SEC BOOKS

Ver. 2.0

ESCR

Embedded System development Coding Reference guide

[C Language Edition]

Written and edited by
Software Reliability Enhancement Center,
Technology Headquarters,
Information-technology Promotion Agency, Japan

IPA

This document is the English edition of ESCR (Embedded System development Coding Reference) [C language edition] Version 2.0 published by IPA/SEC* in Japan. It is the revised English edition of ESCR Version 1.1 made available in 2010 in pdf format. Aimed at improving the quality of the source code written in C language, ESCR collects the important points to be noted as part of the know-how for coding and organizes them as practices and rules.

The purpose of this document is to be used as a reference guide for establishing coding conventions in organizations and groups developing embedded software using C language, and for promoting the standardization of coding styles and uniformity of source code quality. The rules in Ver. 1.1 comply with C90, which was the most widely used C language standard at the time Ver. 1.1 was published. Recently, more and more C language programmers are using C99 after it was released as the successive version of C90. To respond to this trend, MISRA C was revised significantly in March 2013 (and was issued as MISRA C: 2012). In the process of revising ESCR Ver.1.1 to Ver.2.0, the following two objectives were set to align with these recent developments in C language standard and MISRA C guidelines:

- Make the rules compliant with C99, and easy for the programmers to use the new features supported by C99;
- Make ESCR consistent with MISRA C to which it is correlated, by reviewing and updating the references to MISRA C according to its significant revisions in 2013.

To secure the continuity from the previous version, Ver. 2.0 follows the same structure as Ver. 1.1. The practices and rules carried over from the previous version are also numbered the same as in Ver. 1.1. To support the language specifications that have been extended in C99, various descriptions have been added to Ver 2.0, including the introduction of new rules, supplementary explanatory texts, and additional compliant and non-compliant coding examples. Moreover, in this document, the references to the rules in MISRA C that have been updated as a part of the significant revisions made in 2013 are numbered the same as in MISRA C: 2012, while the references to the rules in MISRA C that now exist only in the previous version are numbered the same as in MISRA C: 2004.

We sincerely hope that the effective use of ESCR Ver. 2.0 will lead to the improvement of embedded software productivity and contribute to the attainment of high-quality software development.

July 2014
Software Reliability Enhancement Center, Technology Headquarters,
Information-technology Promotion Agency, Japan

* Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, Japan

Preface

In recent years, the scale of embedded software has been expanding and development by many people is becoming mainstream. In such development projects which have large teams consisting of engineers with various levels of skills, it becomes extremely necessary to take measures to balance out the differences of their skills. In particular, measures for creating high quality source code with consideration to safety, maintainability and portability from the software implementation's standpoint independently from the engineers' skills are crucially important, and are regarded as the basic physical strength of embedded software development.

One effective means to address this issue is to prepare and utilize coding conventions where various coding expertise of the experienced are collected up as knowledge. This background led us to write and edit the "Coding Practices Guide" targeted for those establishing a coding convention for C language. In the development of this guide, the drafts were drew up by experts in programming techniques in the Implementation Quality Technical Committee of the Embedded Software Development Improvement and Promotion Task Force consists of members of the METI, IPA/SEC and related organizations.

This book is structured by practices categorized according to quality characteristics and their corresponding rules. We believe the information provided in this book will help to establish coding conventions that will meet users' requirements.

Not only will this book be useful to engineers establishing coding conventions, but it will also serve as a stimulus for new ways of looking at things with an overview of the C language characteristics for experienced developers, and for beginners, as a practical textbook which will be impart the know-how acquired by their predecessors. We hope that use of this book will lead to improve embedded software productivity and to attain high quality software development.

March 2006

Embedded Software Development Improvement and Promotion Task Force

Implementation Quality Technical Committee

Table of Contents

Part 1

How to Read the Coding Practices Guide

1 Overview

1.1 What are Coding Practices?

Creating source code (code implementation) is an inevitable task for developing embedded software. Success or failure of this task greatly affects the quality of the resulting software. C language, the most commonly used programming language for embedded software development, is said to give the programmers a relatively extensive writing flexibility. The quality of programs written in C thus tends to reflect quite clearly the difference in coding skill level between seasoned and less-experienced programmers. It is undesirable to have source code varying largely in quality, depending on the programmers' individual coding skills and experience. To prevent this risk from leading into serious quality issues, forward-thinking companies are working proactively toward standardization of their source codes by establishing coding standards or conventions to be followed organization-wide or group-wide.

Issues Regarding Coding Conventions

Coding convention is generally regarded as the organized set of "styles of (or rules in) writing code that need to be followed to maintain quality." However, it is becoming a common understanding that various issues exist in the current usage of coding conventions, including those mentioned below.

1) The necessity of rules is not understood. The appropriate methods to deal with rule violations are also not widely shared.
2) There are too many rules to learn. Yet, the existing rules are not compr ehensive enough to cover the entire scope of coding.
3) Since highly reliable tools that can thoroughly and accurately check whether the written code is complying with the relevant rules or not are unavailable, the engineers have to review the code manually through visual check, which is a heavy burden for them.

Due to such circumstances, there are, in fact, some coding conventions established at the organization or department level that have lost their significance and are no longer strictly observed.

Nevertheless, organizations that have coding conventions, no matter what kind of format they

prepare them in, are at least better than those without any. There are still quite a few that cannot reach a consensus of the coding convention to be followed internally, and are relying largely on the programmers' individual judgments to decide how the source code should be written.

What are Coding Practices?

This guide aims at solving on-site issues related to coding conventions, by providing a collection of practical coding techniques considered important from the standpoint of software quality that conform to the basic way of thinking (concept) to be followed in various coding situations. They are referred to as "coding practices" in this guide, and are presented with detailed description and specific examples of related coding conventions (or rules) for reference.

This guide is intended to enable the users to solve the above-mentioned issues in coding conventions by "establishing a concrete and effective coding convention for their own organization", using the set of relevant information provided herein as their reference.

1.2 Purpose and Position of Coding Practices and Target Users

Purpose and Position of Coding Practices

ESCR Ver. 2.0 is a guide on coding practices intended to help enable those who create and/or operate the coding conventions to establish them in their companies or projects. This document is characteristic for regarding coding conventions as "ways of writing code that should be followed by all the programmers in a given project to maintain quality" and organizing the basic rule concepts as practices. These practices are broken down into outline and details, based on the quality concept that complies with "ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models". They are respectively explained with corresponding C programming rules and the rationale for using them. Through these practices and rules, ESCR Ver. 2.0 aims at enabling the users to easily establish their own "coding convention" that meets their practical needs and also clearly explain why the included practices and rules are significant and necessary.

Target Users

This guide has been written on the assumption that it will be read by the following types of users:

Creators of the Coding Conventions

This document can be used as a reference guide to create a new coding convention or to review and reorganize existing coding conventions.

Programmers and Program Reviewers

Highly reliable and maintainable code can be produced with reasonable effort by learning and understanding the practices and rules provided in this guide.

Benefits Gained

The benefits that the users can expect to gain directly from using this guide are as mentioned above. Moreover, as a result of these benefits, the users may also be able to expect the following positive effects:

- Can remove the bottleneck in maintaining software quality caused by inconsistent performance of implementation engineers;
- Can eliminate obvious errors in the source code at an early stage, such as, during the coding phase or in subsequent reviews.

1.3 Characteristics of the Coding Practices

The coding practices in this guide have the following characteristics.

Systematically Organized Practices and Rules

This guide considers that code quality can also be classified like software quality, according to quality characteristics, such as, “reliability”, “maintainability” and “portability”, and organizes the coding practices and rules systematically based on “ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and

software quality models". The coding practices described in this guide are customs and ideas on implementation that have been developed to maintain source code quality, and they reflect the basic concepts of individual coding rules. The coding rules included in this guide have been selected based on the needs of the current conditions (actual situation with the language specifications and processing systems) after closely examining the various coding conventions existing in the world, and are presented in the form of established information that supports the corresponding practices. Classification of practices and rules according to quality characteristics makes it easy for the users of this guide to understand their respective purpose of use in terms of which aspect of quality each of them is primarily focused on maintaining.

The coding conventions referenced in this guide range from local conventions used in companies to which the writers and reviewers of this guide belong, to sets of coding rules established and used widely in different industries, including "MISRA C" and "Indian Hill C Style and Coding Standards". For details, refer to "Citations and References" at the end of this document.

Ready-to-use Reference Rules

This guide presents specific rules for C language as reference information for creating coding conventions. These rules can be used directly as coding conventions. By referring to "3. How to Use this Guide", the users of this guide can easily create their own coding conventions for C language by choosing the rules that meet their respective needs and adding any other rules that they feel are also necessary to cover the areas that are not sufficiently addressed.

Presenting the Necessity of Each Rule

The coding rules covered in this guide are respectively described with explanation of corresponding practices and examples of how the code should be written (to be compliant with the given rule) or should not be written (because that would be non-compliant), to enable the users of this guide to understand clearly why each of them is necessary. Rules considered to be already well-known among experienced programmers are so indicated in the "Preference guide" to help the users of this guide determine whether they need to include these rules in their conventions or not.

Correspondence with Other Coding Conventions

Where applicable, this guide indicates the relationship of each coding rule provided herein with corresponding coding conventions used widely in the world to make it easy for the users of this guide to check the inclusive relations, among others. Corresponding coding conventions referenced in this guide include "MISRA C" and "Indian Hill C Style and Coding Standards".

1.4 Notes on Using this Guide

Keep the following points in mind when using this guide.

Scope of the Rules

In this guide, the rules related to any of the following are considered out of scope of C language reference rules:

- Library functions
- Metrics (numbers of lines in functions/complexity of functions, etc.)
- Errors in writing possibly classified as coding errors

Although “Errors in writing possibly classified as coding errors” have been excluded from the reference rules, some frequently observed coding errors collected while preparing this guide are exemplified in “Part 3 Typical Coding Errors in Embedded Software.” This section (Part 3) is especially recommended to those who are still new to C language and prone to many of these errors. Moreover, it should also be a useful reference to project teams that find it meaningful to establish rules to prevent coding errors from being caused by such common pitfalls in programming.

Cited or Referenced Standards in this Guide

In this guide, the following standards have been cited or referenced.

C90

This is the C language standard defined in “ISO/IEC 9899:1990 Programming Language C”. It is often called C90, where “90” stands for the year ISO/IEC 9899:1990 was published. The C language standard has been revised and is now C99, making C90 an older version.

C99

This is the C language standard defined in “ISO/IEC 9899:1999 Programming Language C”. It is the current standard widely used. Since ISO/IEC 9899:1999 was published in 1999, it is often called “C99”.

C11

This is the most recent C language standard defined in “ISO/IEC 9899:2011 Programming Language C” and thereby is the current C language standard. Since ISO/IEC 9899:2011 was published in 2011, it is often called “C11”.

C++

This is the C++ language standard defined in “ISO/IEC 14882:2003 Programming language C++”.

MISRA C

It refers collectively to the coding guidelines for C language defined by The Motor Industry Software Reliability Association (MISRA) in UK, which include MISRA C:1998, MISRA C:2004 and MISRA C:2012.

MISRA C:1998

The convention in Citations and References [5].

MISRA C:2004

The convention in Citations and References [6]. This is the revised version of MISRA C:1998.

MISRA C:2012

The convention in Citations and References [7]. This is the revised version of MISRA C:2004.

Naming Conventions for Variables and Functions

For the names of variables and functions used in code examples, this guide uses notations that are as simple as possible to prevent the users from becoming confused or misunderstanding the rules unnecessarily.

Difference from Ver. 1.1

This document (Ver. 2.0) has been updated from the previous version (Ver. 1.1) primarily by adding rules considered to be necessary when using the new features defined in C99, and by modifying some rules and descriptions considered necessary to do so in order to maintain consistency with the revisions made in MISRA C:2012. In addition, there are some rules included in Ver. 1.1 that have been deleted in this document based on the judgment that they no longer have to be considered as rules. There are also descriptions that deal with the same matters explained in Ver. 1.1, but have been rewritten in this document with more clarity or in a way that would be easier to understand.

The rule number of each deleted rule is treated as missing number and kept vacant. The rules that deal with the same topic in Ver. 1.1 are numbered the same in this document so that the users of old ESCR edition (Ver 1.1) can find it easy to track the rules.

The rules that have been newly added to this document are R1.3.4, R3.1.3, R3.1.4, R3.6.3 and M4.7.7. The rules in Ver. 1.1 that have been deleted in the document are M3.1.3 and M4.7.4.

2 Understanding Source Code Quality

2.1 Quality Characteristics

For many, speaking of software quality would remind them of “bugs.” However, in the field of software engineering, the quality of software as a product is grasped in a broader perspective. This concept of software product quality is defined in detail and organized systematically in ISO/IEC 25010.

ISO/IEC 25010 and Source Code Quality

ISO/IEC 25010 defines the quality of software product by breaking it down into eight characteristics (quality characteristics): “reliability”, “maintainability”, “portability”, “efficiency”, “security”, “functionality”, “usability” and “compatibility”

Among them, “functionality”, “usability” and “compatibility” are considered to be the three quality characteristics that should be addressed at an early stage, preferably before moving on to the design phases in the upstream process. Whereas, “reliability”, “maintainability”, “portability”, and “efficiency” are considered to be the quality characteristics that have close relevance with the development of high-quality source code and should therefore be examined in depth during the coding phase. “Security”, which has been defined as the quality subcharacteristic of “functionality” in the previous standard, ISO/IEC 9126-1, is considered basically as a quality characteristic that is relevant in the design phase, but coding such as for avoiding stack overflow can also affect security. For more information on coding practices related to security, please refer to “CERT C Secure Coding Standard”.

Based on the above broad categorization, this guide has adopted the latter four quality characteristics - “reliability”, “maintainability”, “portability”, and “efficiency” - as the main focus, and gathered the coding practices that are primarily concerned with any of these four. Table 1 shows the relationship between the “quality characteristics” defined in ISO/IEC 25010 and the “code quality” proposed in this guide, along with the “quality subcharacteristics”.

Table 1 Quality Characteristics of Software and Code Quality

Quality Characteristics (ISO/IEC 25010)		Quality Subcharacteristics (ISO/IEC 25010)		Code Quality
Reliability	Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time	Maturity	Degree to which a system meets needs for reliability under normal operation	Low occurrence of bugs through continued use
		Availability	Degree to which a system, product or component is operational and accessible when required for use	
		Fault Tolerance	Degree to which a system, product or component operates as intended despite the presence of hardware or software faults	Tolerance for bugs and interface violations, etc
		Recoverability	Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system	

Quality Characteristics (ISO/IEC 25010)		Quality Subcharacteristics (ISO/IEC 25010)		Code Quality
Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers	Modularity	Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components	Degree to which the components are composed such that a change to one component of the code has minimal impact on other components.
		Reusability	Degree to which an asset can be used in more than one system, or in building other assets	Degree to which a code can be used in other programs
		Analysability	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified	Easiness of understanding the code
		Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality	Easiness of modifying the code, and lowness of impact from modifications
		Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met	Easiness of testing and debugging the modified code

Quality Characteristics (ISO/IEC 25010)		Quality Subcharacteristics (ISO/IEC 25010)		Code Quality
Portability	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another	Adaptability	Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments	Easiness of adapting to different environments *Including conformance to standards
		Installability	Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment	
		Replaceability	Degree to which a product can be replaced by another specified software product for the same purpose in the same environment	
Performance Efficiency	Performance relative to the amount of resources used under stated conditions	Time Behaviour	Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements	Efficiency with regard to processing time
		Resource Utilization	Degree to which the amounts and types of resources used by a product or system when performing its functions meet requirements	Efficiency with regard to resources
		Capacity	Degree to which the maximum limits of a product or system parameter meet requirements	

Quality Characteristics (ISO/IEC 25010)		Quality Subcharacteristics (ISO/IEC 25010)		Code Quality
Security	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization	Confidentiality	Degree to which a product or system ensures that data are accessible only to those authorized to have access	Degree of certainty that data are accessible only to those authorized to have access
		Integrity	Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data	Degree of prevention of unauthorized access to, or modification of, computer programs or data
		Non-repudiation	Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later	
		Accountability	Degree to which the actions of an entity can be traced uniquely to the entity	
		Authenticity	Degree to which the identity of a subject or resource can be proved to be the one claimed	

2.2 Quality Characteristics, Coding Practices and Rules

Overall Structure

In this guide, the basic matters to be followed when creating source code are organized as "practices". For each "practice", this guide introduces "rules" that are more specific reference information to keep in mind at the time of coding.

These "practices" and "rules" provided in this guide are classified and arranged in order, according to their association to any of the four quality characteristics described earlier in 2.1. The following section defines what "practice" and "rule" actually mean in this guide (see also Figure 1):

Practice

A "practice" is a custom or a set of ideas on implementation to maintain source code quality. Each practice reflects the basic concept of individual coding rule. These practices are broken down into outline and details.

Rule

A "rule" is a specific agreement that must be followed and constitutes a part of coding convention. This guide presents these rules as reference information. In this guide, a "rule" is also sometimes used as a collective term that represents a group of relevant rules.

Correspondence of Practices and Rules

Most practices and rules are related to multiple quality characteristics, but in this guide, they are respectively discussed in the section of the characteristic to which they are most strongly related. Associating each practice with a particular quality characteristic makes it possible and easy for the users of this guide to understand how each practice strongly affects which aspect of quality.

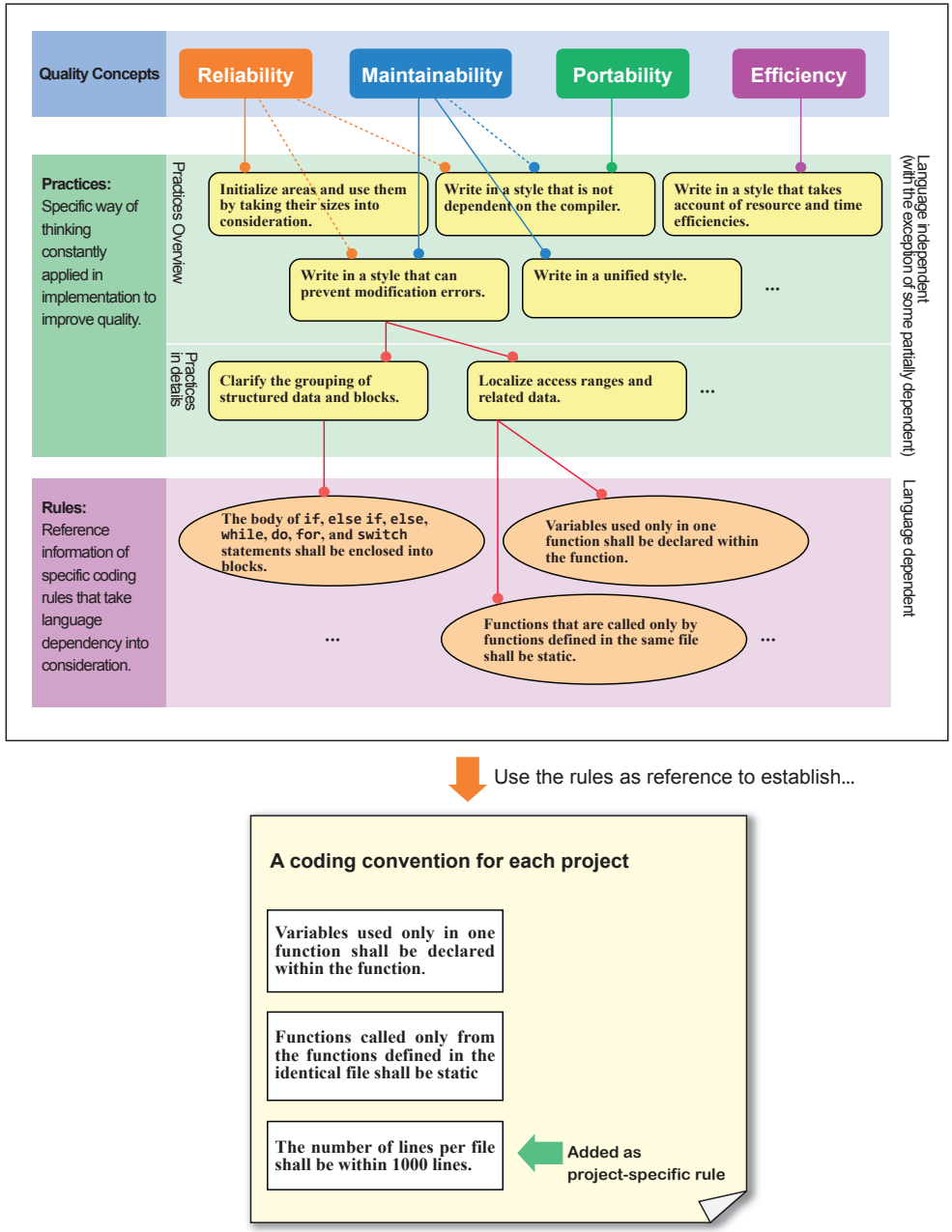


Figure1. Relationship Between Quality Concepts, Practices, and Rules

3 How to Use this Guide

3.1 Scenarios for Using this Guide

Usage Scenarios

This guide intends to support the creation of coding conventions, and assumes that it will be used in the following three scenarios:

1) Creating a new coding convention;
2) Enhancing existing coding conventions;
3) Serving as a learning material for programmers' training and self-study.

Creating a New Coding Convention

Organizations or departments that have not been able to organize any coding conventions to be followed internally can use this guide for reference to establish their own coding convention that suits their respective needs.

Enhancing Existing Coding Conventions

Even in organizations and departments that have already established their coding conventions, it is effective to maintain them regularly. Using this guide as a reference will help them review the contents of their existing coding conventions more efficiently.

Serving as a Learning Material for Programmers' Training and Self-Study

There are many books published on C language. Unlike those existing ones, this guide focuses on implementation quality, and provides an organized set of information on how to create source code that can maintain and improve its quality. In this sense, this guide can also be an excellent material for the users to learn about source code quality from a more practical point of view.

3.2 Creating a New Coding Convention

This section presents the procedure for creating a new coding convention by using this guide. It is intended for projects that do not have any coding conventions of their own.

When to Create

Create the coding convention before proceeding to the program design stage. While a coding convention is a group of rules that are referred to during coding, some rules, such as, the naming convention applied to function names are associated with program design, and therefore need to be decided before starting the program design.

How to Create

Projects creating a new coding convention of their own are recommended to follow the procedure described below, step by step:

Step-1 Decide on the policy for creating a coding convention.
Step-2 Choose the rules based on the creation policy that has been decided.
Step-3 Define the project-dependent parts of the rules.
Step-4 Determine the procedure for setting exceptions to the rules if necessary.

After following these steps in order, add any other rules as needed.

Step-1 Decide on the policy for creating a coding convention

In creating a coding convention, the first thing to do is to decide on its policy. A creation policy defines how the code should be written for the project, based on, such as, the characteristics of the software developed in the project and the members of the project. For example, should the priority be placed on safety and write code that avoids using features that are not safe, even if they are convenient to use? Or, should the code be written in a way that makes careful use of such unsafe but convenient features? These are some of the questions that need to be addressed in the creation policy. When deciding on the creation policy, each project should consider which quality characteristics are particularly important for its software development, and examine what kind of coding practices it should adopt from the following perspectives:

- Coding that takes account of fail-safe;
- Coding that improves the program readability;
- Coding that makes debugging easy, etc.

Step-2 Choose the rules based on the creation policy that has been decided

The next step is to choose the suitable rules from the Practices Chart in Part 2, based on the creation policy decided in Step-1. If the project decides on the policy that prioritizes on portability, for example, efforts should be made to include many rules that address the portability issues in its coding convention.

In "Part 2 - Coding Practices for Embedded Software" of this guide, some rules are marked with either "○" or "●" as a guide to facilitate the selection process. A rule marked with "○" indicates that it is regarded so important for the particular quality characteristic it addresses, that if this rule is not adopted as a part of the coding convention, that quality aspect may be seriously impaired. Whereas, "●" indicates that it is a rule that is already so well-known among those who are very knowledgeable about C language specifications that it may not necessarily be included in the coding convention. The simplest way of creating a coding convention would therefore be to choose only the rules indicated with "○", which would result in a set of widely applied rules.

Step-3 Define the project-dependent parts of the rules

In this guide, the rules are treated as one of the following three types:

1) Rules that can be used as a part of the coding convention without making any changes (In the "Rule specification" field, these rules are not marked.)
2) Rules that need to be chosen from several alternatives, according to the project characteristics (In the "Rule specification" field, these rules are marked as "Choose".)
3) Rules that need to be prescribed more specifically in a document (In the "Rule specification" field, these rules are marked either as "Define" or "Document".)

The rules treated either as type 2) or type 3) cannot be included in the coding convention as they are. For rules treated as type 2) to be adopted as a part of the newly created coding convention, they must be first chosen from the multiple alternatives presented in this guide. To adopt the rules treated as type 3) as a part of the coding convention, they must be more fine-tuned so that they can address the specific needs of each project. In doing so, the supplementary explanation provided to each practice described in this guide should serve as a useful reference on rule definition.

Step-4 Determine the procedure for setting exceptions to the rules

The quality characteristics that should be focused at the time of coding may differ, depending on the feature the project is intending to realize through implementation. (For example, "In this project, efficiency should be prioritized over maintainability..."). There may be cases when writing code that is fully compliant with a certain rule included in the coding convention causes difficulty in achieving the project-specific objective. To deal with such cases, it is necessary to have a procedure to allow

partial exceptions to this rule

WThe important points to be covered in this procedure are as follows:

- Describe what problems may occur by writing code that is compliant with the rule;
- Have experts review the problems and possible solutions;
- Record the review result.

Be sure not to allow exceptions too easily. The substance of the rule will be lost when there are too many exceptions.

The following is an example of the procedure for allowing exceptions.

[Example procedure]

(1) Prepare a form describing the reason for the exception.
(This form should, for example, contain the following items.) - Rule number;
- Location of the code at issue (file name, line number);
- Problem(s) caused by complying with the rule;
- Impact of deviation from the rule.

(2) Have experts review the problems and possible solutions. → Enter the review result in the form.

(3) Gain approval from the head person (manager, project leader, etc) responsible of the coding process. → Record the approval in the form.

3.3 Enhancing Existing Coding Conventions

For projects where coding conventions already exist, this guide can be a useful reference to review and enhance the contents of their coding conventions.

Preventing Oversights and Omissions

By sorting the rules in existing coding conventions based on the concept of practices described in this guide, the project members will be able to identify and supplement the elements that have been overlooked or omitted, and see in a fresh light which tasks they have been placing importance on in their project.

Clarifying the Necessity for Rules

For those who have been feeling compelled to follow some rules without knowing why, this guide will serve as a useful tool to understand clearly why they are necessary by referring to the practices and compliant examples showing how they should be used.

3.4 Serving as a Learning Material for Programmers' Training and Self-Study

This guide is a good learning material for programmers who have studied C language but are still not used to or have little experience in practical coding.

Target Users

This guide is targeted at the following group of programmers:

- Programmers who have studied and acquired the basic skills in C language
- Programmers who have experience in other programming languages but are beginners in C language

What The Users Can Learn

By reading this guide, which is organized from the standpoint of quality characteristics like reliability, maintainability and portability, the users can learn:

- How to write code that can improve reliability;
- How to write code that can prevent bugs from being produced;
- How to write code that can facilitate debugging and testing;
- How to write code that is easy to read, and the reasons why good readability is necessary.

Part 2

Coding Practices for Embedded Software: Practices Chart

- How to Read the Practices Chart
- Terminology Used in the Practices Chart
- Coding Practices for Embedded Software
  - Reliability
  - Maintainability
  - Portability
  - Efficiency

How to Read the Practices Chart

Organizational Structure of the Practices

Coding practices shown in Part 2 are classified according to four software quality characteristics (reliability, maintainability, portability, efficiency).

Practices in Outline

Practices closely related to each characteristic are largely divided into “practices in outline”. For example, the practices closely related to maintainability are largely divided into five practices in outline from “Maintainability M1: Keep in mind that others will read the program” to “Maintainability M5: Write in a style that makes testing easy”.

