Coding Style Guide

[C Language Edition]

Embedded Software Developers' Guide to Coding Practices for SQT Projects

Version 1.0.0

Revision History

Revision History		Embedded Software Developers' Guide to Coding Practices for SQT Projects			
Rev.	Date	Description			
		Page	Summary	Approval	Creation
1.0.0	2020/4/10	-	Create new	AnhLT74	HaoNP2

1 Introduction

This document defines a coding style guide aimed at standardizing the source code for automotive development. For coding regulations that ensure software safety, we use MISRA-C:2012, and this document mainly focuses on the style of writing source code, serving as a supplementary position to the aforementioned regulation.

This document defines SQT project unique coding style guide, using "Revised Edition: Coding Practices Guide for Embedded Software Developers [C Language Edition] Ver1.0"

2 Character Encoding and Line Break Code

2.1 Character Encoding

The character encoding for creating automotive program code shall be UTF-8.

2.2 Line Break Code

The line break code for creating automotive program code shall be CR+LF.

3 Unified Coding Style

3.1 Position of Curly Braces ({ })

The position of curly braces should be standardized to make the beginning and end of blocks easier to see.

(1) Functions

The opening brace should be placed on the same line as the function or on a separate line, and the closing brace should be placed on a separate line from the code.

Example:

```
void Dem_GetVersionInfo( Std_VersionInfoType* versioninfo ) {
}
```

Example:

```
void Dem_GetVersionInfo( Std_VersionInfoType* versioninfo )
{
}
```

(2) Control Statements

The opening brace should be placed on the same line as the control statement, and the closing brace should be placed on a separate line from the code.

Example:

```
if ( NUM_OF_EVENT_DATA == data ){
}
```

(3) else Statement

The else keyword and the opening brace should be placed on the same line as the closing brace of the control statement, and the closing brace should be placed on a separate line from the code.

Example:

```
if ( NUM_OF_EVENT_DATA == data ) {
} else if ( MIN_EVENT_DATA == data ) {
} else {
}
```

3.2 Indentation

Indentation should be used to make declarations and blocks of code easier to read. The indentation should be 4 spaces (^^^^). Tabs should not be used because they can cause misalignment in different editors, so spaces should be used instead.

Note: "^" represents a half-width space

Example:

```
if ( 1 == data ) {
^^^^p = &list[0];
}
```

3.3 Use of Spaces

Insert a single space to make the code easier to read and to help spot coding errors.

(1) Functions

Example:

```
void^Dem_GetVersionInfo(^Std_VersionInfoType*^versioninfo^)^{
}
```

(2) Control statement

Example:

```
if^(^MAX_DATA^==^data^)^{
}
```

Example:

```
for^(^i=0;^i<max;^i++^)^{
}
```

(3) Expression

Example:

```
SEventStatusBuffer[i].faultDetectionCounter^=^0U;
```

3.4 Line Breaks in Continuation Lines

When an expression becomes long and readability is compromised, break the line at an appropriate position. However, to improve readability on the monitor display and when printing, if a line exceeds 100 characters, break the line at 100 characters. Place the operator at the beginning of the continuation line.

The indentation of the continuation line after a line break should be adjusted with spaces (half-width spaces) to a position that enhances readability, and the number of indentation spaces is not specified.

Note: "^" represents a half-width space

Example:

```
dataNum = DemConfigData.FreezeFrameData.FreezeFrameDataNum
^^^^^^^^+ DemConfigData.ExtendedData.ExtendedDataNum
^^^^^^^^+ DemConfigData.ExtendedData.OtherDataNum;
```

Example:

```
if (( SWC_E_SYSTEM_POWER_BROWNOUT == data1 )
^&& ( SWC_E_SENSOR_POWER_BROWNOUT == data2 )) {
```

3.5 Variable Initialization

All variables must be initialized.

3.6 Precautions When Using Global Variables

The use of global variables must be clearly identified, and their necessity and safety must be explained.

3.7 Prohibition of Implicit Type Conversion

All type conversions must be explicit.

