might avoid logical errors where the Variable is used before Initialization.

Example:
Not required

Rationale:
Avoid Logical Errors.

Verification method:
Manual review

4.14.4. **RULES_MISC_004**

Rule:
Multiple assignments shall not be done.

Example:
Not Compliant: `x = y = z;`

Rationale:
Avoid Logical Errors.

Verification method:
Manual review

4.14.5. **RULES_MISC_005**

Rule:
The use of '++' and '--' should be limited to simple cases. They shall not be used in statements where other operators occur. The prefix use is always forbidden.

Example:
Not Compliant: x =- y++;
x = --y;

Rationale:
Avoid Logical Errors.

Verification method:
Manual review

4.14.6. **RULES_MISC_006**

Rule:

- A project shall not contain any unreachable code.
- A project shall not contain unused variables.
- A project shall not contain unused type declarations.
- There shall be no unused parameters in a function.

Example:
Not required

Rationale:
Known as a DU dataflow anomaly, this is a process whereby there is a data flow in which a variable is given a value that is not subsequently used. At best this is inefficient but may indicate a genuine problem. Often the presence of these constructs is due to the wrong choice of statement aggregates such as loops.

Verification method:
Manual review

4.14.7. **RULES_MISC_007**

Rule:
All variables shall have a defined value before they are used.

Example:
Not required

Rationale:
To avoid confusion and possible use of uninitialized data members.

Verification method:
Manual review

4.14.8. **RULES_MISC_008**

Rule:
Try to avoid negative Statements in Condition checking.

Example:
Not Compliant: If(!(CONDITION))

Rationale:
To avoid confusion.

Verification method:
Manual review

5. Recommendations for MCAL Module development

This section specifies recommendation, best practices that are followed for MCAL Development.

5.1. Tool Usage

5.1.1. RECOMMENDATION_TOOL_USAGE_001

- It is recommended to use Notepad++ for Creating/Editing source files.
- When using other IDE for source file editing, make sure the IDE uses 2 spaces instead of Tab for new line inclusion. (This option is present in most of the IDEs)
- Use various color and Highlight to increase the readability of the source code.

Ex: Grey Highlight for Comments, Blue Font for C keywords etc.

- Enable View Spaces/Tabs in "View→Show Symbols→Show White Spaces and TAB"

```
/*******************************************************************************
** Function Name        : Eth_Init
** Description          : Initializes the ETH Driver Module
*******************************************************************************/
void Eth_Init ( const Eth_ConfigType *CfgPtr )
{
  /* SWS_Eth_00028 */
  P2CONST(Eth_ConfigType, AUTOMATIC, ETH_APPL_CONST) EthConfigPtr = &gEthConfig;
  VAR(uint8, ETH_APPL_DATA)              i, j;
```

5.1.2. RECOMMENDATION_TOOL_USAGE_002

Use Beyond Compare for Comparison of files. (Can use 'Ignore Minor Difference' feature to ignore comments)

5.2. Template

5.2.1. RECOMMENDATION_TEMPLATE_001

Since all required items cannot be captured precisely in document, it is recommended to use the Standard Templates for Source coding.

1	Module header file	Mod.h
2	Module source file	Mod.c
3	Module configuration file (customizable data for module configuration)	Mod_Cfg.h
4	Module configuration parameters (for precompile-time configuration parameters are used	Mod_Cfg.c
5	Module ISR source file	Mod_Irq.c
6	Module Memmap file	Mod_MemMap.h

5.3. Project Options

5.3.1. RECOMMENDATION_PROJECT_OPTION_001

When specifying include paths for the Project, use relate path addressing instead of Absolute path. This ensures smooth working of project even if compiled from different folder/computer.

Example: "../Support/inc" (Compliant)
"C:/Project/Test/Support/inc" (Non-Compliant)

5.3.2. RECOMMENDATION_PROJECT_OPTION_002

Do not suppress any Compiler related Errors/Warnings from the Project setting. All the Compiler Errors and Warnings (if any) must be address and Fixed.

5.3.3. RECOMMENDATION_PROJECT_OPTION_003

Include the files to Compile in a neat Hierarchy or Tree view. Make sure to group related *.h, *.c files using View/Customization options provided by the IDE.

5.3.4. RECOMMENDATION_PROJECT_OPTION_004

The Project should define the below symbols as part of the Project setting. It should not be defined in any source file.

- Microcontroller variant. Ex: __HC32xxxx__
- Code execution memory. EX: flash
- Any test related macro. Ex: MCAL_TEST_ENVIRONMENT
- Compiler Related macro (in case it is not defined implicitly by complier). Ex: __ghs__

5.3.5. RECOMMENDATION_PROJECT_OPTION_005

Enable the below options in the Project (if applicable/present)

- Enable "Require Prototype"
- Optimization as "Medium"
- Output file as "Executable" and not "Library"
- Language as "C"
- *C dialect* as "C99"
- Generate Debug Information (to support debugging)

5.3.6. RECOMMENDATION_PROJECT_OPTION_006

The Standard Include files (say "stdint.h") should be available in the project workspace and included from the same. Do not include Standard Include files from the Tool Installation directory.

For Ex, IAR Standard includes are available in "<PROJ_DIR>/Support/inc".

5.3.7. RECOMMENDATION_PROJECT_OPTION_007

The linker files/script should be available in the project workspace and included from the same. Do not select default linker script from Tool Installation directory.

For Ex, IAR Linker scripts are available in “<PROJ_DIR>/Support/linker”.

5.3.8. **RECOMMENDATION_PROJECT_OPTION_008**

The Device description file for the debugger should be available in the Project workspace and included from the same. Do not select default device descriptor file from Tool installation directory.

For Ex, IAR Linker scripts are available in “<PROJ_DIR>/Support/debugger”.

5.3.9. **RECOMMENDATION_PROJECT_OPTION_009**

The Flashloader file for the debugger should be available in the Project workspace and included from the same. Do not select default flashloader file from Tool installation directory.

For Ex, IAR Linker scripts are available in “<PROJ_DIR>/Support/flashloader”.