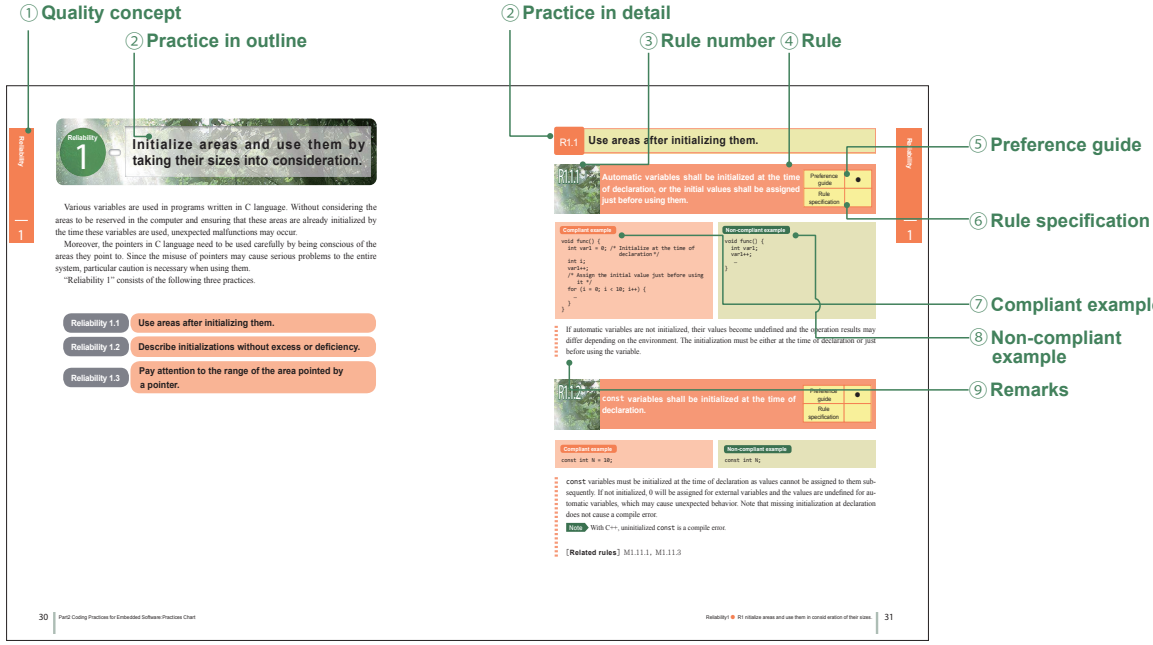
Practices in Detail

Each practice in outline is broken down into more specific subsets called “practices in detail”. For example, the practice in outline “Maintainability M3: Write programs simply” has four practices in detail, which are:

Maintainability 3.1	**Do structured programming.**
Maintainability 3.2	**Limit the number of side effects per statement to one.**
Maintainability 3.3	**Write expressions that differ in purpose separately.**
Maintainability 3.4	**Do not use complicated pointer operation.**

Layout of the Practices Chart

For each practice, reference information on rules to be noted during actual coding is provided in a chart form. The following diagram shows the layout of a sample chart, which is followed by the description of each field composing the chart:



① Quality concept

Quality concepts are related to the main quality characteristics of "ISO/IEC 25010". This guide uses the following four quality concepts:

Reliability **Maintainability** **Portability** **Efficiency**

② Practice

Describes the practice to be followed by programmers during coding:

- In outline — Defines the general concept of the practice. It is not dependent on programming languages.

- In detail — Elaborates the general concept of the practice with more specific points that should be noted. Like practices in outline, it is basically programming language-independent, but some are stated as C language-specific.

③ Rule number

Identification number of each rule

④ Rules

Specific reference rule or rules for C language corresponding to the practice that must be followed. The rules cited from MISRA C are written in the following format.

Examples: **[MISRA C:2004 1.3], [MISRA C:2012 R8.14]**

⑤ Preference guide

Provides supportive information (marks) to indicate whether the corresponding rule described under eachpractice should be chosen as a part of the newly created coding convention or not.

No mark: Rules considered to be appropriate to choose, based on the project characteristics.

● Rules considered unnecessary to be included in the coding convention, when seen from the eyes of those who are very knowledgeable about the language specification (i.e.: rules that are already too common and obvious to experienced programmers).

○ Rules considered to significantly impair the quality characteristics if they are not followed.

⑥ Rule specification

Provides supportive information (verbal indicators) to indicate which rule need to be defined more specifically in detail or not, depending on the project policy, or should be prescribed in a document, such as, when it is recommended to "record the behavior and usage of compiler-dependent language specification as a document" (the latter is referred to as the "documentation rule" which can be used as it is, but is strongly recommended to be documented in more detail for various reasons).

No mark: Rules that do not need to be defined further in depth or prescribed in a documen with more specific details

Choose: Rules required to be chosen from a list of multiple alternatives. Each alternative is numbered, using a parenthesized numeral (e.g.: (1), (2), ...).

Define: Specific rules that need to be defined for each project. The part to be defined is enclosed by《 》

Document: Rules that need to be prescribed in a document. The part to be documented is enclosed by《 》

⑦ Compliant example

Example of source code written in compliance with the rule.

⑧ Non-compliant example

Example of source code violating the rule.

⑨ Remarks

Provides notes pertaining to C language specification, and explanation on why the particular rule is necessary and what kind of problem(s) may be caused by violation of that rule, among others.

Terminology Used in the Practices Chart

The meaning of the terms used in the chart is as respectively explained in the table below:

Term	Meaning
Access	Reference to variables or the reference with modification.
Type specifier	Specifies a data type. There are two type specifiers, one that specifies basic types such as `char`, `int` and float and the other that specifies types defined with typedefs by the programmer for their own.
Type qualifier	Adds specific attributes to types. There are three type qualifiers: `const`, `restrict` and `volatile`.
Storage class specifier	Specifies the location where data are stored. There are four specifiers; `auto`, `register`, `static`, and `extern`.
Boundary alignment	Indicates methods for the compiler to allocate data into memory. For example, if `int` type is 2 bytes, be sure to allocate such data from an even address of the memory and not to allocate from an odd address.
Trigraph sequence	Defined seguences of three characters such as '`??=`', '`??/`', '`??(`'for the compiler to replace with single character. '`??=`', '`??/`', '`??(`' are interpreted into '#', '\' , '[' respectively.
Lifetime	Duration that the references to a variable from the program is guaranteed after it is generated.
Multibyte character	A character expressed by data of two or more bytes. Chinese characters, Japanese characters, and Unicode characters are included.
Null pointer	A pointer that is not equivalent to any pointers that point to data or functions.
Null character	A character that express the end of a string. Expressed with '`\0`'.
Scope	The part of the program within which an identifier can be used to indicate, such as, the variable it defines. File scope refers to the scope up to the end of the file.
Side effect	Processing that cause changes to a state of execution environment. The following processings appliy: reference and change to volatile data, change to data, change to files, and function-calls that perform these operations.
Block	A range that is enclosed with braces '{', '}' in data declarations and programs etc.
Enumeration type	enum type. Constructed with several enumerated members.
Enumerator	Members of an enumerated type (`enum` type).

Coding Practices for Embedded Software

This part presents coding practices for embedded software. As explained carlier, The coding practices are categorized according to the perspective of four characteristics (quality concepts): “reliability”, “maintainability”, “portability” and “efficiency”, which have been adopted from the software quality characteristics defined in ISO/IEC 25010. Please note, however, that these practices have been categorized in this way basically for the sake of convenience of the users of this guide, and that there are actually some useful practices and corresponding rules that can be applied to improve more than one characteristic (e.g.: both reliability and maintainability).

Moreover, the coding practices respectively related to these quality characteristics and the reference rules that support the correct ways of executing these practices are also described in this part of the guide.

Reliability	**R**	Practices to improve the reliability of software that has been developed fall under this category. Main points taken into consideration include: Main points considered include: - Minimizing problems arising while using the software; - Increasing tolerability against bugs and interface violation.
Maintainability	**M**	Practices to create source code that is easy to modify and maintain fall under this category. Main points taken into consideration include: - Making the code easy to understand and modify; - Minimizing the impact of modifications on the entire code; - Making the modified code easy to check.
Portability	**P**	Practices to port the software program that has been created on the assumption of being used to operate under a certain environment to another environment as efficiently as possible without error fall under this category.
Efficiency	**E**	Practices to effectively utilize the performance and resources of the software that has been developed fall under this category. Main points taken into consideration include: - Coding that is processing time-conscious; - Coding that takes account of memory size.

Reliability

A large number of embedded software is incorporated into products and used to support our daily lives in various situations. Consequently, the level of reliability demanded to quite a number of embedded software is extremely high. Software reliability requires the software to be capable of not behaving wrongly (not causing failure), not affecting the functionality of the entire software and system in case of malfunction, and promptly restoring its normal behavior after a malfunction occurs.

At the source code level, the point to be noted in regard to software reliability is the need of contriving methods to avoid coding that may cause such malfunctions as much as possible.

- **Reliability 1:** Initialize areas and use them by taking their sizes into consideration.
- **Reliability 2:** Use data by taking their ranges, sizes and internal representations into consideration.
- **Reliability 3:** Write in a way that ensures intended behavior.

Various variables are used in programs written in C language. Without considering the areas to be reserved in the computer and ensuring that these areas are already initialized by the time these variables are used, unexpected malfunctions may occur.

Moreover, the pointers in C language need to be used carefully by being conscious of the areas they point to. Since the misuse of pointers may cause serious problems to the entire system, particular caution is necessary when using them.

"Reliability 1" consists of the following three practices.

Reliability 1.1	Use areas after initializing them.
Reliability 1.2	Describe initializations without excess or deficiency.
Reliability 1.3	Pay attention to the range of the area pointed by a pointer.

R1.1 Use areas after initializing them.

Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.

Preference guide	●
Rule specification	

Compliant example

```
void func() {
  int var1 = 0; /* Initialize at the time of
                   declaration */
  int i;
  var1++;
  /* Assign the initial value just before using
     it */
  for (i = 0; i < 10; i++) {
    …
  }
}
```

Non-compliant example

```
void func() {
  int var1;
  var1++;
   …
}
```

If automatic variables are not initialized, their values become undefined and the operation results may differ depending on the environment. The initialization must be either at the time of declaration or just before using the variable.

`const` variables shall be initialized at the time of declaration.

Preference guide	●
Rule specification	

Compliant example

```
const int N = 10;
```

Non-compliant example

```
const int N;
```

`const` variables must be initialized at the time of declaration as values cannot be assigned to them subsequently. If not initialized, 0 will be assigned for external variables and the values are undefined for automatic variables, which may cause unexpected behavior. Note that missing initialization at declaration does not cause a compile error.

Note With C++, uninitialized `const` is a compile error.

［**Related rules**］ M1.11.1, M1.11.3

R1.2 Describe initializations without excess or deficiency.

Arrays with specified number of elements shall be initialized with values that match the number of the elements.

Preference guide	●
Rule specification	

Compliant example

```
char var[] = "abc";
       - or -
char var[4] = "abc";
```

Non-compliant example

```
char var[3] = "abc";
```

Initializing an array with a string will not cause an error at declaration even if a space for a null character is not ensured in the array size. This is not a problem if described intentionally. However, when the array is used as an argument for a string handling function etc., the absence of a null character indicating the end of the string is more likely to cause unexpected behavior. When initializing a string, it is necessary to ensure a space for the null character at the end.

［**Related rule**］ M2.1.1

Initialization of enumeration type (enum type) members shall be by either: not specifying any constants; specifying all the constants; or specifying only the first member.

Preference guide	
Rule specification	

Compliant example

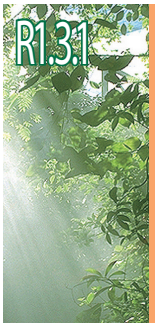
```
/* A different value is assigned respectively
   from E1 to E4 */
enum etag { E1=9, E2, E3, E4 };
enum etag var1;
var1 = E3;
/* E3 and E4 in var1 will never be equal */
if (var1 == E4)
```

Non-compliant example

```
/* Both E3 and E4 become 11 unintentionally */
enum etag { E1, E2=10, E3, E4=11 };
enum etag  var1;
var1 = E3;
/* It will be true despite the intention
   because E3 and E4 are equal */
if (var1 == E4)
```

If an initial value is not specified to a member of an enumeration type, the value of the immediately preceding member plus 1 (the value of the first member is 0) will be specified to this member. If some initial values are specified while others are not, the same value may unintentionally be assigned to different members and may become the cause of unexpected behavior. To prevent the same value from being assigned to different members, initialization of the members must be by either not specifying any constants, specifying all the constants, or specifying only the first member, depending on the usage.

R1.3 Pay attention to the range of the area pointed by a pointer.



(1) Integer addition to or subtraction from (including ++ and --) pointers shall not be made; Array format with [] shall be used for references and assignments to the allocated area.

(2) Integer addition to or subtraction from (including ++ and --) pointers shall be made only when the pointer points to the array and the result must be pointing within the range of the array.

Preference guide	●
Rule specification	Choose

Compliant example

```
#define N 10
int data[N];
int *p;
int i;
p = data;
i = 1;
Compliant example of (1) and (2)
data[i] = 10;   /* conpliant */
data[i+3] = 20; /* conpliant */
Compliant example of (2)
*(p + 1) = 10;
```

Non-compliant example

```
#define N 10
int data[N];
int *p;
p = data;

Non-compliant example of (1)
*(p + 1) = 10;   /* Non-compliant */
p += 2;   /* Non-compliant */
Non-compliant example of (2)
*(p + 20) = 10;  /* Non-compliant */
```

Performing operations on pointers can blur the destinations pointed by the pointers. It raises the possibility of implanting bugs that is likely to refer or write to unsecured areas. Rather, using an array name that points to the beginning of the area and to access elements of the array with indices will make the program safer. A dynamic memory area obtained by `malloc` should be treated as an array, and a pointer to the starting address of the area should be handled as the array name.

For multi-dimensional array, this rule applies to each partial array.

Regarding rule (2), it is permissible to point to the area directly after the last element of the array as long as the array element is not accessed. In other words, in the case where `int data[N]` and `p=data`, `p+N` complies with the rule as long as it is not used for accessing the array elements, whereas, using, such as, `*(p+N)` that accesses an array element is non-compliant.

Subtraction between pointers shall only be applied to pointers that address elements of the same array. 【MISRA C:2012 R18.2】

Preference guide	●
Rule specification	

Compliant example

```
ptrdiff_t off; /* ptrdiff_t is a type of result of
                  subtraction between pointers defined in
                  <stddef.h> */
int var1[10];
int *p1, *p2;
p1 = &var1[5];
p2 = &var1[2];
off = p1 - p2; /* Compliant */
```

Non-compliant example

```
ptrdiff_t off; /* ptrdiff_t is a type of result of
                  subtraction between pointers defined in
                  <stddef.h> */
int var1[10], var2[10];
int *p1, *p2;
p1 = &var1[5];
p2 = &var2[2];
off = p1 - p2; /* Non-compliant */
```

In C language, subtraction between pointers expresses how many elements exist between the two elements pointed by each pointer. In this case, if each pointer points to a different array, the way the variables are laid out between them is implementation-dependent and the execution result is not guaranteed. This implies that subtraction between pointers is meaningful only when both pointers are pointing to elements in the same array. Therefore, before subtracting one pointer from another pointer, the programmer must ensure that both pointers are addressing elements of the same array.

［**Related rule**］R1.3.3

Comparing which pointer is greater or less than the other pointer shall be used only when two pointers are both pointing at either the elements in the same array or the members of the same structure.

Preference guide	●
Rule specification	

Compliant example

```
#define N 10
char var1[N];
void func(int i, int j) {
  if (&var1[i] < &var1[j]) {
    …
```

Non-compliant example

```
#define N 10
char var1[N];
char var2[N];
void func(int i, int j) {
  if (&var1[i] < &var2[j]) {
    …
```

Comparing addresses of different variables does not cause a compile error, but is meaningless because the address of the variable is implementation-dependent. In addition, the behavior of such a comparison is not defined (undefined behavior).

［**Related rules**］R1.3.2, R2.7.3

The `restrict` type qualifier shall not be used. 【MISRA C:2012 R8.14】

Preference guide	
Rule specification	

Compliant example

Non-compliant example

```
void f(int n, int * restrict p,
int * restrict q) {
  while (n-- > 0) {
    *p++ = *q++;
  }
}

void g(void) {
  extern int d[100];

  f(50, d+1, d);  /* Undefined behavior */
```

By using `restrict` type qualifier, efficient code can be generated by a compiler and the accuracy of static analysis by using such as the code checker will improve. However, the use of `restrict` type qualifier will require the programmer to guarantee that the targeted areas will not overlap, and there is a risk involved because the compiler will not output an error.

The data used in programs vary in how they are represented internally and in the range they can be operated, depending on their types. When using these different types of data for operation, they must be written carefully by paying attention to various aspects, including precision and size. Otherwise, unexpected malfunctions may occur when they are processed in, such as, arithmetic operations. Therefore, there is a need to handle data with care, by taking their ranges, sizes and internal representations, among others, into consideration.

Reliability 2.1	Make comparisons that do not depend on internal representations.
Reliability 2.2	When values such as logical values are defined as a range, do not make a judgment by finding whether or not a value is equivalent to any value (representative value that is implemented) within this range
Reliability 2.3	Use the same data type to perform operations or comparisons.
Reliability 2.4	Describe code by taking operation precision into consideration.
Reliability 2.5	Do not use operations that have the risk of information loss.
Reliability 2.6	Use types that can represent the target data.
Reliability 2.7	Pay attention to pointer types.
Reliability 2.8	Write in a way that will enable the compiler to check that there are no conflicting declarations, usages and definitions.

R2.1 Make comparisons that do not depend on internal representations.

Floating-point expressions shall not be used to perform equality or inequality comparisons.

Preference guide	●
Rule specification	

Compliant example

```
#define LIMIT 1.0e-4
void func(double d1, double d2) {
  double diff = d1 - d2;
  if (-LIMIT <= diff && diff <= LIMIT) {
   …
```

Non-compliant example

```
void func(double d1, double d2) {
  if (d1 == d2) {
    …
```

In case of a floating-point type, values written in the source code do not exactly match with those actually implemented. Therefore, the comparison results must be judged by taking account of tolerance.

［**Related rule**］ R2.1.2

Floating-point variable shall not be used as a loop counter.

Preference guide	●
Rule specification	

Compliant example

```
void func() {
  int i;
  for (i = 0; i < 10; i++) {
      …
```

Non-compliant example

```
void func() {
  double d;
  for (d = 0.0; d < 1.0; d += 0.1) {
      …
```

If operations are repeatedly performed to a floating-point variable used as a loop counter, the intended result may not be achieved due to accumulated calculation errors. Therefore, integer type (`int` type) should be used for loop counters.

［**Related rule**］ R2.1.1

memcmp shall not be used to compare structures and unions.

Preference guide	●
Rule specification	

Compliant example

```
struct TAG {
  char c;
  long l;
};
struct TAG var1, var2;
void func() {
  if (var1.c == var2.c && var1.l == var2.l) {
    …
```

Non-compliant example

```
struct TAG {
  char c;
  long l;
};
struct TAG var1, var2;
void func() {
  if (memcmp(&var1, &var2, sizeof(var1)) == 0) {
    …
```

Memories for structures and unions may contain unused areas. Since the values in the areas are unknown, `memcmp` should not be used. When making comparisons of structures or unions, they should be made between the corresponding members.

［**Related rule**］M1.6.2

R2.2 When values such as logical values are defined as a range, do not make a judgment by finding whether or not a value is equivalent to any particular value (representative value) within this range

Comparison with a value defined as true shall not be made in expressions that examine true or false.

Preference guide	
Rule specification	

Compliant example

```
#define FALSE 0
/* func1 may return a value other than 0 and 1
*/
void func2() {
  if (func1() != FALSE) {
 - or -
  if (func1()) {
```

Non-compliant example

```
#define TRUE 1
/* func1 may return a value other than 0 and 1
 */
void func2() {
  if  (func1() == TRUE) {
```

In C language, true is represented by any non-zero value, not necessarily 1.

［**Related rule**］M1.5.2

R2.3 Use the same data type to perform operations or comparisons.

`Unsigned` integer constant expressions shall be described within the range that can be represented with the result type.

Preference guide	
Rule specification	

Compliant example

```
#define MAX 0xffffUL  /* Specify long type*/
unsigned int i = MAX;
if (i < MAX + 1)
/* If long is 32 bits, there is no problem even
   when the number of bits of int is not 32.*/
```

Non-compliant example

```
#define MAX 0xffffU
unsigned int i = MAX;
if (i < MAX + 1)
/* The result varies depending on whether the
   intis16bitsor32bits.Ifintis16 bits, the
   operation result will wrap around and the
   comparison result will be false. If int is
   32 bits, the operation result will be within
   the range of int and the comparison result
   will be true. */
```

`Unsigned` integer operations in C language wrap around without overflow (the result will be the remainder of the maximum representable value). Because the overflow is not flagged, there is a risk of not noticing when the operation result differs from the intended result. For example, when there are two environments that differ in the number of bits of `int`, the same constant expression produces different operation results, depending on whether they exceed the representable value range or not.

When using conditional operator (?: operator), the logical expression shall be enclosed in parentheses ( ) and both return values shall be the same type.

Preference guide	
Rule specification	

Compliant example

```
void func(int i1, int i2, long l1) {
  i1 = (i1 > 10) ? i2 : (int)l1;
```

Non-compliant example

```
void func(int i1, int i2, long l1) {
  i1 = (i1 > 10) ? i2 : l1;
```

When writing code using different types, perform a cast to specify which type is expected as the result.

［**Related rule**］ M1.4.1

Loop counters and variables used for comparison of loop iteration conditions shall be the same type.

Preference guide	●
Rule specification	

Compliant example

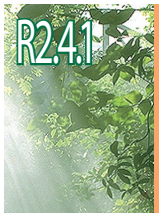
```
void func(int arg) {
  int i;
  for (i = 0; i < arg; i++) {
```

Non-compliant example

```
void func(int arg) {
  unsigned char i;
  for (i = 0; i < arg; i++) {
```

Using comparison between variables with different ranges of representable values as a loop iteration condition may produce unintended results and end up in an infinite loop.

R2.4 Describe code by taking operation precision into consideration.



When the type of an operation and the type of the destination to which the operation result is assigned (assignment destination) are different, the operation shall be performed after casting them to the type of expected operation precision.

Preference guide	●
Rule specification	

Compliant example

```
int i1, i2;
long l;
double d;
void func() {
  d = (double)i1 / (double)i2; /* floating-point
                                  devision */
  l = ((long)i1) << i2; /* Shift using long */
```

Non-compliant example

```
int i1, i2;
long l;
double d;
void func() {
  d = i1 / i2; /* integer division */
  l = i1 << i2; /* Shift using int */
```

The type used in operation is determined by the type of the expression (operand) used for the operation, and the type of the assignment destination is not taken into consideration at compile time. Therefore, do not expect the operation to output its result in the type of the assignment destination if the operating type differs from the destination type. When there is a need to execute an operation in the type that differs from the operand type, perform a cast to convert the type of operand to the intended type before operation.

［**Related rule**］ R2.5.1

R2.4.2 When performing arithmetic operations or comparisons of expressions mixed with `signed` and `unsigned`, an explicit cast to the expected type shall be performed.

Preference guide	
Rule specification	

Compliant example

```
long l;
unsigned int ui;
void func() {
  l = l / (long)ui;
       - or -
  l = (unsigned int)l / ui;

  if (l < (long)ui) {
   - or -
  if ((unsigned int)l < ui) {
```

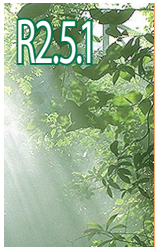
Non-compliant example

```
long l;
unsigned int ui;
void func() {
  l = l / ui;
  if (l < ui) {
     …
```

Some operations, such as, size comparison, multiplication and division output different results, depending on whether they are performed with `signed` or `unsigned`. If an operation is written for a mixture of signedness, `unsigned` operation is not always executed because it is the compiler that determines which type to execute the operation in (whether with `signed` or `unsigned`) by taking account of the respective data sizes. Therefore, when performing an arithmetic operation of mixed signedness, there is a need to check whether the intended operation is with `signed` or `unsigned`, and perform an explicit cast to change the operating type to the desired type before operation so that the intended operation result can be expected.

Note: If there are data types that may have to be changed for use in intended operation, it is often better to change them rather than performing a cast mechanically. Therefore, in such a situation, first consider changing the data type.

R2.5 Do not use operations that have the risk of information loss.



When performing assignments (=operation, actual arguments passing of function calls, function return) or operations to data types that may cause information loss, they shall be first confirmed that there are no problems, and a cast shall be described to explicitly state that they are problem-free.

Preference guide	○
Rule specification	

Compliant example

```
/* Assignment examples */
short s;      /* 16 bits */
long l;       /* 32 bits */
void func() {
  s = (short)l;
  s = (short)(s + 1);
}
/* Operation examples */
unsigned int var1, var2;  /* int size is 16
                             bits */
  var1 = 0x8000;
  var2 = 0x8000;
  if ((long)var1 + var2 > 0xffff) {  /* The
                          result is true */
```

Non-compliant example

```
/* Assignment examples */
short s;      /* 16 bits */
long l;       /* 32 bits */
void func() {
  s = l;
  s = s + 1;
}
/* Operation examples */
unsigned int var1, var2;  /* int size is 16
                             bits */
var1 = 0x8000;
var2 = 0x8000;
if (var1 + var2 > 0xffff) {  /* The result is
                                    false  */
```

When a value is assigned to a variable that differs in type, the value may change (i.e. information may be lost). The assignment destination, therefore, should be the same type whenever possible. When a value is assigned to a different type intentionally in cases, such as, where there is no risk of information loss or no impact even if information is lost, perform a cast to explicitly state the intention.

When performing an operation that outputs a result that exceeds the representable value range of the type used, the result may become an unintended value. Therefore, for safety, carry out the operation after verifying that the operation result is within the representable value range of the type used, or after converting it to the type that could adequately accommodate larger values.

Note: In many cases, it is better to change data types used rather than casting mechanically. Changing data types should be considered first.

［**Related rule**］ R2.4.1

R2.5.2 Unary operator '-' shall not be used in unsigned expressions.

Preference guide	●
Rule specification	

Compliant example

```
int i;
void func() {
  i = -i;
```

Non-compliant example

```
unsigned int ui;
void func() {
  ui = -ui;
```

If a unary operator ‘ - ’ is used in `unsigned` expression and the operation result falls out of representable value range of the original `unsigned` type, unintended behavior may occur.
For example, writing “`if ( - ui < 0 )` ” in the non-compliant example will not make this “ `if` ” true.

R2.5.3 When ones’ complement (~) or left shift (<<) is applied to `unsigned char` or `unsigned short` type data, an explicit cast to the type of the operation result shall be performed.

Preference guide	○
Rule specification	

Compliant example

```
uc = 0x0f;
if((unsigned char)(~uc) >= 0x0f)
```

Non-compliant example

```
uc = 0x0f;
if((~uc) >= 0x0f)  /* It is not true */
```

The result of operation using `unsigned char` or `unsigned short` type will be `signed int` type. When the sign bit turns on due to operation, the intended result may not be achieved. This is why casting to the type of the intended operation is necessary. The above non-compliant example shows that `~uc` always becomes false as it produces a negative value.

［**Related rule**］ R2.5.4

R2.5.4 The right-hand side of a shift operator shall be zero or more, and less than the bit width of the left-hand side.

Preference guide	●
Rule specification	

Compliant example

```
unsigned char   a;  /* 8 bits */
unsigned short  b;  /* 16 bits */
b = (unsigned short)a << 12; /* Clearly
     indicated that the operation is 16 bits */
```

Non-compliant example

```
unsigned char   a;  /* 8 bits */
unsigned short  b;  /* 16 bits  */
b = a << 12; /* There may be an error in the
                                shift count */
```

The behavior of a shift operator whose right-hand side (shift count) specifies a negative value or a value equal to or larger than the bit width* at the left-hand side (value to be shifted) is not defined in C language standard and will vary depending on the compiler used. (* This bit width will be that of `int` type if the size is smaller than `int` .)
The intention of specifying a value up to the bit width of `int` type as the shift count will be unclear if the left-hand side (value to be shifted) is of a type that is smaller in size, even though its behavior is defined in the language standard.

［**Related rule**］ R2.5.3

R2.6 Use types that can represent the target data.

R2.6.1

(1) The types used for bit field shall only be `signed int` or `unsigned int`. If a bit field of 1 bit width is required, `unsigned int` type shall be used, and not the `signed int` type.

Preference guide	
Rule specification	Choose

(2) The types used for bit field shall be `signed int`, `unsigned int` or `_Bool`. If a bit field of 1 bit width is required, `unsigned int` type or `_Bool` type shall be used.

(3) The types used for bit field shall be `signed int`, `unsigned int`, `_Bool`, or those allowed by the compiler that are either `enum` or the type that specifies `signed` or `unsigned`. If a bit field of 1 bit width is required, the type that specifies `unsigned` or `_Bool` type shall be used.