Example:

```
uint8  data; data = (uint8)1U;
```

3.8 Conditions in If Statements

When comparing constants and variables in if statements, write the constant on the left side and the variable on the right side, as shown below.

Example:

```
/* MAX_DATA: constant / data: variable */
if ( MAX_DATA == data ) {
}
```

When the condition expression in an if statement "if (variable == constant)" is mistakenly written as "if (variable = constant)", it is interpreted as an assignment to the variable and does not result in a compile error. This can delay the detection of bugs. By writing the condition expression in the if statement as "if (constant == variable)", a mistaken "if (constant = variable)" will result in a compile error, allowing the bug to be detected at compile time.

3.9 Prohibition of Unconditional Jumps

Do not use goto statements or jump processing.

3.10 Prohibition of Recursive Processing

Do not use recursive processing.

3.11 Return Statements in void Functions

Do not write return statements in void functions. Writing return statements in void functions can sometimes cause the compiler to generate unexpected code.

3.12 Permission for Mid-Function Return

MISRA-C:2012 Rule 15.5 recommends that "a function should have a single exit point at the end of the function." This recommendation is based on the consideration that having multiple exit points in a function can cause control flow to be interrupted in the middle of processing or require different handling for each exit point, which can potentially induce bugs.

However, in cases where code becomes heavily nested (such as during configuration checks), readability can suffer, which in turn might induce bugs. Additionally, modularizing code to reduce nesting can impact execution speed.

Therefore, if there is a justification to prevent increased nesting or a corresponding reason, multiple exit points (returns) within a function are permitted but must be subject to review, and the results should be documented as evidence.

Example:

```
/* If configuration data 1-3 are out of range, return an error and exit processing */
if (((s_config.data1 < MIN1) || (s_config.data1 > MAX1))
 || ((s_config.data2 < MIN2) || (s_config.data2 > MAX2))
 || ((s_config.data3 < MIN3) || (s_config.data3 > MAX3))) {
    return (E_NOT_OK);
}

/* If configuration data 4-5 are out of range, return an error and exit processing */
if (((s_config.data4 < MIN4) || (s_config.data4 > MAX4))
 || ((s_config.data5 < MIN5) || (s_config.data5 > MAX5))) {
    return (E_NOT_OK);
}

/* Normal processing */
...
return (E_OK);
```

4 Standardization of Comment Writing

4.1 How to Write Comments

To improve the readability of the source code, write comments as follows:

(1) Write comments on the line preceding the source code in the same column, independent of the source code.
For long source code lines or control statements, it is better for readability to have comments on a separate line.

Example:

```
/* check DTC par event */
if ( dtc == eventdata.dtc ) {
}
```

Example:

```
/* check EventId
* NOTE: 0xFFFF is not valid
*/
if ( NO_EVENT_ID != eventId ) {
}
```

(2) Write comments on the same line as the source code, following the code.
For data assignments or local variable declarations, it is better for readability to write comments after the source code.

Example:

```
dtc = eventdata.dtc;       /* set DTC */
```

4.2 Special Cases

For cases where it is difficult to determine whether it is a coding mistake, comments are mandatory.

(1) When not writing break in a switch case statement
Provide a comment explaining why break is not needed. Generally, break should be written.

(2) When there is no processing in an if statement
Provide a comment explaining why processing is not needed.

(3) #endif

Include a comment indicating which #if it corresponds to.

4.3 Prohibition of Commenting Out Code Sections

Leaving unused code sections commented out in the source file makes the code harder to read. Therefore, do not comment out parts of the code. If it is necessary to disable a code section, use "#if 0".

[MISRA-C:2012 Dir] 4.4 (recommended)

The following is an example of prohibited code commenting:

Example:

```
/* if ( TRUE == flag ) { }*/
```

If you want to leave a code section in the source file, use "#if 0" as shown below.

Example:

```
#if 0
If ( TRUE == flag ) {
#endif
```

4.4 Comment Blocks

(1) Header Comment Block

At the beginning of the source code, include a description of the source code using the following format.

Example:

```
/***********************************************************************************/
/* Copyright     : 2024 FPT Software Corporation                                   */①
/* System Name   : AUTOSAR BSW                                                     */②
/* File Name     : Dem.c                                                           */③
/* Version       : v1.00.00                                                        */④
/* Contents      : The service component Diagnostic Event Manager (Dem) is         */⑤
/*                 responsible for processing and storing diagnostic events      */
/*                 (errors) and associated data. Further, the Dem provides       */
/*                 fault information to the Dcm (e.g. read all stored DTCs       */
/*                 from the event memory). The Dem offers interfaces to the      */
/*                 application layer and to other BSW modules.                   */
/*                                                                               */
/*                 The basic target of the Dem specification document is to      */
/*                 define the ability for a common approach of “diagnostic       */
/*                 fault memory” for automotive manufacturers and component      */
/*                 suppliers.                                                    */
/*                                                                               */
/*                 This specification defines the functionality, API and         */
/*                 the configuration of the AUTOSAR basic software module        */
/*                 Diagnostic Event Manager (Dem). Parts of the internal         */
/*                 behavior are manufacturer specific and described in the       */
/*                 Limitations chapter.                                          */
```

```
/* Author        : HaoNP2                                                    */⑥
/* Note          :                                                           */⑦
/*****************************************************************************/
```

① Copyright holder and the initial year of publication (if revised in 2025, write as "2024-2025")
② System name, product name
③ File name
④ Version of the file (Refer to "8.2 Software Version of Source Files" for details)
⑤ Description of the file
⑥ Author's name
⑦ Remarks (content is optional)

(2) Footer Comment Block

At the end of the source code, include a comment indicating the end of the source code using the following format.

Example:

```
/* EOF Dem.c ******************************************************************/
```

(3) Function Comment Block

At the beginning of a function, include a description of the function using the following format.

Example:

```
/*****************************************************************************/
/* ModuleID      : MODULE_ID_DEM (054)                                       */①
/* ServiceID     : DEM_INIT_ID (0x02)                                        */②
/* Name          : Dem_Init                                                  */③
/* Param         : (in) ConfigPtr    Pointer to the configuration set in     */④
/*                                   VARIANT-POST-BUILD                      *
/* Return        : void                                                      */⑤
/* Contents      : Initializes or reinitializes this module.                 */⑥
/* Author        : HaoNP2                                                    */⑦
/* Note          : [SWS_Dem_00181]                                           */⑧
/*****************************************************************************/
```

① Module name and Module ID
② Service name and Service ID
③ Function name
④ Function parameters (specify in/out/inout)
⑤ Return value
⑥ Description of the function
⑦ Name of the function creator
⑧ Remarks (SRS/SWS Item, etc.)

(4) Change History

Write the change history in the header comment block and function comment block using the following format.

Note: The history should be recorded when the minor and patch versions of the file are updated. When the major version of the file is updated, the file is newly created, so the history is not retained.

In the header comment block, write the date, file version, and the name of the modified function (or variable).

4.5 Traceability Compliance

To confirm whether the Software Requirements Specification (SRS) or AUTOSAR requirements (SWS) are linked to the API, a traceability matrix must be documented.

[MISRA-C:2012 Dir] 3.1 (Required)

Additionally, based on the created traceability matrix, requirement IDs need to be embedded in the source code.

(1) Function Comment Block

If requirement IDs are defined for the function itself, the requirement ID should be noted in the "Note" section of the function comment block. The requirement ID should be enclosed in brackets [] for extraction by tools.

Example:

```
/*********************************************************************************/
 :
/* Note       : [SWS_Dem_00181]   */
/*********************************************************************************/
```

(2) Code Comments

If requirement IDs are defined for specific functionalities, the requirement ID should be noted in the comments of the code implementing that functionality. The requirement ID should be enclosed in brackets [] for extraction by tools.

Example:

```
/* [SWS_Dem_00181] */
/* check EventId */
if ( NO_EVENT_ID != eventId ) {
}
```

5 Standardization of Naming Conventions

Establish naming conventions for variables, constants, etc., within the project and ensure uniformity in naming. Also, make sure that identifiers within the same namespace (e.g., "label names", "tags of structures, unions, and enumerations", "members of structures or unions", and "all other identifiers") are distinguishable from each other.