Compliant example

```
Compliant example of (1)
struct S {
  signed int   m1:2;
  unsigned int m2:1;
  unsigned int m3:4;
};

Compliant example of (2)
struct S {
  _Bool        m1:1;
};

Compliant example of (3)
struct S {
unsigned char m1:2; /* If char is defined by the
                       compiler as allowable:
                       conpliant */
enum E        m2:2; /* If enum is defined by the
                       compiler as allowable:
                       conpliant */

};
```

Non-compliant example

```
Non-compliant example of (1)
struct S {
 int          m1:2; /* Without sign specification
                       is also non-compliant in
                       (2) and (3) */
signed int    m2:1; /* Use of signed int type of
                       1 bit field is also
                     non-compliant in (2) and (3) */
unsigned char m3:4; /* Use of char type is also
                       non-compliant in (2)
                       char char is conpliant in
                       (3) if it is defined by the
                       compiler as allowable */
enum    E     m4:2; /* Use of enum type is also
                       non-compliant in (2)  enum
                       is conpliant in (3) if it
                       is defined by the compiler
                       as allowable */
_Bool         m5:1; /* Use of _Bool type is
                      conpliant in (2) and (3) */
};

Non-compliant example of (2)
struct S {
int           m1:2; /* Without sign specification
                       is also non-compliant in
                       (1) and (3) */
signed int    m2:1; /* Use of signed int type of
                       1 bit field is also
                     non-compliant in (1) and (3) */
unsigned char m3:4; /* Use of char type is also
                       non-compliant in (1)
                       char is conpliant in (3)
                       if it is defined by the
                       compiler as allowable */
enum    E     m4:2; /* Use of enum type is also
                       non-compliant in (1)
                       enum is conpliant in (3)
                       if it is defined by the
                       compiler as allowable */
};

Non-compliant example of (3)
struct S {
int           m1:2; /* Without sign specification
                       is also non-compliant in
                       (1) and (2) */
signed int    m2:1; /* Use of signed int type of
                       1 bit field is also
                     non-compliant in (1) and (2) */
};
```

(1) To be compatible with C90, use only the `int` type defined in C90, which is the "`int` type that has specified the signedness to either `signed` or `unsigned`". Do not use the "signedness-unspecified `int` type" that may become `signed` or `unsigned`, depending on the compiler used.

(2) Use only the "`int` type that specifies the signedness as `signed`" or `unsigned` defined in C99, or the `_Bool` type, and do not use the "`int` type that does not specify the signedness" because the interpretation of signedness or unsignedness may vary depending on the compiler used.

(3) In addition to the types defined in C99 language specification, the types defined in processing systems can also be used. However, do not the integer type that does not specify the signedness, because the interpretation of signedness and unsignedness may vary depending on the compiler. Moreover, for the bitfield of 1-bit integer type, specify `unsigned` because the values that can be expressed by 1-bit `signed` integer type are only -1 and 0.

Data used as bit sequences shall be defined with `unsigned` type, and not with the `signed` type.

Preference guide	
Rule secification	

Compliant example

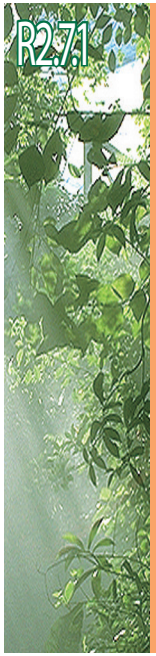
```
unsigned int flags;
void set_x_on() {
  flags |= 0x01;
```

Non-compliant example

```
signed int flags;
void set_x_on() {
  flags |= 0x01;
```

The result of bitwise operation (~, << , >> , & , ^ , |) to `signed` type may vary, depending on the compiler used.

R2.7 Pay attention to pointer types.



(1) Pointer type shall not be converted to other pointer type or integer type, and vice versa, with the exception of mutual conversion between “pointer to data” type and “pointer to `void*`” type.

Preference guide	○
Rule specification	Choose

(2) Pointer type shall not be converted to other pointer type or integer type with less data width than that of the pointer type, with the exception of mutual conversion between “pointer to data” type and “pointer to `void*`” type.

(3) Pointer type shall not be converted to other pointer type or integer type with less data width than that of the pointer type, with the exception of mutual conversion between “pointer to data” type and “pointer to other data” type, and between “pointer to data” type and “pointer to `void*`” type.

Compliant example

```
int  *ip;
int  (*fp)(void) ;
char *cp;
int  i;
void *vp;

Compliant example of (1)
ip = (int*)vp;

Compliant example of (2)
i = (int)ip;

Compliant example of (3)
i = (int)fp;
cp = (char*)ip;
```

Non-compliant example

```
int   *ip;
int   (*fp)(void) ;
char   c;
char  *cp;

Non-compliant example of (1)
ip = (int*)cp;

Non-compliant example of (2)
c =(char) ip;

Non-compliant example of (2)
ip =(int*) fp;
```

If a pointer type variable is casted or assigned to another pointer type, it is difficult to identify what kind of data is contained in the area the pointer points to. With some MPUs, runtime errors occur if the destination of a pointer that is not at word boundaries is accessed as `int` type; thus changing pointer types may cause a risk of unexpected bugs. It is safer not to cast or assign pointer type variables to other pointer types. Converting pointer types to integral types is also risky, involving the same problem stated above. Such conversions, therefore, should be reviewed with experts, whenever deemed necessary. Moreover, attention must also be given to the value ranges of `int` type and pointer type. Be sure to check the specifications of the compiler beforehand, because there may be cases where the size of the pointer type is 64 bits even though the size of `int` type is 32 bits.

`<stdint.h>` defines `intptr_t` and `uintptr_t`, which respectively represents `signed` and `unsigned` integral types with data width capable of holding a value converted from a pointer type and be converted back to that type with a value that equals to the original pointer. These types should be used when converting between pointer type and integrer type.

Pointer conversion rules

As explained in the rule under R.2.7.1, it is a risk to convert (assign) pointer type variable to other pointer type more than is necessary, because the intended result may not gained. The rules on pointer conversion have been organized in the form of a table, as shown below.

In the following table, the origin to convert from are listed in the rows, and the destination to convert to are listed in the columns. ○ indicates that the conversion can be made, and × indicates that the conversion should not be performed.

Details (1)

		Conerted to			
		Pointer to data type	Pointer to function type	Pointer to void type	Other type
Converted from	Pointer to data type	×	×	○	×
	Pointer to function type	×	×	×	×
	Pointer to void type	○	×	—	×
	Other type	×	×	×	

Details (2)

			Conerted to				
			Pointer to data type	Pointer to function type	Pointer to void type	Integrer type	
						With less data width than that of the pointer type	With more data width that than of the pointer type
Converted from	Pointer to data type		×	×	○	×	○
	Pointer to function type		×	×	×	×	○
	Pointer to void type		○	×	—	×	○
	Integrer type	With less data width than that of the pointer type	×	×	×		
		With more data width that than of the pointer type	○	○	○		

Details (3)

			Converted to				
			Pointer to data type	Pointer to function type	Pointer to void type	With less data width than that of the pointer type	With more data width that than of the pointer type
Converted from	Pointer to data type		○	×	○	×	○
	Pointer to function type		×	×	×	×	○
	Pointer to void type		○	×	—	×	○
	Integrer type	With less data width than that of the pointer type	×	×	×		
		With more data width that than of the pointer type	○	○	○		

R2.7.2 A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.【MISRA C:2012 R11.8】

Preference guide	○
Rule specification	

Compliant example

```
void func(const char *);
const char *str;
void x() {
  func(str);
   …
}
```

Non-compliant example

```
void func(char *);
const char *str;
void x() {
  func((char*)str);
   …
}
```

Be careful when accessing the areas qualified by `const` or `volatile`, because they are only for reference and must not be optimized. If a cast that removes any `const` or `volatile` qualification from the type addressed by a pointer is performed, the compiler will not be able to check and detect error descriptions in the program even if there are any, or may perform an unintended optimization.

R2.7.3 Comparison to check whether a pointer is negative or not shall not be performed.

Preference guide	●
Rule specification	

Compliant example

–

Non-compliant example

```
int * func1() {
  …
  return -1;
}
int func2() {
  …
if (func1() < 0) { /* Comparison intended to
                      check whether negative or
                      not */
    …
  }
  return 0;
}
```

It is meaningless to compare whether a pointer is larger or smaller than 0.
When the subject of comparison is a pointer, the compiler will convert 0 into a null pointer. Therefore, even when the comparison of pointer against 0 is intended, the comparison will actually be between two pointers, and the intended behavior may not be achieved.

［**Related rule**］R1.3.3

R2.8 Write in a way that will enable the compiler to check that there are no conflicting declarations, usages and definitions.

Functions with no parameters shall be declared with a `void` type parameter.

Preference guide	○
Rule specification	

Compliant example

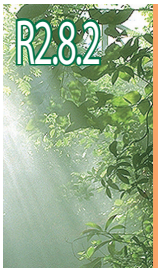
```
int func(void) ;
```

Non-compliant example

```
int func();
```

The declaration `int func()` does not mean that a function has no parameters. It is an old-styled (K&R style) declaration that means that a function has unknown number and types of parameter. Therefore, when declaring functions with no parameters, write `void` explicitly.

［**Related rule**］R2.8.3



(1) Functions shall not be defined with a variable number of arguments.【MISRA C:2004 16.1】
(2) When using functions with a variable number of arguments,《they shall be used after documenting the intended behaviors based on the compiler used.》

Preference guide	
Rule specification	Choose Document

Compliant example

```
Compliant example of (1)
int func(int a, char b);
```

Non-compliant example

```
Non-compliant example of (1)
int func(int a, char b, ... );
```

Without understanding the behavior of functions with a variable number of arguments in the processing system, their use may cause stack overflow or other unexpected results.
In addition, when the number of arguments is variable, the number and the types of the arguments are not explicitly specified, and it will lower the readability of the code.
MISRA C:2012 prohibits the use of functions defined in `<stdarg.h>`.

［**Related rule**］R2.8.3

One prototype declaration shall be made at one place from where it can be referenced by both the function calls and function definition.

Preference guide	○
Rule specification	

Compliant example

```
-- file1.h --
void f(int i);

-- file1.c --
#include "file1.h"
void f(int i) { … }

-- file2.c --
#include "file1.h"
void g(void) { f(10); }
```

Non-compliant example

```
-- file1.c --
void f(int i);     /* Declared in each file  */
void f(int i) { … }

-- file2.c --
void f(int i);     /* Declared in each file */
void g(void) { f(10); }
```

This rule is for preventing the prototype declaration and function definition from becoming inconsistent.

［**Related rules**］R2.8.1，R2.8.2

Reliability 3 Write in a way that ensures intended behavior.

It is essential to be thorough with describing how to handle all the potential errors, by also taking account of unexpected events that may occur in cases that are even conceived as highly unlikely from the standpoint of program specifications. Moreover, writing code in ways that do not rely on language specifications, such as, explicit indication of operator precedence can also improve safety. To achieve high reliability, it is desirable to make every effort to avoid coding that leads to malfunction and write in a way that ensures intended behavior and safety as much as possible.

Reliability 3.1	Write in a way that is conscious of area size.
Reliability 3.2	Prevent operations that may cause runtime error from falling into error cases.
Reliability 3.3	Check the interface restrictions when a function is called.
Reliability 3.4	Do not perform recursive calls.
Reliability 3.5	Pay attention to branch conditions and describe how to handle cases that do not follow the predefined conditions when they occur.
Reliability 3.6	Pay attention to the order of evaluation.

R3.1 Write in a way that is conscious of area size.

(1) In an extern declaration of an array, the number of elements shall always be specified.
(2) In an extern declaration of an array, the number of elements shall always be specified, except for extern declarations of arrays that correspond to the array definition that includes initialization and has omitted the number of elements.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
extern char *mes[3];
…
char *mes[] = {"abc", "def", NULL};

Compliant example of (2)
extern char *mes[];
…
char *mes[] = {"abc", "def", NULL};

Compliant example of (1) and (2)
extern int var1[MAX];
…
int var1[MAX];
```

Non-compliant example

```
Non-compliant example of (1)
extern char *mes[];
…
char *mes[] = {"abc", "def", NULL};

Non-compliant example of (1) and (2)
extern int var1[];
…
int var1[MAX];
```

Making an extern declaration without specifying the size of an array will not cause an error. However, if the size is not specified, it may cause problems in checking outside the array range. Therefore, it is better to explicitly specify the array size in its declaration. However, there are cases where it is better not to specify the array size in the declaration, such as, when the size of the array is determined by the number of initial values and is not fixed in the declaration.

［**Related rule**］ R3.1.2

R3.1.2 Iteration conditions for a loop to sequentially access array elements shall include the judgment on whether the access is within the range of the array or not.

Preference guide	
Rule specification	

Compliant example

```
char var1[MAX];
for (i = 0; i < MAX && var1[i] != 0; i++) {
   /* Even if 0s are not set in the var1 array,
   there is no risk of accessing outside the
   array range */
    …
```

Non-compliant example

```
char var1[MAX];
for (i = 0; var1[i] != 0; i++) {  /* If 0s are
   not set in the var1 array, there is a risk
   of accessing outside the array range */
    …
```

This rule is to prevent accessing outside the range.

［**Related rule**］ R3.1.1

R3.1.3 The size of the array initialized with a designated initializer shall be clearly indicated.

Preference guide	
Rule specification	

Compliant example

```
int a[ 5 ] = { [ 0 ] = 1 };
```

Non-compliant example

```
int b[ ] = { [ 0 ] = 1 };
```

Unless the size of the array is clearly indicated when defining the array, the largest index among the elements that will be initialized will be determined as the size. When a designated iinitializer is used, there are times when it is not clear which index is the largest and should be initialized.

［**Related rule**］ R3.1.1

Variable length array type shall not be used. 【MISRA C:2012 R18.8】

Preference guide	
Rule specification	

Compliant example

```
#define MAX   1024
void func(void)  {
 int  a[MAX]; /* Compliant Secured an array of
                 largest length */
```

Non-compliant example

```
void func(int  n) {
 int  a[n];       /* Non-compliant Variable
                     length array */
```

The use of variable length array type has the following problems:

• Risk of stack overflow

Variable length array can be assigned to a stack area. Therefore, if the variable length array size is big, there is a risk of stack overflow.

• Behavior that is not defined in C language standard

The behavior when the variable length array size is not a positive value is not defined in C language standard.

• Misconceived array size

```
int y = 10;
typedef   int   INTARRAY[y];
y = 20;
INTARRAY  z; /* Array size of z is 10, and not 20. */
```

R3.2 Prevent operations that may cause runtime error from falling into error cases.

Operations shall be performed after confirming that the right-hand side expression of division or remainder operation is not 0.

Preference guide	
Rule specification	

Compliant example

```
if (y != 0)
    ans = x/y;
```

Non-compliant example

```
ans = x/y;
```

Apart from when the value is obviously not 0, the operations should be performed after checking that the right-hand side expression of division or remainder is not 0. Otherwise, division by zero error may occur at runtime.

［**Related rules**］R3.2.2, R3.3.1

Destination pointed by a pointer shall be referenced to after checking that the pointer is not the null pointer.

Preference guide	
Rule specification	

Compliant example

```
if (p != NULL)
   *p = 1;
```

Non-compliant example

```
*p = 1;
```

［**Related rules**］R3.2.1, R3.3.1

R3.3 Check the interface restrictions when a function is called.

If a function returns error information, then that error information shall be tested.【MISRA C:2012 D4.7】

Preference guide	
Rule specification	

Compliant example

```
p = malloc(BUFFERSIZE);
if (p == NULL)
  /* Error handling */
else
  *p = '\0';
```

Non-compliant example

```
p = malloc(BUFFERSIZE);
*p = '\0';
```

When a function returns a value, the code that does not use that return value may cause an error. If it is not necessary to reference the return value, consider setting a project-specific rule to clearly indicate the unnecessity of referencing, such as, by casting to `void`.

［**Related rules**］ R3.2.1, R3.2.2, R3.5.1 R3.5.2

R3.3.2 The function shall check if there are constraints on parameters before starting to process.

Preference guide	
Rule specification	

Compliant example

```
int func(int para) {
  if (!((MIN <= para) && (para <= MAX)))
    return range_error;
  /* Normal processing */
   ...
}
```

Non-compliant example

```
int func(int para) {
  /* Normal processing */
   ...
}
```

Whether the constraints on parameters are checked by the function that calls or the function that is called depends on how the interface is designed. However, in order to prevent checking from being overlooked, the same check should be performed in one place. Therefore, the check should be performed by the function that is called.
In case the function that is called cannot be changed, such as, when it is in a library, create a wrapper function.

Example of wrapper function:

```
int func_with_check(int arg) {
  /* If arg is violating the parameter constraints, return range_error */
  /* If not, call func and return the result */
}
/* Use a wrapper function to make the function call */
if (func_with_check(para) == range_error) {
  /* Error processing */
}
```

C99 allows the specification of the lower limit of array size by using a static qualifier in an array declaration of a formal parameter. For example, in the following function declaration, the lower limit of elements of an argument of array a is specified as 3.

```
void func(int a[static 3]);
```

By specifying such constraints on parameters, tools like the compiler would quite likely check the restrictions applied to arguments.

R3.4 Do not perform recursive calls.

Functions shall not call themselves, either directly or indirectly.【MISRA C:2012 R17.2】

Preference guide	
Rule specification	

Compliant example

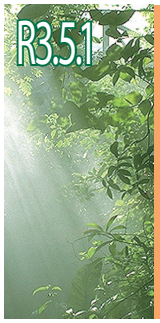
–

Non-compliant example

```
unsigned int calc(unsigned int n)
{
  if (n <= 1) {
    return 1;
  }
   return n * calc(n-1);
}
```

Since the stack size used at runtime for recursive calls cannot be predicted, there is a risk of stack overflow.

R3.5 Pay attention to branch conditions and describe how to handle cases that do not follow the predefined conditions when they occur.



The `else` clause shall be written at the end of an `if-else if` statement. If it is known that the `else` condition does not normally occur, the description of the `else` clause shall be either one of the following:

《(i) Unexpected condition handling process shall be written in the `else` clause.

(ii) A comment specified by the project shall be written in the `else` clause.》

Preference guide	○
Rule specification	Define

Compliant example

```
/* else clause of an if-else if statement where
the else condition does not normally occur */
if (var1 == 0) {
    …
} else if (0 < var1) {
    …
} else {
    /* Write an exception handling process */
    …
}
…
if (var1 == 0) {
    …
} else if (0 < var1) {
    …
} else {
    /* NOT REACHED */
}
```

Non-compliant example

```
/* if-else if statement without the else clause
*/
if (var1 == 0) {
    …
} else if (0 < var1) {
    …
}
```

If there is no `else` clause in an `if-else if` statement, it is not clear whether the programmer has forgotten to write the `else` clause or deliberately left out the else clause be cause the `else` condition dose not occur. Even if it is known that the `else` condition does not normally occur, the behavior of the program when an unexpected condition occurs can be specified by writing the `else` clause as follows:

(i) Write the behavior under unexpected conditions in the `else` condition (How program behaves in case of occurrence of the `else` condition should be determined.)

Or, the program is much easier to understand by just writing a comment that the `else` condition does not occur.

(ii) Write a project-specific comment like `/* NOT REACHED */` clearly indicating that the `else` condition does not occur to express that the `else` clause has not been forgotten.

［**Related rules**］ R3.3.1, R3.5.2

Preference guide	○
Rule specification	Define

《The `default` clause shall be written at the end of a `switch` statement. If it is known that the `default` condition does not normally occur, the description of the `default` clause shall be either one of the following:

《(i) An exception handling process shall be written in the `default` clause.

(ii) A comment specified by the project shall be written in the `default` clause.》

Compliant example

```
/* Default clause in a switch statement where
   the default condition does not normally occur
*/
 switch(var1) {
 case 0:
     …
     break;
 case 1:
     …
     break;
 default:
     /* Write an exception handling process */
     …
     break;
 }
 …
 switch(var1) {
 case 0:
     …
     break;
 case 1:
     …
     break;
 default:
     /* NOT REACHED */
     break;
 }
```

Non-compliant example

```
/* Switch statement without the default clause
*/
switch(var1) {
case 0:
     …
     break;
case 1:
     …
     break;
}
```

If there is no `default` clause in a `switch` statement, it is not clear whether the programmer has forgotten to write the `default` clause or deliberately left out the `default` clause because the `default` condition does not occur. Even if it is known that the `default` condition does not normally occur, the behavior of the program when an unexpected condition occurs can be specified by writing the `default` clause as follows:

(i) Write the behavior under unexpected conditions in the `default` condition (Predefine the behavior of theprogram if by any chance the `default` condition occurs).

Or, the program is much easier to understand by just writing a comment that the `default` condition does not occur.

(ii) Write a project-specific comment like `/* NOT REACHED */` that clearly indicates that the `default` condition does not occur to express that the `default` clause was not written because it was forgotten. Such comment will improve the readability of the program.

［**Related rules**］ R3.3.1, R3.5.1, M3.1.4

Equality operators (==, !=) shall not be used for comparisons of loop counters.

Preference guide	
Rule specification	

Compliant example

```
void func() {
  int i;
  for (i = 0; i < 9; i += 2) {
    …
```

Non-compliant example

```
void func() {
  int i;
  for (i = 0; i != 9; i += 2) {
    …
```

If the amount of change for the loop counter is not 1, an infinite loop may occur. Therefore, for comparison to determine the number of loop iterations, do not use the equality operators (==, !=). (Instead use <=, >=, <, >.)

R3.6 Pay attention to the order of evaluation

Variables whose values are changed in an expression shall not be referred to or modified in the same expression.

Preference guide	●
Rule specification	

Compliant example

```
f (x, x);
x++;
- or -
f (x + 1, x);
x++;
```

Non-compliant example

```
f (x, x++);
```

Compilers do not guarantee the execution (evaluation) order of each actual argument in functions with multiple parameters. The arguments may be executed from the right or from the left. In addition, compilers do not guarantee the execution order of the left-hand and the right-hand side of binary operations such as + operation. Therefore, if the same object is updated and referenced in a sequence of arguments or binary operation expressions, the execution result is not guaranteed. Such a problem, where the execution result is not guaranteed, is called a side effect problem. Do not write code that causes such side effect problems.

This rule, however, does not prohibit descriptions, such as, those shown below which do not cause the side effect problem.

```
x = x + 1;
x = f(x);
```

［**Related rules**］ R3.6.2, M1.8.1

Function calls with side effects and `volatile` variables shall not be described more than once in a sequence of actual arguments or binary operation expressions.

Preference guide	○
Rule specification	

Compliant example

```
1.
extern int G_a;
x = func1();
x += func2();
…
int func1(void) {
     G_a += 10;
    …
}
int func2(void) {
     G_a -= 10;
    …
}

2.
volatile int v;
y = v;
f(y, v);
```

Non-compliant example

```
 1.
 extern int G_a;
x = func1() + func2();  /* With side effect
                           problem */
 …
 int func1(void) {
      G_a += 10;
     …
 }
 int func2(void) {
      G_a -= 10;
     …
 }

 2.
 volatile int v;
 f(v, v);
```

Compilers do not guarantee the execution (evaluation) order of each actual argument for functions with multiple parameters. The arguments may be executed from the right or flom the left. In addition, compilers do not guarantee the execution order of the left-hand and the right-hand side of binary operations such as + operation. Therefore, the execution results of two or more function calls with side effects and `volatile` variables in a sequence of arguments or binary operation expressions may not be guaranteed. Such unsafe descriptions must be avoided.

［**Related rules**］ R3.6.1, M1.8.1

`sizeof` operator shall not be used in expressions that have side effect.

Preference guide	●
Rule specification	

Compliant example

```
x = sizeof(i);
i++;
y = sizeof(int[i]);
i++;
```

Non-compliant example

```
x = sizeof(i++);
y = sizeof(int[i++]);
```

Until C90, the expression in parenthesis of `sizeof` operator was used only for finding the size of the expression type, and was not executed.
Therefore, even when `++` operator like `sizeof`(i++) was described, `i` was not incremented. However, in C99, if the type is a variable length array, there are cases when the expression is evaluated. In those cases, `i` in `sizeof`(`int`[i++]) will be incremented by `++` operator. Such description should not be used because it can easily be misunderstood.

Maintainability

Many embedded software developments require maintenance tasks, including the modification of the software that has already been developed.

There are various reasons for maintenance. For example, maintenance becomes necessary:

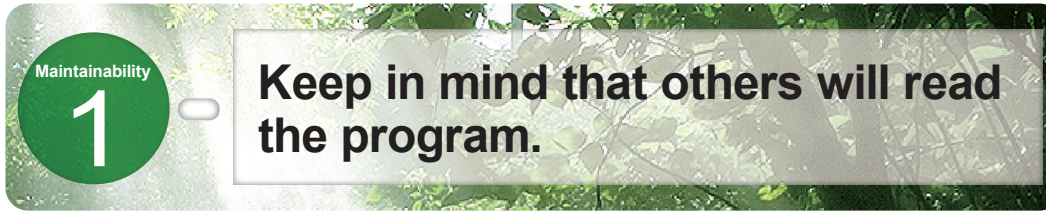
- When a bug is found in one part of the released software and must be modified;
- When a new function is added to existing software in re-

When any kind of additional work is carried out on the already developed software as in the above examples, it is important to perform such work as accurately and efficiently as possible to maintain the quality of the software.

This is called “maintainability” in the field of system development.

This section clarifies the practices to keep and improve the maintainability of embedded software source code.

- Maintainability 1: Keep in mind that others will read the program.
- Maintainability 2: Write in a style that can prevent modification errors.
- Maintainability 3: Write programs simply.
- Maintainability 4: Write in a unified style.
- Maintainability 5: Write in a style that makes testing easy.



It is easily conceivable that source code is reused and maintained by engineers who are not the original creators. Therefore, it is necessary to write source code that is easy to understand by taking account of others who will read it later.

Maintaiability 1.1	Do not leave unused descriptions.
Maintaiability 1.2	Do not writing confusingly.
Maintaiability 1.3	Do not write in an unconventional style.
Maintaiability 1.4	Write in a style that clearly specifies the order of evaluation of operations.
Maintaiability 1.5	Explicitly describe the operations that are likely to cause misunderstanding when they are omitted.
Maintaiability 1.6	Use one area for one purpose.
Maintaiability 1.7	Do not reuse names.
Maintaiability 1.8	Do not use language specifications that are likely to cause misunderstanding.
Maintaiability 1.9	When writing in an unconventional style, explicitly state its intention.
Maintaiability 1.10	Do not embed magic numbers.
Maintaiability 1.11	Explicitly state the area attributes.
Maintaiability 1.12	Correctly describe the statements even if they are not compiled.

M1.1 Do not leave unused descriptions.

M1.1.1 Unused functions, variables, parameters, typedefs, tags, labels or macros shall not be declared (defined).

Preference guide	○
Rule specification	

Compliant example

```
void func(void) {

When necessary in call back function
int cbfunc1(int arg1, int arg2);
int cbfunc2(int arg1, int);
 /* In case the call back function types are
    fixed to be int(*)(int,int), the second
    argument is necessary even when it is not
    used. */
```

Non-compliant example

```
void func(int arg) {
       /* arg is unused */
```

Declarations (definitions) of unused functions, variables, parameters labels, etc. impairs maintainability because it makes it difficult to determine whether the programmer has forgotten to delete them or has made a description error.
However, when writing a call back function, make it explicit that parameter is not used, by not describing the name of the parameter to keep the function types consistent.

［**Related rules**］M1.9.1, M4.7.2

M1.1.2 For commenting out sections of code,《the coding rule shall be specified.》【MISRA C:2012 D4.4】

Preference guide	○
Rule specification	

Compliant example

```
…
#if  0   /* Ineffective due to ~ */
    a++;
#endif
…
```

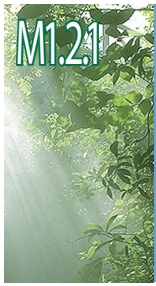
Non-compliant example

```
…
/* a++; */
…
```

Normally, invalidated information should not be left in the code as it may impair the code readability. If there is a need to invalidate a specific section of the code, one way would be to comment out that section. It would be desirable to establish a local coding rule on how comment out should be used. (For example, set a rule to use only `//` comment). Any section of the code can also be invalidated without using comment out by specifying that section in between `#if 0` and `#endif`.

［**Related rules**］M1.12.1, M4.7.2

M1.2 Do not writing confusingly.



(1) Only one variable shall be declared in one declaration statement (avoid multiple declarations.)
(2) Automatic variables of the same type used for the similar purposes may be declared in one declaration statement, but variables with initialization and variables without initialization shall not be mixed.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1)
int i;
int j;
Compliant example of (2)
int i, j;
int k = 0;

int *p;
int i;
```

Non-compliant example

```
Non-compliant example of (1)
int i, j;
Non-compliant example of (2)
int i, j, k = 0;
  /* A variable with
     initialization and variables without
     initialization are mixed (Non-compliant) */

int *p, i;
 /* Variables of different types
     are mixed (Non-compliant) */
```

If the declaration is `int *p;`, the type of `p` is `int*`. However, if the declaration is `int *p, q;`, the type of `q` becomes `int` instead of `int*`.

［**Related rule**］M1.6.1

M1.2.2 Suffixes shall be added to constant descriptions that can use them to indicate appropriate types. Only an uppercase letter "L" shall be used for a suffix indicating a long type integer constant.

Preference guide	
Rule specification	

Compliant example

```
void func(long int);
…
float f;
long int l;
unsigned int ui;

f = f + 1.0F;  /* Explicitly state that it is a
                  float operation */
func(1L); /* Description of L should be an
             uppercase letter */
if (ui < 0x8000U) { /* Explicitly state that it
                       is an unsigned comparison */
    …
```

Non-compliant example

```
void func(long int);
…
float f;
long int l;
unsigned int ui;

f = f + 1.0;
func(11); /* 11 (numeral "1" and alphabet letter
             "l" ) can get easily confused with 11
            (numeral "1" and numeral "1" ). */
if (ui < 0x8000) {
    …
```

Basically, when there is no suffix, an integer constant will be an `int` type and a floating constant will be a `double` type. However, when an integer constant value that cannot be expressed with an `int` type is described, its type will be the one that can express that value. Therefore, `0x8000` will be `unsigned int` if `int` is 16 bits, and `signed int` if `int` is 32 bits. If you would like to use it as `unsigned`, it is necessary to explicitly describe "U" as the suffix. In addition, in case of a target system where the operation speed differs between floating point number of `float` type and that of `double` type, when performing operations between a `float` type variable and a floating constant without a suffix "F," constant, it should be noted that the operation will be a `double` type.

For floating constants, writing at least one digit on both sides of the decimal point will make them easily recognizable as floating constants.

［**Related rule**］M1.8.5

M1.2.3 When expressing a long string literal, successive string literals shall be concatenated without using newlines within the string literal.

Preference guide	
Rule specification	

Compliant example

```
char abc[] = "aaaaaaaa¥n"
             "bbbbbbbb¥n"
             "cccccccc¥n";
```

Non-compliant example

```
char abc[] = "aaaaaaaa¥n¥
             bbbbbbbb¥n¥
             cccccccc¥n";
```

Long strings that extend to multiple lines will become easier to read by describing them as concatenation of multiple string literals.

M1.3 Do not write in an unconventional style.

M1.3.1 Expressions evaluating to true or false shall not be described in `switch ( expression )`.

Preference guide	●
Rule specification	

Compliant example

```
if (i_var1 == 0) {
  i_var2 = 0;
} else {
  i_var2 = 1;
}
```

Non-compliant example

```
switch (i_var1 == 0) {
case 0:
  i_var2 = 1;
  break;
default:
  i_var2 = 0;
  break;
}
```

When an expression evaluating to `true` or `false` is used in a `switch` statement, the number of branch directions will be two, and the necessity of using the `switch` statement as a multiway branch command becomes low. Compared to if statements, switch statements have a higher possibility of errors, such as, writing the default clause wrongly or missing `break` statements. Therefore, it is recommended to use `if` statements unless the number of branch directions is three or more.

M1.3.2 The `case` labels and `default` label in a `switch` statement shall be described only in the compound statement (excluding nested compound statements) within the body of the `switch` statement.

Preference guide	
Rule specification	

Compliant example

```
switch (x) {
case 1:
    {
      …
    }
    …
    break;
case 2:
    …
    break;
default:
    …
    break;
}
```

Non-compliant example

```
switch (x) { /* Compound statement of the
                switch statement body */
case 1:
    {  /* Nested compound statement */
case 2: /* Do not describe case label in
           nested compound statement */
       …
    }
    …
    break;
default:
    …
    break;
}
```

M1.3.3 The types shall be explicitly described for definitions and declarations of functions and variables.

Preference guide	
Rule specification	

Compliant example

```
extern int global;

int func(void) {
    …
}
```

Non-compliant example

```
extern global;

func(void) {
    …
}
```

If data types are not described in definitions and declarations of functions or variables, they are interpreted as `int` type. Explicitly specifying data types improves readability. In C99 language standard, these descriptions that do not explicitly specify the data types are prohibited and will be detected as error by the compiler.

[**Related rule**] M4.5.1

M1.4 Write in a style that clearly specifies the order of evaluation of operations.

M1.4.1 Expressions described at the right hand and left hand of && and || operations shall be either expressions that do not include binary operation or expressions enclosed with ( ). However, if only && operations or only || operations are successively combined, it is not necessary to enclose each && and || expression with ( ).

Preference guide	
Rule specification	

Compliant example

```
if ((x > 0) && (x < 10))
if ((x != 1) && (x != 4) && (x != 10))
if (flag_tb[i] && status)
if (!x || y)
```

Non-compliant example

```
if (x > 0 && x < 10)
if (x != 1 && x != 4 && x != 10)
```

The objective of this rule is to write an expression that prevents confusion in understanding the order of precedence of primary operation of each operand in && or ||. Its aim is to highlight the operation of each operand in && or || to improve the readability by enclosing the expression that contains an operator other than unary, postfix and cast operators with ( ) . Another rule that may be considered is to enclose ! operation with ( ) because the order of precedence may be confusing to beginners.

[**Related rules**] R2.3.2 M1.5.2

M1.4.2 《Usage of parentheses to explicitly indicate operator precedence shall be defined.》

Preference guide	
Rule specification	Define

Compliant example

```
a = (b << 1) + c;
       - or -
a = b << (1 + c);
```

Non-compliant example

```
a = b << 1 + c; /* There is a possibility that
                   the operator precedence is
                   misunderstood */
```

Operator precedence in C language is difficult to capture. Therefore, set a rule as exemplified below to improve its readability. If an expression contains multiple binary operators that differs in the order of operation priority, parentheses ( ) shall be used to explicitly indicate the operator precedence, provided that the parentheses ( ) may be omitted in four arithmetic operations.

To learn more about the operator precedence and its interpretation, refer to MISRA C:2012 Rule 12.1 (p.103).

[**Related rule**] M1.5.1

M1.5 Explicitly describe operations that may lead to misunderstanding when omitted.

M1.5.1 A function identifier (function name) shall only be used with either a preceding "&", or with a parenthesized parameter list, which may be empty. 【MISRA C:2004 16.9】

Preference guide	
Rule specification	

Compliant example

```
void  func(void);
void (*fp)(void) = &func;

if (func()) {
```

Non-compliant example

```
void  func(void);
void (*fp)(void) = func; /* Non-compliant:
                            There is no & */
if (func) {   /* Non-compliant: Address is
                 obtained rather than calling the
                 function. It might be mistakenly
                 written as afunction call without
                 arguments. */
```

In C language, if a function name is written alone, it means obtaining the function address, and not calling the function. This means that, for obtaining the function address, there is no need of placing & in front of the function name. However, the function name without a preceding &, in some cases, may be misunderstood that it is for a function call (for example, when using languages like Ada and Ruby that write only the name to call a subprogram without arguments). By following the rule to add & when obtaining the function address, it will become easier to detect mistakes in function names written as they are without & and subsequent ( ) .

[**Related rule**] M1.4.2

M1.5.2 Comparisons with zero (0) shall be explicitly written in conditional expressions.

Preference guide	
Rule specification	

Compliant example

```
int x = 5;

if (x != 0) {
 …
}
```

Non-compliant example

```
int x = 5;

if (x) {
 …
}
```

In conditional expressions, when the result of the expression is zero (0), it is treated as `false`, and non-zero is treated as `true`. Therefore, comparative operations may be omitted in conditional expressions. However, such description may cause unintended behavior. For this reason, the comparisons should not be omitted to make the intention of the program explicit. Moreover,since `bool`, `true` and `false` are defined as macros in `<stdbool.h>`, any of them should be used to describe a type that represents `true` or `false`, or a constant that represents a `true` or `false` value.

[**Related rules**] R2.2.1 M1.4.1

M1.6 Use one area for one purpose.

M1.6.1 Variables shall be prepared for each purpose.

Preference guide	
Rule specification	

Compliant example

```
 /* Counter variable and work variable for
    replacement are different */
for (i = 0; i < MAX; i++) {
    data[i] = i;
}
if (min > max) {
    wk = max;
    max = min;
    min = wk;
}
```

Non-compliant example

```
 /* Counter variable and work variable for
    replacement are the same */
for (i = 0; i < MAX; i++) {
    data[i] = i;
}
if (min > max) {
    i = max;
    max = min;
    min = i;
}
```

Reusing variables should be avoided as it impairs readability and increases the risk of causing errors during modification.

[**Related rule**] M1.2.1

(1) Unions shall not be used.【MISRA C:2004 18.4】 (2) If unions are used, the same members that are assigned values shall be referenced.

Preference guide	
Rule specification	Choose

Compliant example

```
compliant example of (2)
/* When the typ is INT, i_var is valid.
   When the type is CHAR, c_var[4] is valid. */
struct stag {
  int type;
  union utag {
    char c_var[4];
    int i_var;
  } u_var;
} s_var;
…
int i;
…
if (s_var.type == INT) {
    s_var.u_var.i_var = 1;
}
…
i = s_var.u_var.i_var;
```

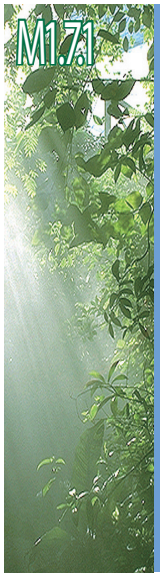
Non-compliant example

```
Non-compliant example of (2)
/* When the typ is INT, i_var is valid.
   When the type is CHAR, c_var[4] is valid. */
struct stag {
  int type;
  union utag {
    char c_var[4];
    int i_var;
  } u_var;
} s_var;
…
int i;
…
if (s_var.type == INT) {
    s_var.u_var.c_var[0] = 0;
    s_var.u_var.c_var[1] = 0;
    s_var.u_var.c_var[2] = 0;
    s_var.u_var.c_var[3] = 1;
}
…
i = s_var.u_var.i_var;
```

Union allows the same memory space to be declared with areas of different sizes. However, since the way bits of data overlap among members is implementation-dependent, unexpected behavior may occur. Therefore, if union is going to be used, follow rule (2) as a precautionary measure.

[**Related rule**] R2.1.3

M1.7 Do not reuse names.



Preference guide	○
Rule specification	

The rules below shall be followed for name uniqueness.

1. An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.【MISRA C:2012 R5.3】
2. A typedef name shall be a unique identifier.【MISRA C:2013 R5.6】
3. A tag name shall be a unique identifier.【MISRA C:2012 R5.7】
4. Identifiers that define objects or functions with external linkage shall be unique.【MISRA C:2012 R5.8】
5. Identifiers that define objects or functions with internal linkage should be unique.【MISRA C:2012 R5.9】
6. No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.【MISRA C:2004 5.6】

Compliant example

```
int var1;
void func(int arg1) {
  int var2;
  var2 = arg1;
  {
    int var3;
    var3 = var2;
    …
  }
}
```

Non-compliant example

```
int var1;
void func(int arg1) {
  int var1;    /* The same name of a variable
                  outside the function is used
*/
  var1 = arg1;
  {
     int var1;  /* The same name of a variable
                   in the outer scope is used */
     …
     var1 = 0;  /* Intention of which var1 is
                   assigned is unclear */
     …
  }
}
```

The program will become easier to read by using unique names within the program, except for cases like automatic variables where the scope is limited.

In C language, in addition to the scope defined by file and block, names have the following four name spaces that vary according to the category they belong to:

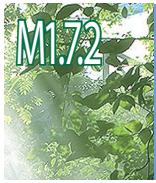
1. Label 2.Tag 3. Member of structure or union 4. Other identifiers

*Macro has no name space.

The language specification allows using the same name to different identifiers if their name spaces differ, but this rule restricts such usage for the purpose of improving the readability of the program.

As the exception of rule 2., the typedef name may be the same as the name of the structure member, union member or tag related to that typedef. As the exception of rule 3., the tag name may be the same as the name of the typedef related to that tag.

[**Related rule**] M4.3.1



Names for functions, variables and macros in the standard library shall not be redefined or reused. In addition, those macro names shall not be undefined.

Preference guide	○
Rule specification	

Compliant example

```
#include <string.h>
void *my_memcpy(void *arg1, const void *arg2,
size_t size) {
      …
}
```

Non-compliant example

```
#undef NULL
#define NULL  ((void *)0)

#include <string.h>
void *memcpy(void *arg1, const void *arg2, size_
t size) {
      …
}
```

Redefining names for functions, variables and macros defined in the standard library degrades the readability of the program.

[**Related rule**] M1.8.2

Names (variables) that start with an underscore shall not be defined.

Preference guide	○
Rule specification	

Compliant example

–

Non-compliant example

```
int _Max1;   /* Reserved */
int __max2;  /* Reserved */
int _max3;   /* Reserved */

struct S {
   int _mem1; /* Not reserved, but shall not be
                 used */
};
```

C language standard defines the following names as reserved.

(1) Name that starts with an underscore and is followed by either an uppercase letter or another underscore;
Examples：`_Abc`, `__abc`

(2) Names that start with an underscore
These names are reserved for variables or functions with file scope and for tag names.
When the reserved names are redefined, the behavior of the compiler will not be guaranteed.
Names that start with an underscore and are followed by a lowercase letter are not reserved for use outside the file scope. But to make it easy to remember, this rule restricts the use of all names starting with an underscore.

[**Related rule**] M1.8.2

M1.8 Do not use language specifications that are likely to cause misunderstanding.

M1.8.1 The right-hand operand of a logical && or || operator shall not contain side effects.【MISRA C:2012 R13.5】

Preference guide	
Rule specification	

Compliant example

```
volatile int *io_port = ...; /* Address for
                             memory mapped I/O */
int io_result = *io_port;
/* I/O is processed, regardless of the
   conditions of the if statement */
if ((x != 0) && (io_result > 0)) {
    …
}
```

Non-compliant example

```
volatile int *io_port = ...; /* Address for
                             memory mapped I/O */
/* Whether I/O is processed or not varies,
   depending on the conditions of the if
   statement */
if ((x != 0) && (*io_port > 0)) {
    …
}
```

The right-hand side of && or || operators may not be executed, depending on the result of the condition of their left-hand side. Take, for example, an expression with a side effect of incrementing. It this expression is written on the right-hand side, whether the increment is executed or not will be difficult to understand, because it depends on the condition of the left-hand side. Therefore, expressions with side effects shall not be described on the right-hand side of && or || operators.

[**Related rules**] R3.6.1, R3.6.2

M1.8.2 C macros shall only expand to a braced initializer, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.【MISRA C:2004 19.4】

Preference guide	
Rule specification	

Compliant example

```
#define   START 0x0410
#define   STOP  0x0401
```

Non-compliant example

```
#define  BIGIN  {
#define  END    }
#define  LOOP_STAT for(;;) {
#define  LOOP_END  }
```

Macro definitions can be leveraged to make the code look like it is written in a language other than C, or greatly reduce the amount of code. However, using macros for these purposes will degrade readability. It is important to use macros only where coding and modification errors can be avoided. For do-while-zero, see MISRA C:2004.

[**Related rule**] M1.7.2

M1.8.3 `#line` shall not be used, unless it is automatically generated by a tool.

Preference guide	
Rule specification	

`#line` serves as the means to intentionally modify file names or line numbers of warning or error messages output from the compiler. It is provided under the assumption that code is generated by tools, and is not intended to be used directly by the programmers.

M1.8.4 Sequences of three or more characters starting with ?? and alternative tokens shall not be used.

Preference guide	
Rule specification	

Compliant example

```
s = "abc?(x)";
```

Non-compliant example

```
s = "abc??(x)";  /* Compilers that can process
                    trigraph sequences interpret
                    this as “abc[x)” */
```

C language standard defines trigraph sequences and alternative tokens, assuming that there may be cases where some characters cannot be used for coding, depending on the environment used for development.
The following nine three-character patterns, known as trigraph sequences:

```
??=  ??(  ??/  ??)  ??’  ??<  ??!  ??>  ??-
```

can be replaced respectively at the beginning of preprocessing with the following corresponding single-character counterparts:

```
#    [    \    ]    ^    {    |    }    ~
```

The following two-character patterns, known as digraph sequences:

```
<%  %>  <:  :>  %:  %:%:
```

are handled respectively as equivalent to

```
{   }   [   ]   #   ##
```

in the lexical analysis.
C99 defines the following macros in the header `<iso646.h>`

```
and and_eq bitand bitor compl
not not_eq or or_eq xor xor_eq
```

as alternative spellings that correspond respectively to the following tokens.

```
&&  &=  & |  ~  !  !=  ||  |=  ^  ^=
```

Since trigraph sequences and alternative tokens are not frequently used, many compilers support them as an optional feature.

M1.8.5 A sequence starting with zero (0) that is two or more digits long shall not be used as a constant.

Preference guide	
Rule specification	

Compliant example

```
/* Digits are not aligned for better
   appearance */
a = 0;
b = 8;
c = 100;
```

Non-compliant example

```
/* Examples of aligning the digits for better
   appearance.*/
a = 000; /* Interpreted as zero (0) in octal
            notation */
b = 010; /* Interpreted as eight (8) and not as
            ten (10) in decimal notation */
c = 100; /* Interpreted as hundred (100) in
            decimal notation */
```

Constants starting with zero (0) are interpreted as octal. No zero (0) can be added in front of decimal numbers to align their digits for the purpose of appearance (i.e.: zero padding is not allowed).

[**Related rule**] M1.2.2

M1.9 When writing in an unconventional style, explicitly state its intention.

M1.9.1 If statements that do nothing need to be intentionally described, comments or empty macros shall be used to make them noticeable.

Preference guide	○
Rule specification	

Compliant example

```
for (;;) {
     /* Waiting for interruption */
}

#define NO_STATEMENT
i = COUNT;
while ((--i) > 0) {
    NO_STATEMENT;
}
```

Non-compliant example

```
for (;;) {
}

i = COUNT;
while ((--i) > 0);
```

[**Related rule**] M1.1.1

M1.9.2 《The unified style of writing infinite loops shall be defined.》

Preference guide	○
Rule specification	Define

Define the unified style of writing infinite loops, for example, by selecting from one of the following:

- Write the infinite loops uniformly as `for(;;)`; .
- Write the infinite loops uniformly as `while(1)`; .
- Use the macro defined for the infinite loop.

M1.10 Do not embed magic numbers.

M1.10.1 A meaningful constant shall be used after defining it as a macro.

Preference guide	○
Rule specification	

Compliant example

```
#define MAXCNT 8
if (cnt == MAXCNT) {
  …
```

Non-compliant example

```
if (cnt == 8) {
  …
```

By defining a constant as a macro, its meaning can be stated explicitly. When modifying a program where the same constant is used in multiple places, modification errors can be prevented much more easily if this same constant is defined as a macro, because then, there will only be a need to modify one macro.

For data size, however, use `sizeof` instead of using a macro.

[**Related rule**] M2.2.4

M1.11 Explicitly state the area attributes.

Read-only areas shall be declared as const type.

Preference guide	○
Rule specification	

Compliant example

```
const volatile int read_only_mem; /* Read-only
                                     memory */
const int constant_data = 10; /* Read-only
                         data that does not
                         require memory
                         allocation */
/* Only reads the contents pointed by arg */
void func(const char *arg, int n) {
   int   i;
   for (i = 0; i < n; i++) {
      put(*arg++);
   }
}
```

Non-compliant example

```
int read_only_mem;   /* Read-only memory */
int constant_data = 10; /* Read-only data that
                    does not require memory
                    allocation */
/* Only reads the contents pointed by arg */

void func(char *arg, int n) {
   int   i;
   for (i = 0; i < n; i++) {
      put(*arg++);
   }
}
```

When a variable is only referenced and not modified, declaring it as `const`-qualified variable makes it clear that it is not modified.That is why read-only variables should be `const`-qualified. Moreover, a memory that is only referenced by the program but modified by other execution units should be declared with `const volatile` qualification so that the compiler can check and prevent the program from renewing it by mistake. Furthermore, function interfaces can be clearly stated by adding `const`s to parameters when the memory spaces indicated by the parameters are only referenced in function processing.

[**Related rule**] R1.1.2

M1.11.2 Areas that may be updated by other execution units shall be declared as volatile.

Preference guide	○
Rule specification	

Compliant example

```
volatile int x = 1;
while (x == 0) {
  /* x is not modified within the loop and is
     modified by other execution units */
}
```

Non-compliant example

```
int x = 1;
while (x == 0) {
  /* x is not modified within the loop and is
     modified by other execution units */
}
```

Areas qualified as volatile prohibit the compiler from optimizing them. Prohibition of optimization means that executable object is generated strictly to every description, including even those considered logically as unnecessary of processing. Suppose there is a description "`x;`" that has no meaning logically except for only referencing variable `x`. If it is not qualified as `volatile` , the compiler will normally ignore such statement and will not generate an executable object. Whereas, if it is qualified as `volatile` , the compiler will generate an executable object that only references variable `x` (loads it to the register). This description can be assumed to have meaning in indicating the interface to IO registers (mapped to the memory) that are reset when the memory is read. Embedded software has IO registers for controlling hardware that should be qualified as volatile when considered appropriate, based on their characteristics.

M1.11.3 《Rules for variable declaration and definition for ROMization shall be defined》

Preference guide	
Rule specification	Define

Compliant example

```
const int x = 100;  /* Allocate to ROM */
```

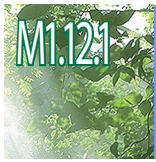
Non-compliant example

```
int x = 100;
```

Variables qualified as const can be allocated to ROMization target areas. For example, when developing a program where ROMization is applied, qualify the read-only variables as const , and specify the name of the section to which these variables are allocated by, such as, `#pragma`.

[**Related rule**] R1.1.2

M1.12 Correctly describe the statements even if they are not compiled.



Correct code shall be described even if it is going to be deleted by the preprocessor.

Preference guide	
Rule specification	

```
#if 0
 /* */
#endif

#if 0
…
#else
 int var;
#endif

#if 0
 /* I don't know */
#endif
```

```
#if 0
 /*
#endif

#if 0
…
#else1
 int var;
#endif

#if 0
 I don't know
#endif
```

[**Related rule**] M1.1.2

Maintainability 2

Write in a style that can prevent modification errors.

One of the patterns that allows bugs to slip into a program easily is when other bugs are created by mistake while fixing detected bugs. Especially if it has been a while since the source code was written or if an engineer other than the creator modifies the source code, unexpected misunderstanding may occur.

Efforts to reduce such modification errors as much as possible are strongly desired.

M2.1 Clarify the grouping of structured data and blocks.

M2.1.1 If arrays and structures are initialized with values other than 0, their structural form shall be indicated by using braces '{ }'. Data shall be described without any omission, except when all values are 0.

Preference guide	○
Rule specification	

Compliant example

```
int arr1[2][3] = {{0, 1, 2}, {3, 4, 5}};
int arr2[3] = {1, 1, 0};
```

Non-compliant example

```
int arr1[2][3] = {0, 1, 2, 3, 4, 5};
int arr2[3] = {1, 1};
```

In initialization of arrays and structures, at least a pair of braces '{ }' is required, but in this case, it is difficult to see how the data for initialization are assigned. It is safer to create blocks according to the structure, and fully describe the data for initialization without omitting any.

[**Related rules**] R1.2.1 M4.5.3

M2.1.2 The body of if, else if, else, while, do, for, and switch statements shall be enclosed into blocks.

Preference guide	
Rule specification	

Compliant example

```
if (x == 1) {
  func();
}
```

Non-compliant example

```
if (x == 1)
  func();
```

If there is only one statement that is controlled by, such as, an `if` statement, there is no need to enclose this statement into a block. However, when the program is modified and this single statement is changed into multiple statements, there is a possibility of forgetting to enclose these multiple statements into a block. To prevent such modification errors, enclose the body of each controlled statement into a block.

M2.2 Localize access ranges and related data.

Variables used only in one function shall be declared within the function.

Preference guide	●
Rule specification	

Compliant example

```
void func1(void)
{
  static int x = 0;
  if (x != 0) {    /* Refer to the value in the
                      immediately preceding call */
      x++;
  }
    …
}
void func2(void)
{
  int  y = 0;    /* Initialize each time */
   …
}
```

Non-compliant example

```
int x = 0; /* x is accessed only from func1 */
int y = 0; /* y is accessed only from func2 */
void func1(void) {
  if ( x != 0 ) {  /* Refer to the value in the
                      immediately preceding call */
      x++;
  }
   …
}
void func2(void) {
  y = 0;      /* Initialize each time */
   …
}
```

To declare variables in functions, it is sometimes effective to declare them with `static` storage class specifiers. The following positive effects can be expected if `static` is specified:

- Static memory space is reserved and the space is valid until the end of the program (Without `static`, generally, stack memory is used and is valid until the end of the function.)
- Initialization occurs only once after the program is started and if a function is called more than once, the value assigned in the previous call is retained.

Therefore, among the variables accessed only within a function, the variables with values that are retained even after the function terminates should be declared with `static` storage class specifiers.
In addition, declaring a large memory space for an automatic variable may cause stack overflow. When there is such risk, one preventive measure is to use `static` to reserve static memory space even if the values do not need to be retained after the function terminates. However, when using `static` for such purpose, its intention should be explicitly stated by, such as, comments (to prevent potential misunderstanding that `static` has been used by mistake).

[**Related rule**] M2.2.2

Variables accessed by several functions defined in the same file shall be declared with `static` in the file scope.

Preference guide	○
Rule specification	

Compliant example

```
/* x is not accessed by other files */
static int  x;
void func1(void) {
  …
  x = 0;
  …
}
void func2(void) {
  …
  if (x == 0) {
    x++;
  }
  …
}
```

Non-compliant example

```
/* x is not accessed by other files */
int   x;
void func1(void) {
  …
  x = 0;
  …
}
void func2(void) {
  …
  if (x==0) {
     x++;
  }
  …
}
```

The fewer the global variables, the higher the readability of the entire program becomes. To prevent the number of global variables from increasing, `static` storage class specifiers should be used as much as possible.

[**Related rules**] M2.2.1, M2.2.3

Functions that are called only by functions defined in the same file shall be `static`.

Preference guide	○
Rule specification	

Compliant example

```
/* func1 is not called from functions in other
   files */
static void func1(void) {
  …
}
void func2(void) {
  …
  func1();
  …
}
```

Non-compliant example

```
/* func1 is not called from functions in other
   files */
void func1(void) {
  …
}
void func2(void) {
  …
  func1();
  …
}
```

The fewer the global functions, the higher the readability of the entire program becomes. To prevent the number of global functions from increasing, `static` storage class specifiers should be used as much as possible.

[**Related rule**] M2.2.2

enum shall be used rather than #define when defining related constants.

Preference guide	
Rule specification	

Compliant example

```
enum ecountry {
  ENGLAND, FRANCE, …
} country;
enum eweek {
  SUNDAY, MONDAY, …
} day;
…
if ( country == ENGLAND ) {
if ( day == MONDAY ) {
if ( country == SUNDAY ) { /* It is possible
                            to check by tools */
```

Non-compliant example

```
#define   ENGLAND  0
#define   FRANCE   1
#define   SUNDAY   0
#define   MONDAY   1
int    country, day;
…
if ( country == ENGLAND ) {
if ( day == MONDAY ) {
if ( country == SUNDAY ) { /* It is impossible
                              to check by tools */
```

To define the constants that are related like a set, use the enumeration type. By defining related constants as `enum` type, and using this type, mistakes caused by the use of incorrect values can be prevented. While macro names defined by `#define` are expanded at the preprocessing stage and the compiler does not process those names, `enum` constants defined by `enum` declaration will be the names processed by the compiler. The names processed by the compiler are easier to debug, because they can be referenced during symbolic debugging.

[**Related rules**] M1.10.1 P1.3.2

From the standpoint of software maintainability, there is no better software than those created from simply written programs.

C language enables the structuring of software by, such as, dividing the program into separate source files and functions. Structured programming that represents program structure through three forms: sequence, selection and repetition, is also one of the applicable techniques to write simple software programs. Writing simple software descriptions through effective use of software structuring is strongly desired. Moreover, particular attention should also be given to writing styles applied to describe, such as, iteration processing, assignment and operations, as some may make the program difficult to maintain.

Maintaiability 3.1	Do structured programming.
Maintaiability 3.2	Limit the number of side effects per statement to one.
Maintaiability 3.3	Write expressions that differ in purpose separately.
Maintaiability 3.4	Do not use complicated pointer operations.

M3.1 Do structured programming.

There should be no more than one `break` or `goto` statement used to terminate any iteration statement. 【MISRA C:2012 R15.4】

Preference guide	
Rule specification	

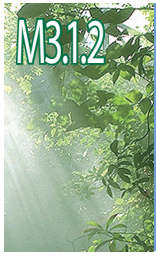
Compliant example

```
end = 0;
for (i=0; loop iteration condition && !end; i++)
{
  Iterated processing 1;
  if (termination condition1 || termination
      condition2) {
    end = 1;
  } else {
    Iterated processing 2;
  }
}
-or-
for (i=0; loop iteration condition; i++) {
  Iterated processing 1;
  if (termination condition1 || termination
      condition2) {
    break;
  }
  Iterated processing 2;
}
```

Non-compliant example

```
for (i=0; loop iteration condition; i++) {
  Iterated processing 1;
  if (termination condition1) {
    break;
  }
  if (termination condition1) {
    break;
  }
  Iterated processing 2;
}
```

This rule is to prevent the program logic from becoming complex. If a flag has to be prepared only for eliminating the `break` statement, sometimes it is better not to prepare the flag and to use a `break` statement. (Be careful, however, when using an end flag like in the case shown above as compliant example, because it may complicate the program.)