For Latin alphabet identifiers, ensure that there are no identifiers with different combinations of the following: [MISRA-C:2012 Dir] 4.5 (recommended)

Table 1: Identifiers That Are Not Distinguishable

Combination of Identifiers. That Are Not Distinguishable	**Example**	
Lowercase/Uppercase	Lowercase	returnCode
	Uppercase	ReturnCode
Underscore/No Underscore	Underscore	returnCode
	No Underscore	return_Code
Letter "O"/Number "0"	Letter "O"	ChannelO
	Number "0"	Channel0
Letter "I"/Number "1"	Letter "I"	counter_I
	Number "1"	counter_1
Letter "I"/Letter "l" (lowercase L)	Letter "I"	counter_I
	Letter "l" (lowercase L)	counter_l
Letter "l" (lowercase L)/Number "1"	Letter "l" (lowercase L)	counter_l
	Number "1"	counter_1
Letter "S"/Number "5"	Letter "S"	counter_S
	Number "5"	counter_5
Letter "Z"/Number "2"	Letter "Z"	counter_Z
	Number "2"	counter_2
Letter "n" (lowercase n)/Letter "h" (lowercase h)	Letter "n" (lowercase n)	counter_n
	Letter "h" (lowercase h)	counter_h
Letter "B"/Number "8"	Letter "B"	counter_B
	Number "8"	counter_8
Sequence of letters "rn" (r followed by n)/Letter "m"	Sequence "rn" (r followed by n)	Dcrn
	Letter "m"	Dcm

6 Standardization of Pointer Writing

6.1 Pointers

Declare pointers without adding a space after the type.

Example:

```
char*^p;
```

6.2 Null Pointer

Write null pointers as "NULL". Do not use "NULL" for anything other than null pointers.

Example:

```
char* p;p = NULL;
```

7 Standardization of Writing Dummy Functions

When implementing a program, there may be instances where you need to implement function (API) calls for modules that are not yet implemented. The following defines how to code in such cases.

7.1 Dummy Header File

To compile code that calls functions (APIs) of unimplemented modules, create a dummy header file (module_name.h) for the unimplemented module, and declare the functions (APIs) there. Do not write the declarations of functions (APIs) in the header files or source files of the module you are trying to compile.

Example:

```
void Dem_GetVersionInfo( Std_VersionInfoType* versioninfo );
Std_ReturnType NvM_RestorePRAMBlockDefaults( NvM_BlockIdType BlockId );
```

7.2 Dummy Program File

To link (build) code that calls functions (APIs) of unimplemented modules, create a dummy source file (module_name.c) for the unimplemented module, and implement the bodies of the functions (APIs) there. Do not write the bodies of functions (APIs) in the header files or source files of the module you are trying to link.

The bodies of the functions (APIs) should be empty (no processing). For functions (APIs) with return values, return a fixed normal value (such as E_OK). For functions (APIs) that return set values through return values or arguments, you can either return fixed values within the valid range or not set anything.

Example:

```
void Dem_GetVersionInfo( Std_VersionInfoType* versioninfo ) {}
Std_ReturnType NvM_RestorePRAMBlockDefaults( NvM_BlockIdType BlockId ) {
return E_OK;
}
```

8 Standardization of Software Version Update Rules

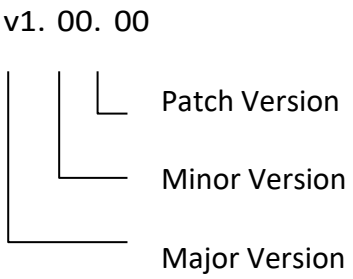
The rules for updating software versions are defined as follows.

8.1 Software Version

Define and manage the software version within the project.

8.2 Software Version of Source Files

Manage the version of each source file according to the following rules, and include it in the header comment block. (Refer to ④ in "(1) Header Comment Block" of "4.4 Comment Blocks")



- Major Version

  Update when adding new features.