(1) The goto statement shall not be used.
(2) When using a goto statement, the destination to jump to shall be the label declared after the goto statement that is within the block enclosing the goto statement.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1) and (2)
for (i = 0; loop iteration condition; i++) {
  Iterated processing;
}

Compliant example of (2)
if (err != 0) {
  goto ERR_RET;
}
 …
ERR_RET:
  end_proc();
  return err;
}
```

Non-compliant example

```
Non-compliant example of (1) and (2)
  i = 0;
LOOP:
  Iterated processing;
  i++;
  if (loop iteration condition) {
    goto LOOP;
  }
```

These rules are to prevent the program logic from becoming complex. The purpose is not to eliminate all the `goto` statements. The important point is to eliminate unnecessary `goto` statements to prevent the program from becoming complicated (i.e., not being able to read it straightforwardly from top to bottom). In some cases, the readability can actually be improved by writing `goto` statements. Therefore, when programming, keep in mind how simply the logic can be expressed.

For example, `goto` statement can be useful to make the program simple, such as, when it is used to jump to error processing or exit from multiple loops.

(1) Each `case` clause and `default` clause in a `switch` statement shall always end with a `break` statement.

(2) If the `case` clause or `default` clause in a `switch` statement is not going to be ended with a `break` statement, 《a project-specific comment shall be defined》 and that comment shall instead be inserted.

Preference guide	○
Rule specification	Choose Define

Compliant example

```
Compliant example of (1) and (2)
switch (week) {
case A:
  code = MON;
  break;
case B:
  code = TUE;
  break;
case C:
  code = WED;
  break;
default:
  code = ELSE;
  break;
}

Compliant example of (2)
dd = 0;
switch (status) {
case A:
  dd++;
  /* FALL THROUGH */
case B:
```

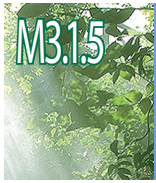
Non-compliant example

```
Non-compliant example of (1) and (2)
/* No matter what the value of week is, the
   code will be ELSE ==> bug */
switch (week) {
case A:
  code = MON;
case B:
  code = TUE;
case C:
  code = WED;
default:
  code = ELSE;
}

/* This This is a case where processing of case
   B is continued after dd++, but it is non-
   compliant not only to (1) but also to (2)
   because there is no comment. */
dd = 0;
switch (status) {
case A:
  dd++;
case B:
```

One of the typical examples of coding error is caused by forgetting to write the `break` statement in a `switch` statement in C language. To prevent it, avoid writing a `case` statement without the `break` statement unnecessarily. If the code is intended to continue processing to the next `case` without the `break` statement, always insert a comment to explicitly indicate that the absence of the `break` statement is not a problem. Define what kind of comment to insert in such `case` in the coding convention. As one example, `/* FALL THROUGH */` is a comment that is frequently used.

[**Related rule**] R3.5.2

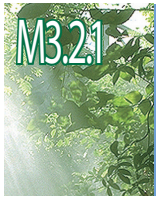


(1) A function shall end with one `return` statement.
(2) A `return` statement to return in the middle of processing shall be written only in case of recovery from abnormality.

Preference guide	
Rule specification	Choose

This rule is to prevent the program logic from becoming complex. When a program has many entry or exit points, they will not only complicate the program but will also make it difficult to set `break` points during debugging. In C language, there is only one entry point for a function but the exit points are where the `return` statements are written.

M3.2 Limit the number of side effects per statement to one.



(1) Comma expressions shall not be used.
(2) Comma expressions shall not be used, other than in expressions for initializing or updating in `for` statements.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1) and (2)
a = 1;
b = 1;

j = 10;
for (i = 0; i < 10; i++) {
   …
   j--;
}

Compliant example of (2)
for (i = 0, j = 10; i < 10; i++, j--) {
   …
}
```

Non-compliant example

```
Non-compliant example of (1) and (2)
a = 1, b = 1;

Non-compliant example of (1)
for (i = 0, j = 10; i < 10; i++, j--) {
   …
}
```

In general, the use of comma expressions make the program complicated. However, the progam may sometimes become easier to understand in expressions for initializing and updating in `for` statements by using comma expressions to collectively describe all the pre-loop operations as one set and all the loop-end operations as another set.

[**Related rule**] M3.3.1

M3.2.2 Multiple assignments shall not be written in one statement, except when the same value is assigned to multiple variables.

Preference guide	○
Rule specification	

Compliant example

```
x = y = 0;
```

Non-compliant example

```
y = (x += 1) + 2;
y = (a++) + (b++);
```

Assignments include the compound assignments (+= -=, etc) beside the simple assignment (=). Multiple assignments may be written in one statement, but since they impair readability, one statement should contain only one assignment.
However, “commonly used conventional descriptions” shown below do not impair readability in many cases. They may be treated as exceptions of this rule.

```
c = *p++;
*p++ = *q++;
```

M3.3 Write expressions that differ in purpose separately.

M3.3.1 The three expressions of a `for` statement shall be concerned only with loop control. 【MISRA C:2004 13.5】

Preference guide	
Rule specification	

Compliant example

```
for (i = 0; i < MAX; i++) {
   …
  j++;
}
```

Non-compliant example

```
for (i = 0; i < MAX; i++, j++) {
  …
}
```

In MISRA C:2012, rules 13.5 and 13.6 in MISRA C:2004 have been consolidated into R14.2, which states that “a `for` loop shall be well-formed”. According to this rule, the first clause of a `for` statement, for example, shall either be empty, assign a value in the loop counter or define and initialize the loop counter (C99).

[**Related rules**] M3.2.1, M3.3.2

M3.3.2 Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.【MISRA C:2004 13.6】

Preference guide	
Rule specification	

Compliant example

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

Non-compliant example

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

See M3.3.1.

[**Related rule**] M3.3.1

M3.3.3 (1) Assignment operators shall not be used in expressions to examine true or false. (2) Assignment operators shall not be used in expressions to examine true or false, except for conventionally used notations.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1) and (2)
p = top_p;
if (p != NULL) {
  …
}

Compliant example of (1)
c = *p++;
while (c != '¥0') {
  …
  c = *p++;
}
```

Non-compliant example

```
Non-compliant example of (1) and (2)
if (p = top_p) {
  …
}

Non-compliant example of (1)
while (c = *p++) {
  …
}
/* Since this is an expression used
   conventionally, it is compliant to (2).
   (However, be careful of its usage, because its
   readability depends on the programmer's coding
   skills.) */
```

The following are the expressions to examine true or false:
`if (`*expression*`)`, `for ( ;` *expression* `; )`, `while (` *expression* `)`, `(` *expression* `)?:`, *expression* && *expression*, *expression* || *expression*

M3.4 Do not use complicated pointer operations.

Three or more pointer indirections shall not be used.

Preference guide	
Rule specification	

Compliant example

```
int **p;
typedef char **strptr_t;
strptr_t q;
```

Non-compliant example

```
int ***p;
typedef char **strptr_t;
strptr_t *q;
```

Since it is difficult to understand the changes in the pointer values in three or more levels, multiple pointer indirections impair maintainability.

Recently, developing programs under the shared efforts of multiple programmers has become a widely accepted approach in software projects. If these programmers apply different coding styles to write their assigned portion of the source code, the reviewers or other programmers may later face difficulty checking what each programmer has written. Moreover, if the naming of variables, information to be described in a file, and the order to describe the information, among others, are not uniform, unexpected misunderstanding or errors may arise from such inconsistencies. This is why writing the source code as much as possible according to a unified coding style in a single project or within the organization is often said to be desirable.

Maintaiability 4.1	Unify the coding styles.
Maintaiability 4.2	Unify the style of writing comments.
Maintaiability 4.3	Unify the naming conventions.
Maintaiability 4.4	Unify the contents to be described in a file and the order of describing them.
Maintaiability 4.5	Unify the style of writing declarations.
Maintaiability 4.6	Unify the style of writing null pointers.
Maintaiability 4.7	Unify the style of writing preprocessor directives.

M4.1 Unify the coding styles.

《Conventions regarding the style of using, such as, the braces'{ }', indentation and space shall be defined.》

Preference guide	○
Rule specification	Define

To make the code easier to read, it is important to unify the coding style applied in the project.
When defining a new style convention to be followed in the project, the recommended approach would be to select from already existing coding styles. Existing coding styles have been developed from various schools, and many programmers create their programs based on any one or more of these pre-established styles. One of the benefits of selecting from these existing coding styles is that the format can be easily specified by the format commands available in editors and other tools. If no coding style is clearly specified in the existing project, the recommendation would be to define a coding convention that matches most closely with the current source code.
What is most important in deciding on the style convention is not in "deciding what kind of style" to define, but is in "defining a unified style to be followed".
Explained below are the set of style-related items to be defined:

(1) Position of braces '{ }'

Unify the position to place the braces '{ }' so that the beginning and end of a block will become easier to read (see Representative styles).

(2) Indentation

Indentation makes a group of declarations and operations easier to read. For unified use of indentation, define the following:

- Whether to use spaces or tabs for indentation;
- If spaces are used, how many space characters are set for one indent? If tabs are used, how many characters are set for each tab?

(3) How to use spacing

Spacing makes the code easier to read. For example, define the following rules:

- Add a space before and after binary and ternary operators, except for the following operators.
  [ ], ->, . (period), , (comma operator)
- Do not add a space between unary operator and its operand.

By applying these rules, coding errors that are attributable to compound assignment operators will become easier to detect.

[Examples]

```
x=-1;    /* Intended to write x-=1, but made a mistake => difficult to distinguish */
x =- 1;  /* Intended to write x-=1, but made a mistake => easy to distinguish */
```

Besides those stated above, the following rules are also defined in some cases:

- Add a space after a comma (except for commas for parameters in macro definitions)
- Add a space before the left parenthesis enclosing control statements such as, `if` and `for`. Do not add a space before the left parenthesis of a function call. This rule makes it easier to identify function calls. This rule makes it easier to identify function calls.

(4) Position to place a new line character for line continuation

When an expression becomes lengthy and extends beyond the length of an easily readable line, a new line character shall be placed at an appropriate position. In placing a new line character, the recommended approach is to apply either one of the following two methods. What is important is to write the continuation line after indenting.

[Method 1] Write an operator at the end of the line.

Example：

```
x = var1 + var2 + var3 + var4 +
    var5 + var6 + var7 + var8 + var9;
if (var1 == var2 &&
    var3 == var4)
```

[Method 2] Write an operator at the beginning of the continuation line.

Example：

```
x = var1 + var2 + var3 + var4
    + var5 + var6 + var7 + var8 + var9;
if (var1 == var2
    && var3 == var4)
```

● Representative styles

(1) K&R style

This is a coding style used in "The C Programming Language" (widely known as K&R)."K&R" used as the acronym of this book derives from the initials of the two authors. In the K&R style, the braces '`{ }`' and indentation are placed in the positions described below:

- **Position of braces:** Place the braces '`{ }`' for function definitions at the beginning of a new line. Place the braces '`{ }`' for others (including structures and control statements, such as, `if`, for and while) on the same line without continuing to a new line (see Example of K&R style).
- **Indentation:** 1 tab. In the first edition of "The C Programming Language", the width of a tab was set to 5 spaces, but in the second edition (ANSI compliant), the number of spaces is set to 4.

(2) BSD style

This is a description style adopted by Eric Allman who wrote many BSD utilities. It is also called the Allman style. In the BSD style, the braces '{ }' and indentation are placed in the positions described below:

- **Position of braces:** Start all the function definitions, `if` , `for` and `while` , etc, from a new line and place the braces '{ }' at the column aligned with the beginning of the previous line (see Example of BSD style).
- **Indentation:** Generally defined as 8 spaces, but 4 is also common.

(3) GNU style

This is a coding style for writing GNU packages. It is defined in "GNU Coding Standards" written by Richard Stallman and volunteers in the GNU project. In the GNU style, the braces '{ }' and indentation are placed in the positions described below:

- **Position of braces:** Start all the function definitions, `if` , `for` and `while` , etc, from a new line. Place the braces '{ }' for function definitions at column 0, and braces '{ }' for others after indenting 2 spaces (see Example of GNU style).
- **Indentation:** 2 spaces. Indent 2 spaces for both the braces '{ }' and their body.

```
(1) Example of K&R style
void func(int arg1)
{ /* Write the { of a function on a
     new line */
   /* Indent is 1 tab */
    if (arg1) {
        …
    }
    …
}
```

```
(2) Example of BSD style
void
func(int arg1)
{ /* Write the { of a function on a
     newline */
    if (arg1)
    {
       …
    }
    …
}
```

```
(3) Example of GNU style
void
func(int arg1)
{ /* Write the { of a function on a
     new line at column 0 */
  if (arg1)
    {
       …
    }
    …
}
```

M4.2 Unify the style of writing comments.

《Convention regarding the style of writing file header comments, function header comments, end of line comments, block comments and copyright shall be defined.》

Preference guide	○
Rule specification	Define

Writing good comments makes the program easier to read. To improve the readability further, a unified style of writing is necessary.

There are document generation tools that create documents for maintenance and examination from the source code. When utilizing such tools, they can be most effectively used by writing in a style that conforms to their specifications. In general, when the explanation of the variables and functions are described according to certain comment conventions, the document generation tools enable these descriptions to be extracted from the source code and reflected in the generated documents. Therefore, it is important to examine the specifications of these tools and define the comment conventions accordingly.

Presented below are some established styles of writing comments that have been extracted from existing coding conventions and related literature.

● Representative styles of writing comments

(1) Indian Hill coding conventions

The following comment rules are described in Indian Hill C Style and Coding Standards:

・Block comments

Comments that describe data structures, algorithms, etc., should be in block comment form with the opening `/` in column 1, a `*` in column 2 before each line of comment text, and the closing `*/` in columns 2-3. (Note that `grep '^.\*'` will catch all block comments in the file.)

Example:

```
/* Write a comment.
 * Write a comment.
 */
```

・Position of comments

- Block comments inside a function

  Should be tabbed over to the same tab setting as the code that they describe.

- End-of-line comments

  Start them apart from the statement by using the tab. If there are more than one of such comments, align them all to the same tab setting.

(2) GNU coding standards

The following comment rules are described in the GNU Coding Standards:

・Language English.

・**Position and contents**

- At the beginning of the program

  Write a comment that briefly explains what the program does at the beginning of every program.

- Function

  Write comments that provide the following information for each function.

  What the function does, explanation of parameters (values, meaning, usage), return value

- `#endif`

  Except for short conditions that are not nested, add comments to explicitly state the conditions at the end of line of every `#endif`.

- Notation for tools

  Place two spaces at the end of each comment sentence.

(3) "The Practices of Programming"

The following comment rules are described in "The Practices of Programming."

・**Position** Describe comments for functions and global data.

・**Other practices**

- Don't belabor the obvious.
- Don't contradict the code.
- Clarify, don't confuse

(4) Others

・Define the policy on when to use `/* */` comment and `//` comment.

Example 1: Use `//` for the comment at the end of statement, and `/* */` for block comment.

Example 2: Use only `//`, because there is a risk of forgetting to close `/* */`.

Example 3: Do not use `/*` or `//` in a comment, provided that `//` may be used in a `//` comment.

・Describe the copyright notice in the comment.

・Define the comment for the `switch` statements without `break`.

Example：

```
switch (status) {
case CASE1:
  Processing;
  /* FALL THROUGH */
case CASE2:
   ・・・
```

・Define the comment for no processing.

Example：

```
if (Condition 1) {
    Processing;
} else if (Condition 2) {
    Processing;
} else {
    /* DO NOTHING */
}
```

・Line-splicing shall not be used in `//` comments. (MISRA C:2012 R3.2)

M4.3 Unify the naming conventions.

《Convention for naming external variables and internal variables shall be defined.》

Preference guide	○
Rule specification	Define

See ☆ **Rules on naming conventions** below.

[**Related rules**] M1.7.1, M1.7.2, M1.7.3, M4.3.2, P1.1.2,P1.2.1

《Convention for naming files shall be defined.》

Preference guide	○
Rule specification	Define

See ☆ **Rules on naming conventions** below.

[**Related rules**] M4.3.1, P1.2.1

☆Rules on naming conventions

Readability of programs is greatly affected by naming. There are various methods for naming but important points are to be consistent and to make the names easy to understand.
For naming, the following items shall be defined:

- Guidelines for names in general
- How to name files (including folders and directories)
- How to name globally and locally
- How to name macros, etc.

Presented in the following are some naming guidelines and rules introduced in existing coding conventions and related literature. They are useful as reference when creating a project-specific naming convention newly. If no naming convention is explicitly defined in the existing project, the recommendation would be to create a naming convention that is closest to the current source code.

● Typical naming conventions

(1) Indian Hill coding conventions

- Names with leading and trailing underscores are reserved for system purposes and should not be used.
- `#define` constants should be in all CAPS.

- `enum` constants should either have the initial character or all the characters capitalized.
- It is best to avoid names that differ only in case, like `foo` and `Foo`.
- Global names should have a common prefix identifying the module that they belong with.
- A file name should be eight characters or less (excluding the extension), starting with an alphabetic character and followed by alphanumeric characters.
- File names that are the same as library header filenames should be avoided.

Overall		• Names with leading and trailing underscores should not be used. • Avoid names that differ only in case. Example: `foo` and `Foo`
Variable and function names	**Global**	A prefix identifying the module should be added.
	Local	Nothing in particular
Other		• Macro names should be in all CAPS. Example: `#define MACRO` • `enum` constants are Capitalized or in all CAPS.

(2) GNU coding standards

- Don't choose terse names—instead, look for names that give useful information about the meaning of the variable or function. Names should be English.
- Use underscores to separate words in a name.
- Stick to lower case; reserve upper case for macros and `enum` constants, and for name-prefixes that follow a uniform convention.

Overall		• Use underscores to separate words in a name. Example: `get_name` • Stick to lower case; reserve upper case for macros and `enum` constants, and name-prefixes that follows a uniform convention.
Variable and function names	**Global**	Don't choose terse names—instead, look for names that give useful information about the meaning of the variable and function. Names should be English.
	Local	Nothing in particular
Other		• Reserve upper case for macros. Example: `#define MACRO` • Reserve upper case for `enum` constants.

(3) "The Practice of Programming"

- Use descriptive names for globals, short names for locals.
- Give related things and related names that show their relationship and highlight their difference.
- Function names should be based on active verbs, perhaps followed by nouns.

Overall		Give related things related names that show their relationship.
Variable and function names	**Global**	Use descriptive names.
	Local	Use short names.
Other		Function names should be based on active verbs, perhaps followed by nouns.

(4) Others

• Identifiers will not differ by:
- The presence/absence of the underscore character
- The interchange of the letter 'O', with the digit '0' or the letter 'D'
- The interchange of the letter 'I', with the letter 'l' (el) or the digit '1'
- The interchange of the letter 'S' with the digit '5'
- The interchange of the letter 'Z' with the digit '2'
- The interchange of the letter 'n' with the letter 'h'.

• **How to separate a name:** A name that consists of multiple words should be either separated with underscore or delimited using an uppercase letter for the first letters of the words. Determine which style to adopt.

• **Hungarian notation:** There is a notation called Hungarian notation that explicitly indicates the type of variable.

• **How to name files:** Give a name with a prefix, for example, that expresses the subsystem.

[**Related rules**] P1.1.2, P1.2.1

M4.4 Unify the contents to be described in a file and the order of describing them.

《The descriptive contents of header files (declarations, definitions, etc) and the order they are described in shall be defined.》

Preference guide	○
Rule specification	Define

Items commonly used in a program shall be described in header files to prevent the risk of modification errors when they are scattered in different places. Header files should contain macro definitions, tag declarations for structures, unions and enumeration types, typedef declarations, external variable declarations and function prototype declarations that are commonly used in multiple source files.
For example, they should be described in the following order:

(1) File header comment
(2) Inclusion of system headers
(3) Inclusion of user defined headers
(4) `#define` macros
(5) `#define` function macros
(6) `typedef` definitions (type definitions for basic types such as `int` or `char`)
(7) `enum` tag definitions (together with `typedef`)
(8) `struct/union` tag definitions (together with `typedef`)
(9) `extern` variable declarations
(10) Function prototype declarations
(11) Inline function

《The descriptive contents of source files (declarations, definitions, etc) and the order they are described in shall be defined.》

Preference guide	○
Rule specification	Define

In a source file, definitions of variables and functions, definitions or declarations of macros, tags, and types (`typedef` types) used only in the individual source file should be described.
For example, they should be described in the following order:

(1) File header comment
(2) Inclusion of system headers
(3) Inclusion of user-defined headers
(4) `#define` macros used only in this file
(5) `#define` function macros used only in this file
(6) `typedef` definitions used only in this file
(7) `enum` tag definitions used only in this file
(8) `struct/union` tag definitions used only in this file
(9) static variable declarations shared in this file
(10) static function declarations
(11) Variable definitions
(12) Function definitions

*Regarding (2) and (3), be careful not to include unnecessary items.
*Avoid describing (4) through (8) as much as possible.

M4.4.3 To use or define external variables or functions (except for functions used only in the file), the header file describing their declarations shall be included.

Preference guide	○
Rule specification	

Compliant example

```
--- my_inc.h ---
extern int   x;
int   func(int);

--------------
#include "my_inc.h"
int x;
int func(int in)
{
  …
```

Non-compliant example

```
/* Declaration of variable x and function func
   are missing */
int x;
int func(int in)
{
   …
```

In C language, variables must either be declared or defined before being used. On the other hand, functions can be used without declarations or definitions. However, to ensure that declarations and definitions are consistent, the declarations should be described in the header file, and that header file should be included.

M4.4.4 External variables shall not be defined in multiple locations.

Preference guide	●
Rule specification	

Compliant example

```
int  x;  /* Definition of one external variable
            shall be only once */
```

Non-compliant example

```
int  x;
int  x;  /* Definition of an external variable
            in multiple locations does not cause
            a compile error */
```

Definitions without initialization for external variables can be described more than once. However, the behavior is not guaranteed when an external variable is initialized in multiple files.

M4.4.5 Variable definitions or function definitions shall not be described in a header file.

Preference guide	○
Rule specification	

Compliant example

```
--- file1.h ---
extern int  x;    /* Variable declaration */
int  func(void);  /* Function declaration */

--- file1.c ---
#include   "file1.h"
int  x;           /* Variable definition */
int  func(void)   /* Function definition */
{
  …
}
```

Non-compliant example

```
--- file1.h ---
int  x;  /* External variable definition */
static int  func(void) /* Function definition */
{
   …
}
```

Header files might be included into several source files. Therfore, describing variable definitions and function definitions in a header file may unnecessarily enlarge object code size generated after compilation. Basically, only declarations or type definitions should be described in a header file.

M4.4.6 Header files shall be descriptively capable of handling redundant inclusions.《The descriptive method to achieve this capability shall be defined.》

Preference guide	○
Rule specification	Define

Compliant example

```
--- myheader.h ---
#ifndef MYHEADER_H
#define MYHEADER_H
    Contents of the header file
#endif     /* MYHEADER_H */
```

Non-compliant example

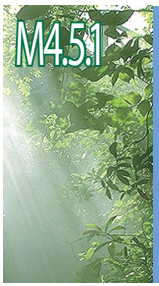
```
--- myheader.h ---
void func(void);
/* end of file */
```

The descriptive contents of header files should be organized to avoid redundant inclusions. However, there are cases when redundant inclusions become unavoidable. To prepare for such cases, header files should be written in such a way that will make them possible of handling multiple inclusions.
As an example, the following may be defined as the rule for writing header files that are capable of handling redundant inclusions:

Example of the rule:

`#ifndef` macro that judges whether the header has already been included or not shall be written at the beginning of the header file, so that the descriptions that follow will not be compiled in subsequent inclusions. In this case, the macro name should be the same as the header file name but replacing all the lowercase letters to uppercase letters, and the period '`.`' to underscore '`_`'.

M4.5 Unify the style of writing declarations.



(1) In a function prototype declaration, all the parameters shall not be named (types only.)
(2) In a function prototype declaration, all the parameters shall be named. In addition, the types of the parameters, their names and the type of the return value shall be literally the same as those of the function definition.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
int func1(int, int);

int func1(int x, int y)
{
   /* Process the function */
}

Compliant example of (2)
int func1(int x, int y);
int func2(float x, int y);

int func1(int x, int y)
{
   /* Process the function */
}

int func2(float x, int y)
{
   /* Process the function */
}
```

Non-compliant example

```
Non-compliant example of (1) and (2)
int func1(int x, int y);
int func2(float x, int y);

int func1(int y, int x)  /* The parameter
                            name differs from
                            the name in the
                            prototype
                            declaration */
{
   /* Process the function */
}

typedef  int INT;
int func2(float x, INT y) /* The type of y is
                             not literally the
                             same as in the
                             prototype
                             declaration */
{
   /* Process the function */
}
```

In a function prototype declaration, parameter names can be omitted, but describing appropriate parameter names is useful as function interface information. When describing parameter names, use the same name as in the definition to avoid unnecessary confusion. As for the parameter type name, making it literally the same as the function definition is also recommended to make the code easier to read.
Moreover, if the parameter is an array of specific size, it is desirable to specify the number of its elements.

Example: `void func(int a[4]) {…}`

[**Related rule**] M1.4.1

M4.5.2 Structure tags and variables shall be declared separately.

Preference guide	
Rule specification	

Compliant example

```
struct TAG  {
 int mem1;
 int mem2;
};
struct TAG x;
```

Non-compliant example

```
struct TAG  {
 int mem1;
 int mem2;
} x;
```

M4.5.3 (1) “,” shall not be placed before the last “}” in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators.

Preference guide	
Rule specification	Choose

(2) “,” shall not be placed before the last “}” in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. However, placing “,” before the last “}” in the list of initial values for array initialization is allowed.

Compliant example

```
Compliant example of (1)
struct tag  data[] = {
  { 1, 2, 3 },
  { 4, 5, 6 },
  { 7, 8, 9 }    /* There is no comma after the
                    last element */
};
Compliant example of (2)
struct tag  data[] = {
  { 1, 2, 3 },
  { 4, 5, 6 },
  { 7, 8, 9 },    /* There is a comma after the
                     last element */
};
```

Non-compliant example

```
Non-compliant example of (1) and (2)
struct tag  x = { 1, 2, };
/* Not clear whether there are only two members
   or there are three or more */
```

The usage of comma in descriptions for initializing multiple data is generally divided into two schools of coding rules. One school follows the tradition of not placing a comma after the last initial value in order to indicate the end of initialization explicitly. Another school follows the tradition of placing a comma at the end by considering the easiness of adding or deleting initial values. Decide on which rule to follow by weighing the importance of the usage of comma for such descriptions in your specific cases.
In C90 standard, it was not acceptable to have “,” just before “}” that indicates the end of an enumerator list, but this became acceptable in C99 standard.

[**Related rule**] M2.1.1

M4.6 Unify the style of writing null pointers.

(1) `0` shall be used for the null pointer. `NULL` shall not be used in any case.
(2) `NULL` shall be used for the null pointer. `NULL` shall not be used for anything other than the null pointer.

Preference guide	○
Rule specification	Define

Compliant example

```
Compliant example of (1)
char *p;
int dat[10];

p = 0;
dat[0] = 0;

Compliant example of (2)
char *p;
int dat[10];

p = NULL;
dat[0] = 0;
```

Non-compliant example

```
Non-compliant example of (1)
char *p;
int dat[10];

p = NULL;
dat[0] = NULL;

Non-compliant example of (2)
char *p;
int dat[10];

p = 0;
dat[0] = NULL;
```

`NULL` has been conventionally used to express the null pointer, but the expression of the null pointer varies, depending on the execution environment. For this reason, some people think that it is safer to use `0`.

M4.7 Unify the style of writing preprocessor directives.

The body and parameters of a macro that includes operators shall be enclosed with parentheses ( ).

Preference guide	○
Rule specification	

Compliant example

```
#define M_SAMPLE(a, b) ((a)+(b))
```

Non-compliant example

```
#define M_SAMPLE(a, b) a+b
```

If the body and parameters of a macro are not enclosed with parentheses ( ) , there is a risk of bug being produced when the operations are not performed in the expected order, since the operation order depends on the order of precedence of operators that come next to the macro after expanding the macro and the operators in the macro.