- Minor Version

  Update for incompatible changes such as modifications to functions involving changes to the interface (IF). Set to "0" when the major version is updated.

- Patch Version

  Update for compatible changes such as modifications to functions not involving changes to the interface (IF). Set to "0" when the major or minor version is updated.

Note: The version of this source file should be updated for each file (this is different from the software version)

9 Functional Safety Compliance

The guidelines and rules of MISRA-C:2012 related to functional safety compliance are defined as follows.

Documentation of System Definitions

The behavior of system definitions on which program output depends must be documented and understood. Therefore, this will be described in this section.

[MISRA-C:2012 Dir] 1.1 (Required)

9.1 The behavior of dependent system definitions is defined as follows:

- C Language Standard

  Targeting C99.

9.2 Detection of Compile Errors and Warnings

Compile errors and warnings must not be detected in any source files.

Depending on the functionality being implemented, fixing warning points might cause issues that prevent the intended purpose from being achieved. In such cases, deviations should be made, and the reasons for the exceptions should be clearly stated in the release notes.

[MISRA-C:2012 Dir] 2.1 (Required)

9.3 Error Checking Implementation

9.3.1 Runtime Error Checking

Perform error checks at places where runtime errors are likely to occur.

[MISRA-C:2012 Dir] 4.1 (Required)

Below are examples of locations where error checks should be performed.

- Consider overflow

Example:

```
/* Left side: buffer size, Right side: actual data length */
/* Check if the actual data is larger than the buffer size */
if ( s_Dlt_MBuff_RemainSize < ( message_length + header_length + 1U ) ) {
    /* Error handling */
}
```

- Guard against division by zero

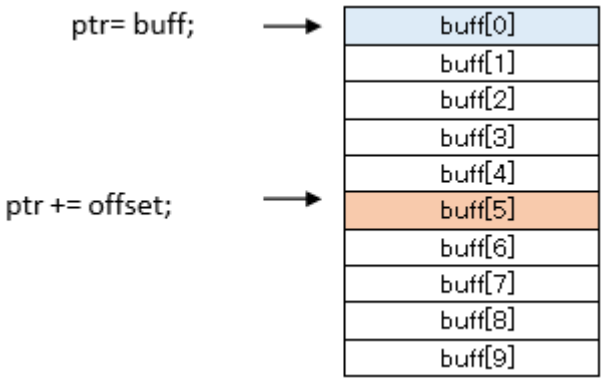
Example:

```
/* data: denominator, Division: quotient */
/* Check if the denominator is 0 */
if ( 0U == data ) {
    /* If the denominator is 0, do not perform the calculation */
    Division = 0;
} else {
    /* Perform the calculation only if the denominator is not 0 (normal processing) */
    Division = 100 / data;
}
```

- For pointer arithmetic, ensure that the results are as intended by reviewing the code to confirm that the intended operations can be performed correctly.

Example:

```
uint8 buff[10];
uint8 offset = 5;
uint8* ptr = buff;
ptr += offset;
```



In this case, during code review, it is necessary to confirm that ptr correctly points to buff[5] as intended by the operation.

- Check the parameters of functions

  For details, refer to '9.3.3 Argument Error Checking.

- Do not access NULL pointers

Example:

```
/* Check if the address is not NULL when using a pointer */
if ( NULL == PduInfoPtr ) {
    /* Error handling */
}
```

- Ensure that array indices are not negative or out of range

Below is an example of checking to ensure that array indices are not negative.

Example:

```
uint8 buff[200];

/* Check that the index is 0 or above */
for ( i = (sint8)100 ; i >= 0 ; i-- ) {
    /* Normal processing */
    buff[i] = TRUE;
}
```

Below is an example of checking to ensure that array indices are not out of range.