`#else`, `#elif` or `#endif` that corresponds to `#ifdef`, `#ifndef` or `#if` shall be described in the same file, and《their correspondence relationship shall be clearly stated with a comment defined in the project》.

Preference guide	○
Rule specification	Define

Compliant example

```
#ifdef  AAA
/* Process when AAA is defined */
  …
#else       /* not AAA */
/* Process when AAA is not defined */
  …
#endif      /* end AAA */
```

Non-compliant example

```
#ifdef  AAA
/* Process when AAA is defined */
  …
#else
/* Process when AAA is not defined */
  …
#endif
```

If `#else` or `#endif` is described in a distant location or nested in a partitioned process by macros, such as, `#ifdef` , their correspondence becomes difficult to understand. Therefore, add a project-defined comment to `#else` or `#endif` that corresponds with, such as, `#ifdef` to make their correspondence easier to understand.

[**Related rules**] M1.1.1, M1.1.2

M4.7.3 `defined(`*macro_name*`)` or `defined` *macro_name* shall be used to check whether the macro name has already been defined by `#if` or `#elif`.

Preference guide	●
Rule specification	

Compliant example

```
#if  defined(AAA)
…
#endif
- or -
#if defined AAA
…
#endif
```

Non-compliant example

```
#if  AAA
  …
#endif
- or -
#defined DD(x) defined(x)
#if  DD(AAA)
  …
#endif
```

"`#if` macro name" does not determine whether a macro is defined or not. For example, when `#if AAA` is written, it will be evaluated as `false` not only when macro `AAA` is not defined, but also when the value of macro `AAA` is 0. The C language standard does not define how to process `defined` operator. Therefore, to check whether a macro is `defined` or not, `defined` operator should be used.
`defined` operator should not be described other than by `defined(`*macro_name*`)` or `defined` *macro_name*, because they are the only two ways of describing `defined` that are supported in C language standard, and any other descriptions of `defined` may cause an error or may be interpreted differently, depending on the compiler used.

[**Related rule**] M4.7.7

M4.7.5 Macros shall not be `#define`'d or `#undef`'d within a block. [MISRA C:2004 19.5]

Preference guide	
Rule specification	

Compliant example

```
#define   AAA    0
#define   BBB    1
#define   CCC    2
struct stag {
  int     mem1;
  char   *mem2;
};
```

Non-compliant example

```
/* Members with restriction on the assignable
   values exist */
struct stag {
  int mem1;  /* The following values are
                assignable: */
#define  AAA   0
#define  BBB   1
#define  CCC   2
  char *mem2;
};
```

In general, macro definitions (#define) are all described together at the beginning of the file. If they are scattered in various parts of the file, for example, by describing them in blocks, they will become difficult to read. Moreover, cancellation of definitions (#undef) within a block will also degrade the readability. Also note that, unlike the scope of variables, macro definitions are valid only up to the end of the file. The description below shows how the program in the above non-compliant example can be rewritten to make it compliant:

```
enum etag { AAA, BBB, CCC };
struct stag {
    enum etag    mem1;
    char        *mem2;
};
```

[**Related rule**] M4.7.6

#undef shall not be used.【MISRA C:2012 R19.6】

Preference guide	
Rule specification	

#undef can change the state of #define'd macro name to undefined. But the use of #undef involves the risk of degrading the readability, because the interpretation of #undef may differ, depending on where the macro name is referred to.

[**Related rule**] M4.7.5

The controlling expression of a `#if` or `#elif` preprocessing directive shall be evaluated to 0 or 1. 【MISRA C:2012 R20.8】

Preference guide	
Rule specification	

Compliant example

```
#define TRUE  1
#define FALSE 0
#if TRUE
  …
#if defined(AAA)
  …
#if VERSION == 2
  …
#if 0  /* Invalidated due to ～ */
 …
```

Non-compliant example

```
#define ABC 2
#if ABC
  …
```

In case of `#if` or `#elif` controlling expression, `true` or `false` is evaluated by the controlling expression. Therefore, the controlling expression should be described in a way that would make it easy to evaluate `true` or `false`, thus making the program easy to read.

[**Related rule**] M4.7.3

One of the essential tasks in embedded software development is to check the behaviors (through testing). However, with recent complex embedded software, it is becoming increasingly challenging to fulfill this task when faced with difficulties caused by, such as, bugs and malfunctions detected during tests that cannot be reproduced. Therefore, when writing the source code, it is desirable to be more conscious of writing in a style that will make the root cause analysis easy to perform when problem arises. Moreover, particular attention must also be given to descriptions that involve, such as, the use of dynamic memory, by keeping in mind the risk of memory leak, among other points of concern.

Maintaiability 5.1 **Write in a style that makes it easy to investigate the causes of problems when they occur.**

Maintaiability 5.2 **Be careful when using dynamic memory allocations.**

M5.1 Write in a style that makes it easy to investigate the causes of problems when they occur.

《The rules for writing the code for setting debug options and for recording logs in release modules shall be defined.》

Preference guide	○
Rule specification	Define

Besides implementing the specified functionalities correctly, a good program requires coding that also takes account of the easiness to debug and investigate into the causes of problems when they occur. Descriptions that make investigation of problems easy to conduct can be achieved by writing descriptions for debugging that are not reflected in the release modules and descriptions for outputting logs after release that are reflected in the release modules. Explained below are the points to take into consideration when determining the rules to be followed in writing each of these descriptions.

● Descriptions for debugging

Descriptions for debugging, including print statements used during program development, need to be written as isolated descriptions that are not reflected in the release module. Explained below are two ways of writing the descriptions for debugging: (a) by isolating the debug descriptions using macro definitions; and (b) by using `assert` macros for debugging purpose.

(a) Using macro definitions to isolate debug descriptions

Use the macro definitions to identify the code parts to be compiled so that the debug descriptions are not reflected in the provided release module. Strings, such as, "`DEBUG`" and "`MODULEA_DEBUG`" that contain "`DEBUG` " as part of the name are commonly used as those macro names.

Example of rule definition:
`#ifdef DEBUG` shall be used to isolate the debug code. (`DEBUG` macro shall be specified at compile time.)

```
[Code example]
     #ifdef DEBUG
     fprintf(stderr, "var1 = %d/n", var1);
     #endif
```

The following macro definitions can also be used.

Example of rule definition:
`#ifdef DEBUG` shall be used to isolate the debug code. (`DEBUG` macro shall be specified at compile time). In addition, the following macro shall be used to output debug information.

```
     DEBUG_PRINT(str);   /* Output str to standard output */
```

Since this macro is defined in the common header of the project, `debug_macros.h` , this header shall be
included when using this macro.

```
-- debug_macros.h --
 #ifdef DEBUG
 #define DEBUG_PRINT(str)    fputs(str, stderr)
 #else
 #define DEBUG_PRINT(str)    ((void) 0)  /* no action */
 #endif /* DEBUG */

[Code example]
void func(void) {
   DEBUG_PRINT(">> func\n");
   • • •
   DEBUG_PRINT("<< func\n");
}
```

(b) Using the assert macro

In C language standard, `assert` macro is provided as a macro for program diagnosis. It is useful for making coding errors easier to detect during debugging. To facilitate debugging, define where to use the `assert` macro and follow this defined usage throughout the project. By doing so, it will be possible to collect consistent debug information during, such as, the integration test, and such information, as a result, will help make debugging easier.

The following is a brief explanation on how to use the `assert` macro, using a coding example that shows how this macro is used in a function definition written under the precondition that the null pointer is never passed as the argument.

```
void func(int *p) {
    assert(p != NULL);
    *p = INIT_DATA;
    • • •
}
```

If the NDEBUG macro is defined at compile time, the `assert` macro does nothing. On the other hand, if the NDEBUG macro is not defined and the expression passed to the `assert` macro is `false`, the program abends after outputting the file name and the line number of the source to the standard error. Note that the macro name is NDEBUG, not DEBUG.

`assert` macro is a macro provided by the compiler in `<assert.h>` . By using the following example as a reference, examine how to abort the program and determine whether to use the macro provided by the compiler or to provide your own `assert` function.

```
#ifdef NDEBUG
#define    assert(exp)    ((void) 0)
#else
#define    assert(exp)    (void) ((exp)) || (_assert(#exp, __FILE__, __LINE__)))
#endif
void _assert(char *mes, char *fname, unsigned int lno) {
    fprintf(stderr, "Assert:%s:%s(%d)¥n", mes, fname, lno);
    fflush(stderr);
    abort();
}
```

C11 allows the offset of struct member and string constant size to be checked at compile time by writing static `assert` that can be evaluated at compile time in the source code.

```
_Static_assert( sizeof(t) <= 4, "The size of t is exceeding 4 bytes.");
```

● Outputting logs after release

It is also useful to include descriptions for problem investigation in the release module that dose not contain descriptions for debug. One common method is to record the result of the investigation as log information. Log information is helpful for validation testing of the release module as well as for investigation of problems that occurred in the system provided to the customer.
In case of recording the log information, the following items should be determined in advance and defined as the coding convention.

- **When to output logs**

  Logs should be output not only when an abnormal condition is detected, but also at the timing of, such as, data communication with an external system. The point is to output logs at appropriate timing (such as, when key events occur) that will make it easier to trace the history and faster to identify the root cause of the detected abnormality.

- **What to output in logs**

  Information on the process executed immediately before the occurrence of the abnormal condition, the data values processed at that time, and information for tracing memory usage are some of the log information that should be recorded to enhance the traceability of the history and facilitate the investigation of the cause of the abnormality.

- **Macro or function for outputting log information**

  Localize the log information output as a macro or a function. It is often preferable to make the log output destination changeable.

(1) The # and ## preprocessor operators should not be used.【MISRA C:2012 R20.10】
(2) A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.【MISRA C:2012 R20.11】

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (2)
#define AAA(a, b)  a#b
#define BBB(x, y)  x##y
```

Non-compliant example

```
Non-compliant example of (1) and (2)
#define  XXX(a, b, c)     a#b##c
```

The evaluation order of # operator and ## operator is not defined. Therefore, # and ## operators should not be mixed, nor used twice or more.

Function shall be used rather than using function-like macro.

Preference guide	
Rule specification	

Compliant example

```
int func(int arg1, int arg2)
{
  return arg1 + arg2;
}
```

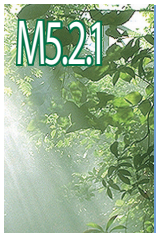
Non-compliant example

```
#define func(arg1, arg2) (arg1 + arg2)
```

Using functions rather than function-like macros facilitates tracking processes by stopping at the beginning of a function during debugging etc.
In addition, the compiler performs type checking with the functions and helps to detect coding mistakes. The functions can be inline functions. For more information about the performance of inline functions and object code size, see E1.1.1.

[**Related rules**] E1.1.1, P2.1.1

M5.2 Be careful when using dynamic memory allocations.



(1) Dynamic memory shall not be used.
(2) If dynamic memory is used,《the maximum amount of memory that can be used, process to be taken when running out of memory, and debugging procedure shall be defined.》

Preference guide	
Rule specification	Choose Define

Using dynamic memory involves the risk of accessing invalid memory as well as the risk of memory leak that leads to depletion of system resources, which may be caused by forgetting to return the obtained memory space to the system.
Some compilers provide functions for debug shown below. First, check the compiler used. Open source software also has pieces of source code for debug and they are useful as references when creating your own.

Example of rule definition:
To obtain and return dynamic memories, do not use standard functions such as `malloc` or `free`, but `X_MALLOC` and `X_FREE` functions provided by the project should be used. Code for debug should be created by compiling the source with `-DDEBUG` option.

```
-- X_MALLOC.h --
#ifdef DEBUG
void *log_malloc(size_t size, char*, char*);
void log_free(void*);
#define X_MALLOC(size)  log_malloc(size, __FILE__, __LINE__)
#define X_FREE(p)  log_free(p, __FILE__, __LINE__)
#else
#include  <stdlib.h>
#define X_MALLOC(size)  malloc(size)
#define X_FREE(p)  free(p)
#endif
```

[Code example]

```
#include  "X_MALLOC.h"
…
p = X_MALLOC(sizeof(*p) * NUM);
if (p == NULL) {
    return (MEM_NOTHING);
}
…
X_FREE(p);
return (Compliant);
```

● Reference: Problems when using dynamic memory

Described below are problems that tend to occur when dynamic memory is used.

・Buffer overflow

This occurs as a result of referencing or updating areas beyond the range of obtained memory. In particular, when an area outside the range is accidentally updated, this failure does not occur at the time of update but will occur when the memory destroyed by the update is referenced. The problem with dynamic memory is that it is very difficult to locate which part of the memory has been destroyed.

・Forgetting to initialize

When a class area is allocated with new and an appropriate constructor is defined, there is no risk of initialization being forgotten. However, be careful when an area other than class is allocated with new, because initialization will not be performed automatically.

・Memory leak

There is a risk of this problem being caused by forgetting to return the obtained memory space. This problem does not occur with programs that terminate each time after running once. However, with programs that keep running, memory leak can occur and become the cause of memory depletion and system malfunction.

・Use after return

When the obtained memory space is returned due to, such as, delete, the returned memory space may be reused later, such as, when new is called. In case the deleted memory address is used to update the memory, the memory space will be destroyed if it is already being reused for other purpose. As explained in the case with buffer overflow, it is very difficult to locate which part of the memory has been destroyed.

The code that leads to these problems does not cause a compile error. In addition, problems do not occur at the location where the bugs were implanted, making them undetectable in tests that are just for checking the normal specifications. They cannot be discovered unless the code is carefully reviewed or tests are performed after inserting a test code specifically written to detect such problems or after adding a special library to the program for similar purpose.

Portability

One of the distinctive aspects of embedded software is that there are diverse options in the platform used for software operation. This also means that there are many possible combinations of MPU options and OS options to select the hardware and software platforms from. As the number of functionalities realized by the embedded software increases, opportunities to port the existing software to other platforms by modifying or remodeling it to make it compatible with multiple platforms are also on the rise.

Due to this trend, software portability is becoming an extremely important element also at the source code level. In particular, writing in a style that is implementation-dependent is one of the most common mistakes made on a regular basis.

- **Portability 1: Write in a style that is not dependent on the compiler.**
- **Portability 2: Localize the code that has a problem with portability.**

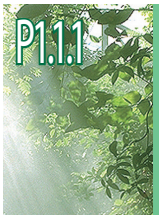
Write in a style that is not dependent on the compiler.

Use of compilers is unavoidable when programming in C++ language. Various compilers are available in the world and each has its own features. If the source code is written poorly, the code may become dependent on the features of the compiler used, and may cause unexpected results when a different compiler is used.

For this reason, programming must be performed carefully with an awareness that the code must be written in a style that is not implementation-dependent.

Portability 1.1	Do not use functionalities that are advanced features or implementation-defined.
Portability 1.2	Use only the characters and escape sequences defined in the language standard.
Portability 1.3	Confirm and document data type representations, behavioral specifications of advanced functionalities and implementation-dependent parts.
Portability 1.4	For source file inclusion, confirm the implementation-dependent parts and write in a style that is not implementation-dependent.
Portability 1.5	Write in a style that does not depend on the environment used for compiling.

P1.1 Do not use functionalities that are advanced features or implementation-defined.



(1) Functionalities not specified in the language standard shall not be used.
(2) If functionalities not specified in the language standard are used,《the functionalities used and their usage shall be documented.》

Preference guide	
Rule specification	Choose Document

At present, while C99 is the widely used C language standard, the latest version is C11. In addition, there are still many compilers that also support the older version, C90.
One way of thinking would be to choose rule (2) and allow the use of functionalities defined in the latest language standard, C11, that are specifically supported by the compiler used.
Regarding the acceptable ways of using the functionalities that are not specified in the language standard, the details are provided in the following related rules.

［**Related rules**］P1.1.3, P1.2.1, P1.2.2, P1.3.2, P2.1.1, P2.1.2

《All usage of implementation-defined behavior shall be documented.》【MISRA C:2004 3.1】

Preference guide	○
Rule specification	Document

In the language standard, there are implementation-defined items whose behavior varies depending on the compiler used. For example, the following are implementation-defined and should be documented if they are used.

- How to represent floating-point numbers
- For C90, how to handle signs of remainders for integer division
- The search order of files for the `#include` directive
- `#pragma`

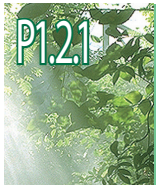
To use a program written in another language, 《its interface shall be documented and its usage shall be defined.》

Preference guide	
Rule specification	Document Define

The C language standard does not define the interface for making programs written in other languages available from a C language program. In other words, using a program written in another language requires the use of an advanced functionality, which means that portability will be impaired. Therefore, when using such a program, document the specifications of the compiler used and define its usage, regardless of the possibility of porting.

［**Related rules**］P1.1.1, P2.1.1

P1.2 Use only the characters and escape sequences defined in the language standard.



To use characters other than those defined in the language standard for writing a program, the compiler specifications shall be confirmed, and 《their usage shall be defined.》

Preference guide	
Rule specification	Define

The basic character set defined in the language standard as usable for source code are upper and lower case letters of the Latin alphabet, decimal digits, graphic characters (_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & ¦ = , ” ’), space character, and control characters that represent the horizontal tab, vertical tab, form feed and new line.
International characters and multibyte characters (like Japanese) can be used as identifiers and characters, but they may not be supported by some compilers. Therefore, if these characters are going to be used, check beforehand that they can be used in the following locations and define their usage.

- Identifiers
- Comments
- String literals
  - Processing when \ exists in the character codes of the string (whether special care is required or whether options are required at compile time etc.)
  - The necessity to write using wide string literals (with the prefix L such as L“*string*”.)

・Character constants
  - The bit length of the character constant
  - The necessity to write using wide character constants (with the prefix L such as L'あ').
・The file name of #include

For example, define the following rules.

・As the identifier, only the alphanumeric characters and underscore should be used.
・Comments can be written in Japanese. The character code used should be Shift_JIS. 1 byte Kana shall not be used.
・Japanese shall not be used in strings, string constants and file names of #include.

［**Related rules**］M4.3.1, M4.3.2, P1.1.1

Only escape sequences defined in the language standard shall be used.

Preference guide	
Rule specification	

Compliant example

```
char c = '\t';   /* compliant */
```

Non-compliant example

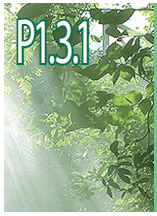
```
char c = '\e'; /* Non-compliant: The escape is
                  sequence not defined in the
                  language standard. It is not
                  portable */
```

The language standard defines the following seven nongraphic characters as escape sequences:

\a (alert)　\b (backspace)　\f (form feed)　\n (new line)
\r (carriage return)　\t (horizontal tab)　\v (vertical tab)

［**Related rule**］P1.1.1

P1.3 Confirm and document data type representations, behavioral specifications of advanced functionalities and implementation-dependent parts.



Simple char type (that does not specify the signedness) shall be used only for storing character values. If a process that depends on signedness (implementation-defined) is required, unsigned char or signed char that specifies its signedness shall be used.

Preference guide	○
Rule specification	

Compliant example

```
char c = 'a';     /* Used to store characters */
int8_t  i8 = -1; /* To use it as 8-bit data,
                    use a type defined, for
                    example, with typedef */
```

Non-compliant example

```
char c = -1;
if (c > 0) { … }
/* Non-compliant: char can be signed or unsigned
   depending on the compiler, and this difference
   affects the result of the comparison. */
```

Unlike other integer types like `int` , `char` that does not specify its signedness will be either signed or unsigned depending on the compiler (`int` type is the same as `signed int` type.) Therefore, using `char` that depends on the signedness is not portable. This is because `char` that does not specify its signedness is an independent type provided for storing characters (comprised of three types: `char` , `unsigned char` and `signed char`) and the language standard assumes such usage. For using `char` as a small integer type, such as, when a process that depends on signedness is required, use either `unsigned char` or `signed char` that specifies its signedness. In this case, it is desirable to use typedef as the type to localize the range of modification during porting.

A problem similar to this rule exists with the returned type of the standard function `getc` that is `int` and must not be received by `char` . However, this is rather a problem pertaining to function interface (assignment that may cause information loss).

［**Reference for those seeking further details**］

"MISRA C:2012" Rule 10.1

［**Related rule**］ P2.1.3

P1.3.2 The members of an enumeration type (enum) shall be defined with values that can be represented as `int` type.

Preference guide	
Rule specification	

Compliant example

```
/* If int is 16bits and long is 32bits */
enum largenum {
  LARGE = INT_MAX
};
```

Non-compliant example

```
/* If int is 16bits and long is 32bits */
enum largenum {
  LARGE = INT_MAX+1
};
```

In the C language standard, the members of an enumeration type must have values that can be represented as `int` type. However, some compilers that support this functionality extendedly may not cause an error even if the value exceeds the range of `int` type for the members of an enumeration type.

Reference C++ allows values in the range of `long` type for the members of an enumeration type.

［**Related rule**］P1.1.1

P1.3.3 (1) Bit fields shall not be used. (2) Bit fields shall not be used for data whose bit positions are meaningful. (3)《If it is being relied upon, the implementation-defined behavior and packing of bit fields shall be documented.》［MISRA C:2004 3.5］

Preference guide	○
Rule specification	Choose Document

Compliant example

```
Compliant example of (2)
struct S {
   unsigned int bit1:1;
   unsigned int bit2:1;
};
extern struct S * p;    /* Compliant if p points
                           to a date that is, for
                           example, just a set of
                           flags and bit1 can be
                          any bit in that data */

p->bit1 = 1;
```

Non-compliant example

```
Non-compliant example of (2)
struct S {
   unsigned int bit1:1;
   unsigned int bit2:1;
};
extern struct S * p;
/* If the bit positions are meaningful, for
   example, when p points at IO ports; in
   other words, if there is a meaning for bit1
   to point at either the lowest bit or the
   highest bit of the data, the program is non-
   portable */

p->bit1 = 1;    /* As to which bit of the data,
                   p will point at, is
                   implementation-dependent */
```

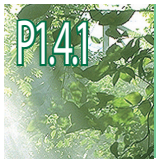
The following behaviors of bit field vary depending on the compiler used:

(1) Whether the bit field of an `int` type that does not specify its signedness will be handled as signed ;
(2) Assignment order of the bit fields within a unit
(3) Boundary of a bit field in a storage unit

If bit fields are used to access data whose bit positions are meaningful, such as, the IO ports, portability problem arises due to rules (2) and (3). Therefore, in such cases, do not use bit fields, but instead, use bitwise operations, such as, & and | .

［**Related rule**］ R2.6.1

P1.4 For source file inclusion, confirm the implementation-dependent parts and write in a style that is not implementation-dependent.



The `#include` directive shall be followed by either a <filename> or “filename” sequence. 【MISRA C:2012 R20.3】

Preference guide	●
Rule specification	

Compliant example

```
#include <stdio.h>
#include  "myheader.h"
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
#endif
#include INCFILE
```

Non-compliant example

```
#include  stdio.h
   /* Neither <> nor “ ” is placed */
#include  "myheader.h"  1
   /* 1 is specified at the end */
```

In C language standard, the behavior is not defined for cases where the format of the header name does not match with neither of the two styles (< > or “ ”) after macro-expansion of the #include directive. Most compilers will output an error if it cannot match the format with neither of the two styles, while some others may not handle it as an error. Therefore, write the header name format in either of the two styles to ensure safety.

P1.4.2 《The usage of <> format and "" format for #include file specification shall be defined.》

Preference guide	
Rule specification	Define

Compliant example

```
#include <stdio.h>
#include "myheader.h"
```

Non-compliant example

```
#include "stdio.h"
#include <myheader.h>
```

There are two ways of writing #include . To unify the writing style, define rules, for example, that include the following:

- Specify the header provided by the compiler by enclosing it with < > ;
- Specify the header created in the project by enclosing it with " " ;
- Specify the header provided by the purchased software by enclosing it with " " .

P1.4.3 Characters ', \, ", /*, // and : shall not be used for file specification in #include.

Preference guide	○
Rule specification	

Compliant example

```
#include "inc/my_header.h"  /* compliant */
```

Non-compliant example

```
#include "inc\my_header.h"  /* Non-compliant */
```

The language standard does not define the behavior when the characters mentioned above are used (more specifically, in the following cases); that is to say, the operation result is not certain when these characters are used in the following cases, which consequently make the code non-portable:.

- When characters ' \ " or /* appear in the string enclosed with < > ;
- When characters ' \ " or /* appear in the string enclosed with " " .

Also, the behavior of the character : (colon) differs depending on the compiler, and makes the code nonportable.

P1.5 Write in a style that does not depend on the environment used for compiling.

The absolute path shall not be written for `#include` file specification.

Preference guide	
Rule specification	

Compliant example

```
#include "h1.h"
```

Non-compliant example

```
#include "/project1/module1/h1.h"
```

If an absolute path is written in the code, there will be a need to modify the path when the program is compiled after changing the directories.

Portability 2

Localize the code that has a problem with portability.

The principle is not to write implementation-dependent source code as much as possible. But in some cases, this may be unavoidable. A typical example is when an assembly language program is called from C language. In such case, it is recommended to localize the implementation-dependent parts of the code as much as possible.

Portability 2.1 Localize the code that has a problem with portability.

P2.1 Localize the code that has a problem with portability.

When assembly language programs are called from C language,《how to localize such parts shall be defined》,such as, by expressing them as functions of C language that contain only inline assembly language code or describing them using macros.

Preference guide	○
Rule specification	Define

Compliant example

```
#define SET_PORT1  asm("   st.b 1, port1")
void f() {
  …
  SET_PORT1;
  …
}
```

Non-compliant example

```
void f() {
  …
  asm("  st.b 1,port1");
  …
}
/* asm and other processes are mixed */
```

In C99, inline-specified functions can be written. Many compilers provide extended support to asm (*string*) format as a method to include the assembly language code. However, there are some compilers that do not provide such support. Moreover, the same format may lead to different behavior depending on the compiler used, thus making it non-portable.

［**Related rules**］M5.1.3, P1.1.1, P1.1.3, E1.1.1

Keywords extended by the compiler shall be used by localizing them after《defining the macros.》.

Preference guide	○
Rule specification	Define

Compliant example

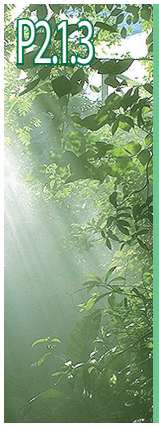
```
/* interrupt is defined as a keyword extended by
   a specific compiler. */
#define INTERRUPT interrupt
INTERRUPT  void int_handler (void) {
  …
}
```

Non-compliant example

```
/* interrupt is defined as a keyword extended
   by a specific compiler. It is used without
   being defined by a macro */
interrupt  void int_handler(void) {
  …
}
```

Some compilers provide extended keywords instead of using the `#pragma` directive, But the code will become non-portable when these keywords are used. Therefore, when using them, localize them, such as, by defining them as macros. For the macro name, use the keyword written in uppercase letters, as shown above in the compliant example.

［**Related rule**］ P1.1.1



(1) The basic types (`char`, `int`, `long`, `long long`, `float`, `double` and `long double`) shall not be used. Instead, the types defined by `typedef` shall be used.《The types defined by `typedef` that are used in the project shall be defined.》

(2) When using any of the basic types (`char`, `int`, `long`, `long long`, `float`, `double` and `long double`) in a form that is dependent on its size, the type defined by `typedef` for each of these basic types shall be used.《The types defined by `typedef` that are used in the project shall be defined.》

Preference guide	○
Rule specification	Choose Document

Compliant example

```
Compliant example of (1) and (2)
uint32_t flag32;   /* Use unit32_t if 32bits is
                                    assumed */

Compliant example for (2)
int i;
for (i = 0; i < 10; i++) {  … }
/* i is used as an index. It can be 8bits,
   16bits or 32bits and a basic type in the
   language specification can be used for i */
```

Non-compliant example

```
Non-compliant example of (1) and (2)
unsigned int flag32;  /* used by assuming int as
                                    32bits */
```

The size and internal representation of integer types and floating point types vary depending on the compiler. C99 specifies the following `typedef`s to be provided as the language standard. Therefore, these type definitions should be used.

`int8_t  int16_t  int32_t  int64_t  uint8_t  uint16_t  uint32_t  uint64_t`

When using a compiler that supports C90, it is advisable to refer to them as the typedef names for these basic types.

［**Related rule**］ P1.3.1

Efficiency

Embedded software is characteristic for being embedded in a product and operating together with hardware to serve its purposes in the real world. The increasing demand for further product cost reduction has imposed various restrictions, not only on, such as, MPU or memory, but also on software.

In addition, requirements, such as, on real-time property have placed stricter time constraints that need to be met. Embedded software must therefore be coded with particular attention on resource efficiency like efficient use of memory and time efficiency that takes account of time performance.

- **Efficiency1: Write in a style that takes account of resource and time efficiencies.**

Depending on how the source code is written, the object size may increase and the execution speed may slow down. If there are restrictions on memory size and processing time, the code must be written thoughtfully with additional considerations given to these restrictions.

Efficiency 1.1 Write in a style that takes account of resource and time efficiencies.

E1.1 Write in a style that takes account of resource and time efficiencies.

E1.1.1 Macro functions shall be used only in parts related to speed performance.

Preference guide	●
Rule specification	

Compliant example