Example:

```
/* CANIF_Q_NUM_OF_TRCV: Number of elements in the array */
CanIf_TrcvInfoType s_TrcvInfo[CANIF_Q_NUM_OF_TRCV];

/* Check that the index is less than the number of elements */
for ( i = (uint32_least)0x0U; i < CANIF_Q_NUM_OF_TRCV; i++ ) {
    /* Normal processing */
    s_TrcvInfo[i].WakeupValidationCtrl = FALSE;
}
```

- Do not use dynamic memory allocation

  For details, refer to '9.10 Prohibition of Dynamic Memory Allocation'

9.3.2 Error Checking for Functions that Return Error Information

If a function returns error information, it is considered that an error might occur internally. Therefore, when calling a function that returns error information, proper checks of the output values should be performed.

[MISRA-C:2012 Dir] 4.7 (Required)

Below are examples of proper error checking for return values.

Example:

```
/* The return value of Func1 includes error information (= E_NOT_OK) */
Std_ReturnType Func1 ( uint8 Num );
void main( void ) {
    uint8 num = 0x01U;
    Std_ReturnType retCode = E_OK;
    retCode = Func1( num );
    /* Check if the return value of function Func1 is error information */
    if ( E_NOT_OK == retCode ) {
        /* Error handling */
    }
}
```

Example:

```
/* The return value of Func2 is a pointer */
uint8* Func2 ( uint8 Num );
void main( void ) {
    uint8 num = 0x01U;
    uint8* retPtr = NULL;
    retPtr = Func2( num );
    /* Check if the address returned by function Func2 is NULL */
    if ( NULL == retPtr ) {
        /* Error handling */
    }
}
```

Below is an example of performing appropriate checks on arguments.

Example:

```
/* The argument pointer of Func3 contains error information */
void Func3 ( uint8* Status );
void main( void ) {
    uint8 status;
    Func3( &status );
    /* Check if the argument pointer of function Func3 contains error information */
    if ( XXX_ERROR_STATUS == status ) {
        /* Error handling */
    }
}
```

Example:

```
/* When the argument pointer address of Func4 is NULL */
void Func4 ( uint8* Ptr );
void main( void ) {
    uint8* ptr = NULL;
    Func4( ptr );
    /* Check if the argument pointer address of function Func4 is NULL */
    if ( NULL == ptr ) {
        /* Error handling */
    }
}
```

9.3.3 Argument Error Checking

At the beginning of each API processing, perform a validity check of the input values (excluding internal functions).

[MISRA-C:2012 Dir] 4.14 (Required)

(1) When checking the validity of input values against a threshold

Example:

```
void Func1 ( uint8 Number) {
    /* Check if the argument exceeds the threshold */
    if ( NUM_OF_CONFIG < Number ) {
        /* Error handling */
```

```
    }
}
```

(2) When checking if the input pointer is not NULL

Example:

```
void Func2 ( uint8* Ptr) {
    /* Check if the argument is not NULL */
    if ( NULL == Ptr) {
        /* Error handling */
    }
}
```

(3) When the input value is an enumerated type

It is necessary to perform error checks in case a value that is not set in the enumeration is passed to the function.

Example:

```
typedef enum {
    BSWM_FALSE = ( 0U ),
    BSWM_TRUE,
    BSWM_UNDEFINED
} BswM_Q_RuleStateType;

void Func2 ( BswM_Q_RuleStateType Status ) {
    switch ( Status ) {
        case BSWM_FALSE:
            /* Processing */
            break;
        case BSWM_TRUE:
            /* Processing */
            break;
        case BSWM_UNDEFINED:
            /* Processing */
            break;
        default:
            /* Error handling for arguments that are not declared in the enumerated type */
            break;
    }
}
```

9.4 Prohibition of Using Assembly Language

Assembly language should not be used as it is dependent on the compiler.

If it is absolutely necessary to use assembly language, the files using assembly language should be isolated in separate files from the C language files to clearly identify the parts that need to be rewritten during porting (prohibition of inline assembly). [MISRA-C:2012 Dir] 4.3 (Required)

Additionally, when using assembly language, the compiler dependency and the reasons for using assembly language must be documented as comments in the source file. [MISRA-C:2012 Dir] 4.2 (Recommended)

9.5 Prohibition of Using Basic Types

The size of types may vary depending on the system. Therefore, basic types (such as "int") should not be used directly. Instead, use typedefs (such as "uint8") that clearly indicate the size and whether the type is signed or unsigned. [MISRA-C:2012 Dir] 4.6 (Recommended)

Below are specific examples of types on a 32-bit machine.