```
extern void func1(int,int); /* func1: called
                               only once */
#define func2(arg1, arg2)   /*  func2: called
                               many times */

func1(arg1, arg2);
for  (i = 0; i < 10000; i++) {
  func2(arg1, arg2);  /* Speed performance is
                         critical for this
                         process. */
}
```

Non-compliant example

```
#define func1(arg1, arg2)   /* func1: called only
                               once */
extern void func2(int, int); /* func2: called
                                many times */

func1(arg1, arg2);
for (i = 0; i < 10000; i++) {
  func2(arg1, arg2);  /* Speed performance is
                         critical for this
                         process. */
}
```

Function is safer than macro function. So, use function as much as possible (see M5.1.3). However, processing of function calls and returns may slow down the speed performance. The use of inline function can be one way of preventing the processsing speed from slowing down. But since inlining is implementation-dependent, inline function may not be expanded as intended, depending on the complier used. Therefore, if speed performance is an issue that has to be improved, use macro function instead.

Yet, the frequent use of macro function may increase the object size because the code will be spread to wherever the macro function is used.

［**Related rule**］M5.1.3

E1.1.2 Operations that remain unchanged shall not be performed within an iterated process.

Preference guide	●
Rule specification	

Compliant example

```
var1 = func();
for (i = 0; (i + var1) < MAX; i++) {
  …
}
```

Non-compliant example

```
/* Function func returns the same result */
for (i = 0; (i + func()) < MAX; i++) {
  …
}
```

Repeating the same process that returns the same result is inefficient. Although optimization of the compiler is often reliable for preventing such inefficiency, attention is still necessary in some cases, like in the non-compliant example shown above, where the compiler used cannot determine the invariance..

Instead of structures, pointers to structures shall be used as function parameters.

Preference guide	
Rule specification	

Compliant example

```
typedef struct stag {
  int mem1;
  int mem2;
    …
}  STAG;
int func (const STAG *p) {
  return p->mem1 + p->mem2;
}
```

Non-compliant example

```
typedef struct stag {
  int mem1;
  int mem2;
    …
}  STAG;
int func (STAG x) {
  return x.mem1 + x.mem2;
}
```

If a structure is passed as a function argument, all the structure data are copied into the area for storing arguments when the function is called. If the size of the structure is large, it will become the cause of speed performance degradation.
If a structure passed is read-only, not only pass it as a pointer to the structure, but also qualify it with `const`.

《The policy of selecting either `switch` or `if` statement shall be determined and defined by taking readability and efficiency into consideration.》

Preference guide	
Rule specification	Define

`switch` statements often provide higher readability than `if` statements. In addition, recent compilers tend to output optimized code using, such as, table jump or binary search when they process `switch` statements. Take these matters into consideration when defining this rule.

Example of the rule:
`switch` statement shall be used instead of `if` statement when:
- a process branches according to the value of the expression (integer value), and
- the number of branches is three or more.

However, this rule shall not apply if:
- using the `switch` statement causes an efficiency issue that impedes the improvement of program performance.

Part 3

Typical Coding Errors in Embedded Software

Typical Coding Errors in Embedded Software

This section focuses on showing some typical examples of coding errors that are easily made, not only by C language beginners, but even by skilled programmers as well. Some recent compilers provide enhanced warning functions as options, and some of the examples taken up here can be captured by means of compiler warnings or static analysis tools. However, by being careful not to make such coding errors during the coding stage, the amount of corrective work in later processes can be reduced.

Some existing coding conventions provide rules for preventing such coding errors. The members of software development projects or organizations are recommended to examine whether to include such rules into their coding conventions or not, by taking account of the skill levels of those involved in the development of software programs.

In this section, the following six error-prone points are highlighted and explained with examples.

- Meaningless expressions and statements
- Wrong expressions and statements
- Wrong memory usage
- Errors due to misunderstanding in logical operations
- Mistakes due to typos
- Wrong descriptions that do not cause errors in some compilers

1 Meaningless expressions and statements

Leaving statements or expressions that are not executed in the source code is likely to create misunderstanding that often leads to problems as a result. It is said that confusion tends to be caused especially when the source code is modified by engineers who are not the originator of that particular code.

Example 1: Writing statements that are not executed

```
return ret;
ret = ERROR;
```

This problem is caused either by putting a statement to branch the program control flow (`return`, `continue`, `break`, `goto` statement) into the wrong place, or forgetting to delete unnecessary statements when putting such a branch statement.

Example 2: Writing statements whose execution result is not used

```
void func( ··· ) {
    int    cnt;
    ···
    cnt = 0;
    return;
```

Automatic variables and formal parameters cannot be referenced after the function return. Therefore, if the updated variables are not referenced between the update and the return statement, the update becomes an unnecessary expression (statement). There is a possibility that some operations have been missed or unnecessary statements may have been left undeleted due to slippage during program modification.

Example 3: Writing expressions whose execution result is not used

```
int func( ··· ) {
    int    cnt;
    ···
    return cnt++;
```

The postfix ++ operation updates the value of the variable after it is referenced, so increments as shown in the above example are meaningless. If there is a need to return the incremented value to the caller, the prefix increment must be used.

Example 4: Values passed as arguments are not used

```
int func(int in) {
  in = 0;  /* Overwriting the parameter */
  ···
}
```

Overwriting a parameter without referencing its value, as shown in the above example, means that the value of the argument set by the caller is ignored. This may be a coding error.

2 Wrong expressions and statements

To write proper source code, it must be written according to the grammar of the programming language being used. But even programmers who are familiar with the programming language being used can make careless mistakes. Presented below are some examples of wrong expressions and statements that are often seen.

Example 1: Incorrect range specification

```
if (0 < x < 10)
```

The program shown in the above example appears to be a correct description at first sight. But in C language, such description is not interpreted mathematically and is treated as a conditional expression that always becomes true.

Example 2: Comparing outside the range

```
unsigned char uc;
unsigned int   ui;
   ...
if (uc == 256)
   ...
switch (uc) {
case 256:
   ...
}
if (ui < 0)
   ...
```

The variable is compared with a value beyond the range it can express. `uc` can only express values between 0 and 255. `ui` can never be negative.

Example 3: String comparisons cannot be performed with == operation

```
if (str == "abc")
```

The condition shown in the above example compares addresses, and is not a condition for evaluating whether the string “abc” is equal to the string pointed by str or not.

Example 4: Inconsistency between a function type and `return` statement of the function

```
int func1(int in) {
     if (in < 0) return;  /* No good */
     return in ;
}
int func2(void) {   /* No good */
    ...
    return;
}
```

In the definition of a function that returns a value, all the return statements must describe the value to be returned in `return` *expression* (as function func1.) In addition, the type of function with one or more return statements that respectively do not return a value should be `void` (as function func2.) In C99, the inconsistency between such type of function and `return` statement is detected as an error by the compiler.

3 Wrong memory usage

One of the characteristics of C language is that memory can be handled directly. While this is a very useful feature when creating embedded software, it also often causes incorrect operations and must therefore be used carefully.

Example 1: Reference and update outside the array bounds

```
char var1[N];
...
for (i = 1; i <= N; i++) {/* Accessing outside the array bounds (error) */
     var1[i] = i;
}
var1[-1] = 0;  /* error */
var1[N]  = 0;  /* error */
```

The array index in C language starts with 0 and its maximum value is 1 less than the element count.

Example 2: Passing the address of an automatic variable to the caller mistakenly

```
int *func(tag *p) {
    int  x;
    p->mem = &x; /* The automatic variable memory area is referenced after the
                                                    function return (risky) */
    return &x;  /* The automatic variable memory area is referenced after the
                                                    function return (risky) */
}
...
    tag  y;
    int  *p;
    p = func(&y);
    *p = 10;      /* Destroying invalid memory area */
    *y.mem = 20; /* Destroying invalid memory area */
```

Areas for automatic variables or parameters are freed to the system when the function ends, and may be reused for other purposes. If the address of an automatic variable is specified as a function return value or set in an area that can be referenced by the caller, as shown in the above example, unexpected faults may occur when the area that has been returned to the system is referenced or updated.

The area for compound literal introduced in C99 is freed and may be reused for other purposes when the execution proceeds outside the enclosing block of the compound literal.

```
void f ()
{
    ...
    int   *p;
    {
          p = (int []) {2,  4};
      ...
    }
    x = p[0]; /* The memory area may be referenced after it is freed (risky) */
   ...
}
```

Example 3: Referencing memory after being freed as dynamic memory

```
struct stag { /* A list structure */
  struct stag *next;
  ...
};
struct stag *wkp; /* Pointer to the list structure */
struct stag *top; /* Start pointer to the list structure */
  ...
/* Process to free the list structure by its elements one after another */
/* No good: It will accesses to an already freed area at the third control expression
                                                              in the for statement */
for (wkp = top; wkp != NULL; wkp = wkp-> next) {
  free(wkp);
}
```

Memories obtained with, such as, `malloc` function need to be freed to the system by using `free` function. The areas that have been freed by `free` function must not be referenced because they may be reused by the system.

Example 4: Writing into string literals mistakenly

```
char *s;
s = "abc"; /* The string literal may be in ROM area */
s[0] = 'A'; /* Cannot be written */
```

Depending on the compiler, string literals may be allocated in the `const` area. Programmers must therefore be careful not to overwrite string literals.

Example 5: Specifying copy sizes mistakenly

```
#define A 10
#define B 20
char a[A];
char b[B];
...
memcpy(a, b, sizeof(b));
```

When one array is copied to another, it will corrupt the memory area if the copy is made in the size of the source that is larger than the size of the destination. The best way to copy from one array to another is to use arrays of the same size. Or specifying the size of the destination as the copy size will at least prevent the memory from corrupting.

4 Errors due to misunderstanding in logical expressions

The use of logical operators is relatively error-prone. In situations where they are used, close attention must be given especially to the operation results, since in many cases, they lead to different subsequent processes.

Example 1: Using a logical product mistakenly instead of a logical sum

```
if (x < 0 && x > 10)
```

The above example shows a logical product written mistakenly instead of a logical sum. In C language, conditions must be written carefully because they will not be processed as compile error even if it is not possible to fulfill them.

Example 2: Using a logical sum mistakenly instead of a logical product

```
int i, data[10], end = 0;
for (i = 0; i < 10 || !end; i++) {
     data[i] = Value_assigned; /* risk of corrupting outside the area */
     if (termination_condition) {
          end = 1;
     }
}
```

When a different condition is added as a condition for an iteration statement that sequentially references or updates the array elements to a condition for ensuring that array bounds are not exceeded, these conditions must be specified with a logical product. The logical sum, as shown in the above example, may cause the system to access outside the array bounds.

Example 3: Using a bitwise operation mistakenly instead of a logical operation

```
if (len1 & len2)
```

This is an example showing that bitwise AND operator (**&**) has been written mistakenly instead of a logical product operator (**&&**). The bitwise AND operator does not mean that the conditions are processed to gain a logical product. Make sure that the intention of the program is correctly described.

5 Mistakes due to typos

Some operators in C language like = and == have completely different meaning even though they do not differ that much. When writing these operators, sufficient attention must be given to prevent careless mistakes or typos.

Example 1: Writing = operator instead of == operator

```
if (x = 0)
```

To check whether two values are equal or not, == must be written as the operator instead of =. Rules to prevent such errors caused by typos include "Assignment operators shall not be used in expressions to examine true or false."

There are also reverse cases like a==b; where == operator is written mistakenly instead of = operator. Easy mistake like this must also be carefully avoided.

6 Wrong descriptions that do not cause errors in some compilers

Each compiler has various characteristics of its own. Note that some compilers do not cause compile errors during compilation even if the program contains inappropriate descriptions.

Example 1: Macro with the same name that has multiple definitions

```
/* Depending on where AAA is referenced, what is expanded varies */
#define AAA  100
  a = AAA;   /* 100 is assigned */
#define AAA  10
  b = AAA;  /* 10 is assigned */
```

Macro name defined by #define will not become a compile error in some compilers even when it is redefined without applying #undef beforehand. Macro name that may be processed differently depending on where it is used should be avoided since it has the risk of impairing the readability of the program.

Example 2: Writing into the const area mistakenly