Table 2: Specific Examples of Types on a 32-bit Machine

typedef	Type Names	typedef	Type Names
unsigned char	uint8	unsigned long	uint8_least
unsigned short	uint16	unsigned long	uint16_least
unsigned long	uint32	unsigned long	uint32_least
unsigned long long	uint64	signed long	sint8_least
signed char	sint8	signed long	sint16_least
signed short	sint16	signed long	sint32_least
signed long	sint32	float	float32
signed long long	sint64	double	float64

Note: Align with the ECU being used. For loop variables, use the type defined as least to optimize processing speed. (Defined in SWS_Platform)

9.6 Caution When Creating Functions and Function Macros

When using functions or function-like macros, consider whether functions or function-like macros are more appropriate for the specific processing.

[MISRA-C:2012 Dir] 4.9 (Recommended)

Ensure the code complies with the following guidelines:

- If the function is called frequently, functions are more suitable than function macros.
- If speed is a priority, function macros are more suitable than functions. Specific values will be considered separately.

9.7 Caution When Using Pointers

When using pointers, it must be clear what type of pointer variables are being used, and the necessity and safety must be explainable.

Additionally, it is necessary to consider whether it is better to declare an incomplete type pointer or a complete type pointer.

[MISRA-C:2012 Dir] 4.8 (Recommended)

Below are examples of incomplete and complete type pointers.

Example:

```
struct ImpfType;  /* Incomplete type */
```

```
struct ImpfType*  ptr1; /* Pointer to an incomplete type = Incomplete type pointer */
uint8*  ptr2; /* Pointer to a complete type = Complete type pointer */
```

When an address is assigned to the above pointers, an incomplete type pointer (ptr1) cannot reference the members of the assigned address, whereas a complete type pointer (ptr2) can reference the members of the assigned address. By using an incomplete type pointer only for passing addresses, the type can be hidden from external access.

9.8 Prohibition of Multiple Includes

Multiple includes may result in undefined or incorrect behavior; therefore, multiple includes are prohibited. Accordingly, it must be ensured that multiple includes are prevented. However, multiple includes are allowed when used for memory mapping.

[MISRA-C:2012 Dir] 4.10 (Required)

Below is an example of preventing multiple includes in a header file.

Example:

```
#ifndef BSWM_H
#define BSWM_H
/* Processing */
#endif /* BSWM_H */
```

Below is an example of an exception where multiple includes are used for memory mapping.

Example:

```
#define BSWM_START_SEC_VAR_INIT_LOCAL_8
#include "BswM_MemMap.h"
static bool_t s_BswMInit[ BSWM_Q_P_NUM ] = {FALSE};
#define BSWM_STOP_SEC_VAR_INIT_LOCAL_8
#include "BswM_MemMap.h"
```

9.9 Prohibition of Standard Library Usage

The standard library shall not be used. Therefore, standard library header files shall not be included.

[MISRA-C:2012 Dir] 4.11 (Required)

[MISRA-C:2012 Rule] 22.1, 22.3 (Required)

9.10 Prohibition of Dynamic Memory Allocation

All objects shall be statically allocated, ensuring no objects are created only for a specific period. Dynamic memory allocation shall not be performed.

[MISRA-C:2012 Dir] 4.12 (Required)

The following are prohibited, and it must be confirmed during code review that no dynamic memory allocation is used:

- Use of standard library functions for dynamic memory allocation. Representative examples: calloc, malloc, realloc, and free functions.
- Custom dynamic memory allocation (including third-party packages) that does not use standard library functions.

9.11 Prohibition of File Operations

No operations shall be performed on files.
[MISRA-C:2012 Dir] 4.13 (Recommended)
[MISRA-C:2012 Rule] 22.3 (Required), 22.4, 22.6 (Mandatory))

The following operations are prohibited:

- Resource allocation (e.g., opening a file).
- Resource deallocation (e.g., closing a file).
- Other operations (e.g., reading from a file).