```
void func(const int *p) {
     *p = 0;  /* Writing into the const area (error) */
```

Some compilers do not cause a compile error even if the `const` area is rewritten. Programmers should be careful not to rewrite the `const` area.

Appendices

Appendix A List of practices and rules

[Reliability 1] R1 Initialize areas and use them by taking their sizes into consideration.				
Practice in detail		Rule		Page
R1.1	Use areas after initializing them.	R1.1.1	Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.	31
		R1.1.2	const variables shall be initialized at the time of declaration.	31
R1.2	Describe initializations without excess or deficiency	R1.2.1	Arrays with specified number of elements shall be initialized with values that match the number of the elements.	32
		R1.2.2	Initialization of enumeration type (enum type) members shall be by either: not specifying any constants; specifying all the constants; or specifying only the first member.	32
R1.3	Pay attention to the range of the area pointed by a pointer.	R1.3.1	(1) Integer addition to or subtraction from (including ++ and --) pointers shall not be made; Array format with [] shall be used for references and assignments to the allocated area. (2) Integer addition to or subtraction from (including ++ and --) pointers shall be made only when the pointer points to the array and the result must be pointing within the range of the array.	33
		R1.3.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array. 【MISRA C:2012 R18.2】	34
		R1.3.3	Comparison between pointers shall be used only when the two pointers are both pointing at either the elements in the same array or the members of the same structure.	34
		R1.3.4	The restrict type qualifier shall not be used. 【MISRA C:2012 R8.14】	35

[Reliability 2] R2 Use data by taking their ranges, sizes and internal representations into consideration.				
Practice in detail		Rule		Page
R2.1	Make comparisons that do not depend on internal representations.	R2.1.1	Floating-point expressions shall not be used to perform equality or inequality comparisons.	37
		R2.1.2	Floating-point variable shall not be used as a loop counter.	37
		R2.1.3	memcmp shall not be used to compare structures and unions.	38
R2.2	When values such as logical values are defined as a range, do not make a judgment by finding whether or not a value is equivalent to any particular value (representative value) within this range.	R2.2.1	Comparison with a value defined as true shall not be made in expressions that examine true or false.	39
R2.3	Use the same data type to perform operations or comparisons.	R2.3.1	Unsigned integer constant expressions shall be described within the range that can be represented with the result type.	39
		R2.3.2	When using conditional operator (?: operator), the logical expression shall be enclosed in parentheses ( ) and both return values shall be the same type.	39
		R2.3.3	Loop counters and variables used for comparison of loop iteration conditions shall be the same type.	40
R2.4	Describe code by taking operation precision into consideration.	R2.4.1	When the type of an operation and the type of the destination to which the operation result is assigned (assignment destination) are different, the operation shall be performed after casting them to the type of expected operation precision.	40
		R2.4.2	When performing arithmetic operations or comparisons of expressions mixed with signed and unsigned, an explicit cast to the expected type shall be performed.	41
R2.5	Do not use operations that have the risk of information loss.	R2.5.1	When performing assignments (=operation, actual arguments passing of function calls, function return) or operations to data types that may cause information loss, they shall be first confirmed that there are no problems, and a cast shall be described to explicitly state that they are problem-free.	42

[Reliability 2] R2 Use data by taking their ranges, sizes and internal representations into consideration.				
Practice in detail		Rule		Page
R2.5	Do not use operations that have the risk of information loss.	R2.5.2	Unary operator '-' shall not be used in unsigned expressions.	43
		R2.5.3	When ones' complement (~) or left shift (<<) is applied to unsigned char or unsigned short type data, an explicit cast to the type of the operation result shall be performed.	43
		R2.5.4	The right-hand side of a shift operator shall be zero or more, and less than the bit width of the left-hand side.	44
R2.6	Use types that can represent the target data.	R2.6.1	(1) The types used for bit field shall only be signed int or unsigned int. If a bit field of 1 bit width is required, unsigned int type shall be used, and not the unsigned int type. (2) The types used for bit field shall be signed int, unsigned int or _Bool. If a bit field of 1 bit width is required, unsigned int type or _Bool type shall be used. (3) The types used for bit field shall be signed int, unsigned int, _Bool, or those allowed by the compiler that are either enum or the type that specifies signed or unsigned. If a bit field of 1 bit width is required, the type that specifies unsigned or _Bool type shall be used.	44
		R2.6.2	Data used as bit sequences shall be defined with unsigned type, and not with the signed type.	46
R2.7	Pay attention to pointer types.	R2.7.1	(1) Pointer type shall not be converted to other pointer type or integer type, and vice versa, with the exception of mutual conversion between "pointer to data" type and "pointer to void*" type. (2) Pointer type shall not be converted to other pointer type or integer type with less data width than that of the pointer type, with the exception of mutual conversion between "pointer to data" type and "pointer to void*" type. (3) Pointer type shall not be converted to other pointer type or integer type with less data width than that of the pointer type, with the exception of mutual conversion between "pointer to data" type and "pointer to other data" type, and between "pointer to data" type and "pointer to void*" type.	47

[Reliability 2] R2 Use data by taking their ranges, sizes and internal representations into consideration.				
Practice in detail		Rule		Page
R2.7	Pay attention to pointer types.	R2.7.2	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.【MISRA C:2012 R11.8】	50
		R2.7.3	Comparison to check whether a pointer is negative or not shall not be performed.	50
R2.8	Write in a way that will enable the compiler to check that there are no conflicting declarations, usages and definitions.	R2.8.1	Functions with no parameters shall be declared with a void type parameter.	51
		R2.8.2	(1) Functions shall not be defined with a variable number of arguments.【MISRA C:2004 16.1】 (2) When using functions with a variable number of arguments,《they shall be used after documenting the intended behaviors based on the compiler used.》	51
		R2.8.3	One prototype declaration shall be made at one place from where it can be referenced by both the function calls and function definition.	52
[Reliability 3] R3 Write in a way that ensures intended behavior.				
Practice in detail		Rule		Page
R3.1	Write in a way that is conscious of area size.	R3.1.1	(1) In an extern declaration of an array, the number of elements shall always be specified. (2) In an extern declaration of an array, the number of elements shall always be specified, except for extern declarations of arrays that correspond to the array definition that includes initialization and has omitted the number of elements.	54
		R3.1.2	Iteration conditions for a loop to sequentially access array elements shall include the decision to whether the access is within the range of the array or not.	55
		R3.1.3	The size of the array initialized with a designated initializer shall be clearly indicated.	55
		R3.1.4	Variable length array type shall not be used.【MISRA C:2012 R18.8】	56
R3.2	Prevent operations that may cause runtime error from falling into error cases.	R3.2.1	Operations shall be performed after confirming that the right-hand side expression of division or remainder operation is not 0.	57

[Reliability 3] R3 Write in a way that ensures intended behavior.				
Practice in detail		Rule		Page
R3.2	Prevent operations that may cause runtime error from falling into error cases.	R3.2.2	Destination pointed by a pointer shall be referenced to after checking that the pointer is not the null pointer.	57
R3.3	Check the interface restrictions when a function is called.	R3.3.1	If a function returns error information, then that error information shall be tested.【MISRA C:2012 D4.7】	58
		R3.3.2	The function shall check if there are constraints on parameters before starting to process.	59
R3.4	Do not perform recursive calls.	R3.4.1	Functions shall not call themselves, either directly or indirectly.【MISRA C:2012 R17.2】	60
R3.5	Pay attention to branch conditions and describe how to handle cases that do not follow the predefined conditions when they occur.	R3.5.1	The else clause shall be written at the end of an if-else if statement. If it is known that the else condition does not normally occur, the description of the else caluse shall be either one of the following: 《(i) An exception handling process shall be written in the else clause. (ii) A comment specified by the project shall be written in the else clause.》	61
		R3.5.2	《The default clause shall be written at the end of a switch statement. If it is known that the default condition does not normally occur, the description of the default clause shall be either one of the followings. 《(i) An exception handling process shall be written in the default clause. (ii) A comment specified by the project shall be written in the default clause.》	62
		R3.5.3	Equality operators (==) or inequality operators (!=) shall not be used for comparisons of loop counters. (<=, >=, <, or > shall be used.)	63
R3.6	Pay attention to the order of evaluation.	R3.6.1	Variables whose values are changed in an expression shall not be referred to or modified in the same expression.	64
		R3.6.2	Function calls with side effects and volatile variables shall not be described more than once in a sequence of actual arguments or binary operation expressions.	65
		R3.6.3	sizeof operator shall not be used in expressions that have side effect.	66

[Maintainability1] M1 Keep in mind that others will read the program				
Practice in detail		Rule		Page
M1.1	Do not leave unused descriptions.	M1.1.1	Unused functions, variables, parameters, typedefs, tags, labels or macros shall not be declared (defined).	69
		M1.1.2	Sections of code should not be “commented out”. 【MISRA C:2012 D4.4】	69
M1.2	Do not writing confusingly.	M1.2.1	(1) Only one variable shall be declared in one declaration statement (avoid multiple declarations.) (2) Automatic variables of the same type used for the similar purposes may be declared in one declaration statement, but variables with initialization and variables without initialization shall not be mixed.	70
		M1.2.2	Suffixes shall be added to constant descriptions that can use them to indicate appropriate types. Only an uppercase letter “L” shall be used for a suffix indicating a long type integer constant.	71
		M1.2.3	When expressing a long string literal, successive string literals shall be concatenated without using newlines within the string literal.	71
M1.3	Do not write in an unconventional style.	M1.3.1	Expressions evaluating to true or false shall not be described in switch (expression).	72
		M1.3.2	The case labels and default label in a switch statement shall be described only in the compound statement (excluding nested compound statements) within the body of the switch statement.	72
		M1.3.3	The types shall be explicitly described for definitions and declarations of functions and variables.	73
M1.4	Write in a style that clearly specifies the operator precedence.	M1.4.1	Expressions described at the right hand and left hand of && and || operations shall be either simple variables or expressions enclosed with ( ). However, if only && operations or only || operations are successively combined, it is not necessary to enclose each && and || expression with ( ).	73
		M1.4.2	《Usage of parentheses to explicitly indicate operator precedence shall be defined.》	74

[Maintainability1] M1 Keep in mind that others will read the program				
Practice in detail		Rule		Page
M1.5	Explicitly describe the operations that are likely to cause misunderstanding when they are omitted.	M1.5.1	A function identifier (function name) shall only be used with either a preceding “&”, or with a parenthesized parameter list, which may be empty.【MISRA C:2004 16.9】	74
		M1.5.2	Comparisons with zero (0) shall be explicitly written in conditional expressions.	75
M1.6	Use one area for one purpose.	M1.6.1	Variables shall be prepared for each purpose.	75
		M1.6.2	(1) Unions shall not be used. (2) If unions are used, the same members that are assigned values shall be referenced.	76
M1.7	Do not reuse names.	M1.7.1	The rules below shall be followed for name uniqueness. 1. An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.【MISRA C:2012 R5.3】 2. A typedef name shall be a unique identifier.【MISRA C:2013 R5.6】 3. A tag name shall be a unique identifier.【MISRA C:2012 R5.7】 4. Identifiers that define objects or functions with external linkage shall be unique.【MISRA C:2012 R5.8】 5. Identifiers that define objects or functions with internal linkage should be unique.【MISRA C:2012 R5.9】 6. No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.【MISRA C:2004 R5.6】	77
		M1.7.2	Names for functions, variables and macros in the standard library shall not be redefined or reused. In addition, those macro names shall not be undefined.	78
		M1.7.3	Names (variables) that start with an underscore shall not be defined.	79
M1.8	Do not use language specifications that are likely to cause misunderstanding.	M1.8.1	The right-hand operand of a logical && or || operator shall not contain side effects.【MISRA C:2012 R13.5】	80

[Maintainability1] M1 Keep in mind that others will read the program				
Practice in detail		Rule		Page
M1.8	Do not use language specifications that are likely to cause misunderstanding.	M1.8.2	C macros shall only expand to a braced initializer, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct. 【MISRA 2004 19.4】	80
		M1.8.3	#line shall not be used, unless it is automatically generated by a tool.	81
		M1.8.4	Sequences of three or more characters starting with ?? and alternative tokens shall not be used.	81
		M1.8.5	A sequence starting with zero (0) that is two or more digits long shall not be used as a constant.	82
M1.9	When writing in an unconventional style, explicitly state its intention.	M1.9.1	If statements that do nothing need to be intentionally described, comments or empty macros shall be used to make them noticeable.	82
		M1.9.2	《The unified style of writing infinite loops shall be defined.》	83
M1.10	Do not embed magic numbers.	M1.10.1	A meaningful constant shall be used after defining it as a macro.	83
M1.11	Explicitly state the area attributes	M1.11.1	Read-only areas shall be declared as const type.	84
		M1.11.2	Areas that may be updated by other execution units shall be declared as volatile.	84
		M1.11.3	《Rules for variable declaration and definition for ROMization shall be defined》	85
M1.12	Correctly describe the statements even if they are not compiled.	M1.12.1	Correct code shall be described even if it is going to be deleted by the preprocessor.	86
[Maintainability2] M2 Write in a style that can prevent modification errors.				
Practice in detail		Rule		Page
M2.1	Clarify the grouping of structured data and blocks.	M2.1.1	If arrays and structures are initialized with values other than 0, their structural form shall be indicated by using braces '{ }'. Data shall be described without any omission, except when all values are 0.	88
		M2.1.2	The body of if, else if, else, while, do, for, and switch statements shall be enclosed into blocks.	88

[Maintainability2] M2 Write in a style that can prevent modification errors.			
Practice in detail	Rule		Page
M2.2 Localize access ranges and related data.	M2.2.1	Variables used only in one function shall be declared within the function.	89
	M2.2.2	Variables accessed by several functions defined in the same file shall be declared with static in the file scope..	90
	M2.2.3	Functions that are called only by functions defined in the same file shall be static.	90
	M2.2.4	enum shall be used rather than #define when defining related constants.	91
[Maintainability3] M3 Write programs simply.			
Practice in detail	Rule		Page
M3.1 Do structured programming.	M3.1.1	For any iteration statement, there shall be at most one break statement used for loop termination.【MISRA C:2012 R15.4】	93
	M3.1.2	(1)The goto statement shall not be used. (2) When using a goto statement, the destination to jump to shall be the label declared after the goto statement that is in the same block or within the block enclosing the goto statement.	94
	M3.1.4	(1) Each case clause and default clause in a switch statement shall always end with a break statement. (2) If the case clause or default clause in a switch statement is not going to be ended with a break statement,《a project-specific comment shall be defined》 and that comment shall instead be inserted.	95
	M3.1.5	(1) A function shall end with one return statement. (2) A return statement to return in the middle of processing shall be written only in case of recovery from abnormality.	96
M3.2 One statement should have one side effect.	M3.2.1	(1) Comma expressions shall not be used. (2) Comma expressions shall not be used, other than in expressions for initializing or updating in for statements.	96
	M3.2.2	Multiple assignments shall not be written in one statement, except when the same value is assigned to multiple variables.	97

[Maintainability3] M3 Write programs simply.				
Practice in detail		Rule		Page
M3.3	Separately describe expressions with different purposes.	M3.3.1	The three expressions of a for statement shall be concerned only with loop control.【MISRA C:2004 13.5】	97
		M3.3.2	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.【MISRA C:2004 13.6】	98
		M3.3.3	(1) Assignment operators shall not be used in expressions to examine true or false. (2) Assignment operators shall not be used in expressions to examine true or false, except for conventionally used notations.	98
M3.4	Do not use complicated pointer operations.	M3.4.1	Three or more pointer indirections shall not be used.	99
[Maintainability4] M4 Write in a unified style.				
Practice in detail		Rule		Page
M4.1	Unify the coding styles.	M4.1.1	《Conventions regarding the style of using, such as, the braces'{ }', indentation and space shall be defined.》	101
M4.2	Unify the style of writing comments.	M4.2.1	《Convention regarding the style of writing file header comments, function header comments, end of line comments, block comments and copyright shall be defined.》	104
M4.3	Unify the naming conventions.	M4.3.1	《Convention for naming external variables and internal variables shall be defined.》	106
		M4.3.2	《Convention for naming files shall be defined.》	106
M4.4	Unify the contents to be described in a file and the order of describing them.	M4.4.1	《The descriptive contents of header files (declarations, definitions, etc) and the order they are described in shall be defined.》	109
M4.4	Unify the contents to be described in a file and the order of describing them.	M4.4.2	《The descriptive contents of source files (declarations, definitions, etc) and the order they are described in shall be defined.》	110
		M4.4.3	To use or define external variables or functions (except for functions used only in the file), the header file describing their declarations shall be included.	111

[Maintainability4] M4 Write in a unified style.				
Practice in detail		Rule		Page
M4.4	Unify the contents to be described in a file and the order of describing them.	M4.4.4	External variables shall not be defined in multiple locations.	111
		M4.4.5	Variable definitions or function definitions shall not be described in a header file.	112
		M4.4.6	Header files shall be descriptively capable of handling redundant inclusions.《The descriptive method to achieve this capability shall be defined.》	112
M4.5	Unify the style of writing declarations.	M4.5.1	(1) In a function prototype declaration, all the parameters shall not be named (types only.) (2) In a function prototype declaration, all the parameters shall be named. In addition, the types of the parameters, their names and the type of the return value shall be literally the same as those of the function definition.	113
		M4.5.2	Structure tags and variables shall be declared separately.	114
		M4.5.3	(1) "," shall not be placed before the last "}" in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. (2) "," shall not be placed before the last "}" in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. However, placing "," before the last "}" in the list of initial values for array initialization is allowed.	114
M4.6	Unify the style of writing null pointers.	M4.6.1	(1) 0 shall be used for the null pointer. NULL shall not be used in any case. (2) NULL shall be used for the null pointer. NULL shall not be used for anything other than the null pointer.	115
M4.7	Unify the style of writing preprocessor directives.	M4.7.1	The body and parameters of a macro that includes operators shall be enclosed with parentheses ( ).	116
		M4.7.2	#else, #elif or #endif that correspond to #ifdef, #ifndef or #if shall be described in the same file, and《their correspondence relationship shall be clearly stated with a comment defined in the project》.	116

[Maintainability4] M4 Write in a unified style.				
Practice in detail		Rule		Page
M4.7	Unify the style of writing preprocessor directives.	M4.7.3	defined(macro_name) or defined macro_name shall be used to check whether the macro name has already been defined by #if or #elif.	117
		M4.7.5	Macros shall not be #define'd or #undef'd within a block.【MISRA C:2004 19.5】	117
		M4.7.6	#undef shall not be used.【MISRA C:2012 R19.6】	118
		M4.7.7	Controlling expression of #if or #elif preprocessing directive shall be evaluated as 0 or 1.【MISRA C:2012 R20.8】	119
[Maintainability5] M5 Write in a style that makes testing easy.				
Practice in detail		Rule		Page
M5.1	Write in a style that makes it easy to investigate the causes of problems when they occur.	M5.1.1	《The rules for writing the code for setting debug options and for recording logs in release modules shall be defined.》	121
		M5.1.2	(1) The # and ## preprocessor operators should not be used.【MISRA C:2012 R20.10】 (2) A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.【MISRA C:2012 R20.11】	123
		M5.1.3	Function shall be used rather than using function-like macro.	124
M5.2	Be careful when using dynamic memory allocations.	M5.2.1	(1) Dynamic memory shall not be used. (2) If dynamic memory is used,《The maximum amount of memory that can be used, process to be taken when running out of memory, and debugging procedure shall be defined.》	124
[Portability 1] P1 Write in a style that is not dependent on the compiler.				
Practice in detail		Rule		Page
P1.1	Do not use functionalities that are advanced features or implementation-defined.	P1.1.1	(1) Functionalities not specified in the language standard shall not be used. (2) If functionalities not specified in the language standard are used,《the functionalities used and their usage shall be documented.》	129
		P1.1.2	《All usage of implementation-defined behavior shall be documented.》【MISRA C:2004 3.1】	129

[Portability 1] P1 Write in a style that is not dependent on the compiler.				
Practice in detail		Rule		Page
P1.1	Do not use functionalities that are advanced features or implementation-defined.	P1.1.3	To use a program written in another language,《its interface shall be documented and its usage shall be defined.》	130
P1.2	Use only the characters and escape sequences defined in the language standard.	P1.2.1	To use characters other than those defined in the language standard for writing a program, the compiler specifications shall be confirmed, and《their usage shall be defined.》	130
		P1.2.2	Only escape sequences defined in the language standard shall be used.	131
P1.3	Confirm and document data type representations, behavioral specifications of advanced functionalities and implementation-dependent parts.	P1.3.1	Simple char type (that does not specify the signedness) shall be used only for storing character values. If a process that depends on signedness (implementation-defined) is required, unsigned char or signed char that specifies its signedness shall be used.	132
		P1.3.2	The members of an enumeration type (enum) shall be defined with values that can be represented as int type.	133
		P1.3.3	(1) Bit fields shall not be used. (2) it fields shall not be used for data whose bit positions are meaningful. (3)《If it is being relied upon, the implementation-defined behavior and packing of bit fields shall be documented.》【MISRA C:2004 3.5】	133
P1.4	For source file inclusion, confirm the implementation-dependent parts and write in a style that is not implementation-dependent.	P1.4.1	The #include directive shall be followed by either a <filename> or “filename” sequence. 【MISRA C:2012 R20.3】	134
		P1.4.2	《The usage of <> format and “” format for #include file specification shall be defined.》	135
		P1.4.3	Characters ‘, \, “, /*, // and : shall not be used for file specification in #include.	135
P1.5	Write in a style that does not depend on the environment used for compiling.	P1.5.1	The absolute path shall not be written for #include file specification.	136

[Portability 2] P2 Localize the code that has a problem with portability.			
Practice in detail	Rule		Page
P2.1 Localize the code that has a problem with portability.	P2.1.1	When assembly language programs are called from C language,《how to localize such parts shall be defined》,such as, by expressing them as functions or inline functions of C language that contain only inline assembly language code or describing them using macros.	138
	P2.1.2	Keywords extended by the compiler shall be used by localizing them after《defining the macros.》.	138
	P2.1.3	(1) The basic types (char, int, long, long long, float, double and long double) shall not be used. Instead, the types defined by typedef shall be used.《The types defined by typedef that are used in the project shall be defined.》 (2) When using any of the basic types (char, int, long, long long, float, double and long double) in a form that is dependent on its size, the type defined by typedef for each of these basic types shall be used.《The types defined by typedef that are used in the project shall be defined.》	139
[Efficiency1] E1 Write in a style that takes account of resource and time efficiencies.			
Practice in detail	Rule		Page
E1.1 Write in a style that takes account of resource and time efficiencies.	E1.1.1	Write in a style that takes account of resource and time efficiencies.	143
	E1.1.2	Operations that remain unchanged shall not be performed within an iterated process.	143
	E1.1.3	Instead of structures, pointers to structures shall be used as function parameters.	144
E1.1 Write in a style that takes account of resource and time efficiencies.	E1.1.4	《The policy of selecting either switch or if statement shall be determined and defined by taking readability and efficiency into consideration.》	144

Appendix B Rule classification based on the C language grammar

The rules are classified according to the C language grammar shown below.

Classification based on the grammar		No.	Rule
1. Style			
1.1	Syntax style	M4.1.1	《Conventions regarding the style of using, such as, the braces'{ }', indentation and space shall be defined.》
1.2	Comments	M4.2.1	《Convention regarding the style of writing file header comments, function header comments, end of line comments, block comments and copyright shall be defined.》
1.3	Naming	M1.7.1	The rules below shall be followed for name uniqueness. 1. An identifi er declared in an inner scope shall not hide an identifi er declared in an outer scope.【MISRA C:2012 R5.3】 2. A typedef name shall be a unique identifi er.【MISRA C:2013 R5.6】 3. A tag name shall be a unique identifi er.【MISRA C:2012 R5.7】 4. Identifiers that define objects or functions with external linkage shall be unique.【 MISRA C:2012 R5.8】 5. Identifiers that define objects or functions with internal linkage should be unique.【MISRA C:2012 R5.9】 6.No identifi er in one name space should have the same spelling as an identifi er in another name space, with the exception of structure member and union member names.【MISRA C:2004 5.6】
		M1.7.2	Names for functions, variables and macros in the standard library shall not be redefined or reused. In addition, those macro names shall not be undefined.
		M1.7.3	Names (variables) that start with an underscore shall not be defined.
		M4.3.1	《Convention for naming external variables and internal variables shall be defined.》
		M4.3.2	《Convention for naming files shall be defined.》
1.4	Composition of a file	M4.4.1	《The descriptive contents of header files (declarations, definitions, etc) and the order they are described in shall be defined.》
		M4.4.2	《The descriptive contents of source files (declarations, definitions, etc) and the order they are described in shall be defined.》
		M4.4.3	To use or defi ne external variables or functions (except for functions used only in the file), the header file describing their declarations shall be included.
		M4.4.5	Variable definitions or function definitions shall not be described in a header file.

Classification based on the grammar		No.	Rule
1. Style			
1.4	Composition of a file	M4.4.6	Header files shall be descriptively capable of handling redundant inclusions. 《The descriptive method to achieve this capability shall be defined》
1.5	Constants	M1.2.3	When expressing a long string literal, successive string literals shall be concatenated without using newlines within the string literal.
		M1.2.2	Suffixes shall be added to constant descriptions that can use them to indicate appropriate types. Only an uppercase letter “L” shall be used for a suffix indicating a long type integer constant.
		M1.8.5	A sequence starting with zero (0) that is two or more digits long shall not be used as a constant.
		M1.10.1	A meaningful constant shall be used after defining it as a macro.
1.6	Other (Style)	M1.1.2	Sections of code should not be “commented out”. 【MISRA C:2012 D4.4】
		M1.8.4	Sequences of three or more characters starting with ?? and alternative tokens shall not be used.
		M1.9.1	If statements that do nothing need to be intentionally described, comments or empty macros shall be used to make them noticeable.
		M5.1.3	Function shall be used rather than using function-like macro.
2. Type			
2.1	Basic types	P1.3.1	Simple char type (that does not specify the signedness) shall be used only for storing character values. If a process that depends on signedness (implementation-defined) is required, unsigned char or signed char that specifi es its signedness shall be used.
		P2.1.3	(1) The basic types (char, int, long, long long, float, double and long double) shall not be used. Instead, the types defined by typedef shall be used. 《The types defined by typedef that are used in the project shall be defined.》 (2) When using any of the basic types (char, int, long, long long, fl oat, double and long double) in a form that is dependent on its size, the type defined by typedef for each of these basic types shall be used. 《The types defined by typedef that are used in the project shall be defined》
		R2.6.2	Data used as bit sequences shall be defined with unsigned type, and not with the signed type

Classification based on the grammar		No.	Rule
2. Type			
2.2	Structures/Unions	R2.1.3	memcmp shall not be used to compare structures and unions.
		M1.6.2	(1) Unions shall not be used.【MISRA C:2004 18.4】 (2) If unions are used, the same members that are assigned values shall be referenced.
		M1.7.2	Names for functions, variables and macros in the standard library shall not be redefined or reused. Inaddition, those macro names shall not be undefined.
		M4.5.2	Structure tags and variables shall be declared separately.
2.3	Bit fields	R2.6.1	(1) The types used for bit field shall only be signed int or unsigned int. If a bit field of 1 bit width is required, unsigned int type shall be used, and not the unsigned int type (2) The types used for bit field shall be signed int, unsigned int or _Bool. If a bit field of 1 bit width is required, unsigned int type or _Bool type shall be used. (3) The types used for bit field shall be signed int, unsigned int, _Bool, or those allowed by the compiler that are either enum or the type that specifi es signed or unsigned. If a bit field of 1 bit width is required, the type that specifi es unsigned or _Bool type shall be used.
		P1.3.3	(1) Bit fields shall not be used. (2) Bit fields shall not be used for data whose bit positions are meaningful. (3)《If it is being relied upon, the implementationdefined behavior and packing of bit fields shall be documented.》【MISRA C:2004 3.5】
2.4	Enumerated type	R1.2.2	Initialization of enumeration type (enum type) members shall be by either: not specifying any constants; specifying all the constants; or specifying only the first member.
		M2.2.4	enum shall be used rather than #define when defi ning related constants.
		P1.3.2	The members of an enumeration type (enum) shall be defined with values that can be represented as int type.
3. Declaration/Definition			
3.1	Initialization	R1.1.1	Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.
		R1.1.2	const variables shall be initialized at the time of declaration.
		R1.2.1	Arrays with specifi ed number of elements shall beinitialized with values that match the number of the elements.

Classification based on the grammar		No.	Rule
3. Declaration/Definition			
3.1	Initialization	M2.1.1	If arrays and structures are initialized with values other than 0, their structural form shall be indicated by using braces '{ }'. Data shall be described without any omission, except when all values are 0.
		M4.5.3	(1) "," shall not be placed before the last "}" in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators (2) "," shall not be placed before the last "}" in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. However, placing "," before the last "}" in the list of initial values for array initialization is allowed.
3.2	Variable declaration/ definition	M1.2.1	(1) Only one variable shall be declared in one declaration statement (avoid multiple declarations.) (2) Automatic variables of the same type used for the similar purposes may be declared in one declaration statement, but variables with initialization and variables without initialization shall not be mixed.
		M1.6.1	Variables shall be prepared for each purpose.
		M1.11.1	Read-only areas shall be declared as const type.
		M1.11.2	Areas that may be updated by other execution units shall be declared as volatile.
		M1.11.3	《Rules for variable declaration and definition for ROM ization shall be defined》
		M2.2.1	Variables used only in one function shall be declared within the function.
		M2.2.2	Variables accessed by several functions defined in the same file shall be declared with static in the file scope.
		M4.4.4	External variables shall not be defined in multiple locations.
3.3	Function declaration/ definition	R2.8.1	Functions with no parameters shall be declared with a void type parameter.
		R2.8.2	(1) Functions shall not be defined with a variable number of arguments. 【MISRAC:2004 16.1】 (2) When using functions with a variable number of arguments, 《 they shall be used after documenting the intended behaviors based on the compiler used.
		R2.8.3	One prototype declaration shall be made at one place from where it can be referenced by both the function calls and function defi nition.

Classification based on the grammar		No.	Rule
3. Declaration/Definition			
3.3	Function declaration/ definition	M2.2.3	Functions that are called only by functions defined in the same file shall be static.
		M4.5.1	(1) In a function prototype declaration, all the parameters shall not be named (types only.) (2) In a function prototype declaration, all the parameters shall be named. In addition, the types of the parameters, their names and the type of the return value shall be literally the same as those of the function defi nition.
3.4	Array declaration/ definition	R3.1.1	(1) In an extern declaration of an array, the number of elements shall always be specified. (2) In an extern declaration of an array, the number of elements shall always be specified, except for extern declarations of arrays that correspond to the array defi nition that includes initialization and has omitted the number of elements.
		R3.1.4	Variable length array type shall not be used.【MISRA C:2012 R18.8】
3.5	Other (declaration/ definition)	M1.1.1	Unused functions, variables, parameters, typedefs, tags, labels or macros shall not be declared (defined).
		M1.3.3	The types shall be explicitly described for defi nitions and declarations of functions and variables.
4. Expression			
4.1	Function call	R3.3.1	If a function returns error information, then that error information shall be tested【MISRA C:2012 D4.7】
		R3.3.2	The function shall check if there are constraints on parameters before starting to process
		R3.4.1	Functions shall not call themselves, either directly or indirectly【MISRA C:2012 R17.2】
4.2	Pointer	R2.7.1	(1) Pointer type shall not be converted to other pointer type or integer type, and vice versa, with the exception of mutual conversion between “pointer to data” type and “pointer to void*” type. (2) Pointer type shall not be converted to other pointer type or integer type with less data width than that of the pointer type, with the exception of mutual conversion between “pointer to data” type and “pointer to void*” type (3) Pointer type shall not be converted to other pointer type or integer type with less data width than that of the pointer type, with the exception of mutual conversion between “pointer to data” type and “pointer to other data” type, and between “pointer to data” type and “pointer to void*” type.

Classification based on the grammar		No.	Rule
4. Expression			
4.2	Pointer	R1.3.1	(1) Integer addition to or subtraction from (including ++ and --) pointers shall not be made; Array format with [ ] shall be used for references and assignments to the allocated area. (2)Integer addition to or subtraction from (including ++ and --) pointers shall be made only when the pointer points to the array and the result must be pointing within the range of the array.
		R1.3.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array.【MISRA C:2012 R18.2】
		R1.3.3	Comparison between pointers shall be used only when the two pointers are both pointing at either the elements in the same array or the members of the same structure.
		R2.7.3	Comparison to check whether a pointer is negative or not shall not be performed.
		R3.2.2	Destination pointed by a pointer shall be referenced to after checking that the pointer is not the null pointer.
		M3.4.1	Three or more pointer indirections shall not be used.
		M4.6.1	(1) 0 shall be used for the null pointer. NULL shall not be used in any case. (2) NULL shall be used for the null pointer. NULL shall not be used for anything other than the null pointer.
4.3	Cast	R2.4.2	When performing arithmetic operations or comparisons of expressions mixed with signed and unsigned, an explicit cast to the expected type shall be performed.
		R2.7.2	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.【MISRA C:2012 R11.8】
4.4	Unary operation	R2.5.2	Unary operator '-' shall not be used in unsigned expressions.
		R3.6.3	sizeof operator shall not be used in expressions that have side effect.
		M1.5.1	A function identifier (function name) shall only be used with either a preceding "&", or with a parenthesized parameter list, which may be empty. 【MISRA C:2004 16.9】

Classification based on the grammar		No.	Rule
4. Expression			
4.5	The four arithmetic operations	R3.2.1	Operations shall be performed after confi rming that the right-hand side expression of division or remainder operation is not 0.
4.6	Shift	R2.5.4	The right-hand side of a shift operator shall be zero or more, and less than the bit width of the left-hand side.
4.7	Comparison	R2.1.1	Floating-point expressions shall not be used to perform equality or inequality comparisons.
		R2.2.1	Comparison with a value defined as true shall not be made in expressions that examine true or false.
		M1.5.2	Comparisons with 0 shall be explicitly written.
4.8	Bit operation	R2.5.3	When ones' complement (~) or left shift (<<) is applied to unsigned char or unsigned short type data, an explicit cast to the type of the operation result shall be performed.
4.9	Logical operation	M1.4.1	Expressions described at the right hand and left hand of && and || operations shall be either simple variables or expressions enclosed with ( ). However, if only && operations or only || operations are successively combined, it is not necessary to enclose each && and || expression with ( ).
		M1.8.1	The right-hand operand of a logical && or || operator shall not contain side effects.【MISRA C:2012 R13.5】
4.10	Ternary operation	R2.3.2	When using conditional operator (?: operator), the logical expression shall be enclosed in parentheses ( ) and both return values shall be the same type.
4.11	Assignment	R2.4.1	When the type of an operation and the type of the destination to which the operation result is assigned (assignment destination) are different, the operation shall be performed after casting them to the type of expected operation precision.
		R2.5.1	When performing assignments (=operation, actual arguments passing of function calls, function return) or operations to data types that may cause information loss, they shall be first confirmed that there are no problems, and a cast shall be described to explicitly state that they are problem-free.

Classification based on the grammar		No.	Rule
4. Expression			
4.11	Assignment	M3.3.3	(1) Assignment operators shall not be used in expressions to examine true or false. (2) Assignment operators shall not be used in《expressions to examine true or false, except for conventionally used notations.》
4.12	Comma	M3.2.1	(1) Comma expressions shall not be used. (2) Comma expressions shall not be used, other than in expressions for initializing or updating in for statements.
4.13	Priority and Side effect	R3.6.1	Variables whose values are changed in an expression shall not be referred to or modified in the same expression.
		R3.6.2	Function calls with side effects and volatile variables shall not be described more than once in a sequence of actual arguments or binary operation expressions.
		M1.4.2	《Usage of parentheses to explicitly indicate operator precedence shall be defined.》
4.14	Other (Expression)	R2.3.1	Unsigned integer constant expressions shall be described within the range that can be represented with the result type.
5. Statement			
5.1	if statement	R3.5.1	The else clause shall be written at the end of an if-else if statement. If it is known that the else condition does not normally occur, the description of the else caluse shall be either one of the following: 《(i) An exception handling process shall be written in the else clause. (ii) A comment specifi ed by the project shall be written in the else clause. 》
5.2	switch statement	M3.1.4	(1) Each case clause and default clause in a switch statement shall always end with a break statement. (2) If the case clause or default clause in a switch statement is not going to be ended with a break statement,《 a project-specifi c comment shall be defined》 and that comment shall instead be inserted
		R3.5.2	《The default clause shall be written at the end of a switch statement. If it is known that the default condition does not normally occur, the description of the default clause shall be either one of the following. 《(i) An exception handling process shall be written in the default clause. (ii) A comment specifi ed by the project shall be written in the default clause.》

Classification based on the grammar		No.	Rule
5. Statement			
5.2	switch statement	M1.3.1	Expressions evaluating to true or false shall not be described in switch (expression).
		M1.3.2	The case labels and default label in a switch statement shall be described only in the compound statement (excluding nested compound statements) within the body of the switch statement.
5.3	for/while statement	R2.1.2	Floating-point variable shall not be used as a loop counter.
		R2.3.3	Loop counters and variables used for comparison of loop iteration conditions shall be the same type.
		R3.1.2	Iteration conditions for a loop to sequentially access array elements shall include the decision to whether the access is within the range of the array or not.
		R3.5.3	Equality operators (==) or inequality operators (!=) shall not be used for comparisons of loop counters. (<=, >=, <, or > shall be used.)
		M1.9.2	《The unified style of writing infinite loops shall be defined》
		M3.1.1	For any iteration statement, there shall be at most one break statement used for loop termination.【MISRA C:2012 R15.4】
		M3.3.1	The three expressions of a for statement shall be concerned only with loop control.【 MISRA C:2004 13.5】
		M3.3.2	Numeric variables being used within a for loop for iteration counting shall not be modifi ed in the body of the loop.【MISRA C:2004 13.6】.
5.4	Other (statement)	M2.1.2	The body of if, else if, else, while, do, for, and switch statements shall be enclosed into blocks.
		M3.1.2	(1) The goto statement shall not be used. (2) When using a goto statement, the destination to jump to shall be the label declared after the goto statement that is in the same block or within the block enclosing the goto statement.
		M3.1.5	(1) A function shall end with one return statement. (2) A return statement to return in the middle of processing shall be written only in case of recovery from abnormality.

Classification based on the grammar		No.	Rule
5. Statement			
5.4	Other (statement)	M3.2.2	Multiple assignments shall not be written in one statement, except when the same value is assigned to multiple variables.
6. Macro/preprocessor			
6.1	#if related	M4.7.2	#else, #elif or #endif that correspond to #ifdef, #ifndef or #if shall be described in the same file, and《 their correspondence relationship shall be clearly stated with a comment defined in the project 》.
		M4.7.3	“defined(macro_name) or defined macro_name shall be used to check whether the macro name has already been defined by #if or #elif.
		M4.7.7	Controlling expression of #if or #elif preprocessing directive shall be evaluated as 0 or 1.【MISRA C:2012 R20.8】.
6.2	#include	P1.4.1	The #include directive shall be followed by either a <filename> or “filename” sequence.【MISRA C:2012 R20.3】
		P1.4.2	《The usage of <> format and “” format for #include file specifi cation shall be defined.》
		P1.4.3	Characters ‘, \, “, /*, // and : shall not be used for file specifi cation in #include.
		P1.5.1	The absolute path shall not be written for #include file specification.
6.3	Macro	M1.8.2	C macros shall only expand to a braced initializer, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.【 MISRA C:2004 19.4】
		M4.7.1	The body and parameters of a macro that includes operators shall be enclosed with parentheses ( ).
		M4.7.5	Macros shall not be #define’d or #undef’d within a block.【MISRA C:2004 19.5】
		M4.7.6	#undef shall not be used.【MISRA C:2012 R19.6】
6.4	Other (preprocessor)	M1.8.3	#line shall not be used, unless it is automatically generated by a tool.
		M1.12.1	Correct code shall be described even if it is going to be deleted by the preprocessor.

Classification based on the grammar		No.	Rule
6. Macro/preprocessor			
6.4	Other (preprocessor)	M5.1.2	(1) The # and ## preprocessor operators should not be used.【MISRA C:2012 R20.10】】 (2) A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.【MISRA C:2012 R20.11】
7. Environment/Other			
7.1	Portability	P1.1.1	(1) Functionalities not specifi ed in the language standard shall not be used. (2) If functionalities not specifi ed in the language standard are used,《the functionalities used and their usage shall be documented.》
		P1.1.2	《All usage of implementation-defined behavior shall be documented.》【MISRA C:2004 3.1】
		P1.2.1	To use characters other than those defined in the language standard for writing a program, the compiler specifi cations shall be confirmed, and《their usage shall be defined.》
		P1.2.2	Only escape sequences defined in the language standard shall be used.
		P1.1.3	To use a program written in another language,《its interface shall be documented and its usage shall be defined.》
		P2.1.1	When assembly language programs are called from C language,《how to localize such parts shall be defined》,such as, by expressing them as functions or inline functions of C language that contain only inline assembly language code or describing them using macros.
		P2.1.2	Keywords extended by the compiler shall be used by localizing them after《defi ning the macros.》
7.2	Performance	E1.1.1	Macro functions shall be used only in parts related to speed performance.
		R1.3.4	The restrict type qualifier shall not be used.【MISRA C:2012 R8.14】.
		E1.1.2	Operations that remain unchanged shall not be performed within an iterated process
		E1.1.3	Instead of structures, pointers to structures shall be used as function parameters.
		E1.1.4	《The policy of selecting either switch or if statement shall be determined and defined by taking readability and effi ciency into consideration.》

Classification based on the grammar		No.	Rule
7. Environment/Other			
7.3	Description for debug	M5.1.1	《The rules for writing the code for setting debug options and for recording logs in release modules shall be defined.》
7.4	Other	M5.2.1	(1) Dynamic memory shall not be used. (2) If dynamic memory is used 《, the maximum amount of memory that can be used, process to be taken when running out of memory, and debugging procedure shall be defined.》

Appendix C Regarding the implementation-defined behaviors

C language standard has behaviors that are unspecified or undefined in its language specifications. (Refer to "Column: Unspecified Behavior and Undefined Behavior" below.) Some of the unspecified behaviors are defined by the compiler, and they are referred to as "implementation-defined behaviors". Every implementation-defined behavior is compiler-specific. In other words, it always behaves the same way when it is processed by the same type of compiler.

What this also means is that the behavior may not always be the same when it is processed by a different type of compiler, even if the code written in the source program is the same. Therefore, attention is necessary when the program is ported or when the compiler is changed. Moreover, if the programmers are used to working in an environment that only uses a specific type of compiler, they may incorrectly assume that implementation-defined parts of the code used in their development project tare all specified in the C language standard and do not consider the possibility of changes in their implementation-defined behaviors when a different type of compiler is used to process the program that they write. To prevent them from causing any unexpected errors, it is desirable to check and keep in mind which behaviors are implementation-defined before starting the programming process.

Implementation-defined behaviors are normally listed in the manual of each compiler. Some of the widely-known implementation-defined behaviors are outlined below.

Representative implementation-defined behavior 1: Execution environment

A term that often appears in descriptions about implementation-defined behavior is "freestanding environment". Simply put, freestanding environment is an environment that does not have an operating system. In such environment, the name and type of the function called at program startup are implementation-defined. Normally, `main` function is called, but which function (invisible to programmers) is called before the `main` function is called after program startup depends on the compiler that is used.

Moreover, when the main function is terminated or when the program is suspended because the `exit` function is called, the subsequent behavior is implementation-defined. Although writing a program that does not behave differently depending on the compiler used comes first and foremost, it is also necessary for programmers to understand the different kinds of behaviors that can be expected when they are implementation-defined and how they may affect the program execution.

Representative implementation-defined behavior 2: Character code

Character codes are a set of values assigned respectively to the characters, symbols, etc. processed in a computer. Each character coding system has one or more charts that show which character corresponds to which character code in tabular form. In Table 1, the horizontal axis of the chart shows the upper 3-bit and the vertical axis shows the lower 4-bit. As to which character code system will be used is implementation-defined. In case of alphabetic character "A", for example, upper 3-bit

is “4” and lower 4-bit is “1”, which mean that the corresponding code is “0x41”.

The explanation of the above example is based on ASCII (American Standard Code for Information Interchange) 7-bit coding system. But in case of EBCDIC (Extended Binary Coded Decimal Interchange Code), which is an 8-bit coding system as shown below in Table 2, the corresponding code for alphabetic character “A” would be “0xC1”, and not “0x41” as in ASCII.

Lower order bits \ High order bits	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	´	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table 1 ASCII code chart

Lower order bits \ High order bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-						{	}	\	0
1	SOH	DC1	SOS				/		a	j	~		A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	TM							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	ETB	UC					f	o	w		F	O	W	6
7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM						`	i	r	z		I	R	Z	9
A	SMM	CC	SM		¢	!	|	:							
B	VT	CU1	CU2	CU3	·	$	,	#								
C	FF	IFS		DC4	<	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	'								
E	SO	IRS	ACK		+	;	>	=								
F	SI	IUS	BEL	SUB	|	¬	?	"								

Table 2 EBCDIC code chart

As you can see, the code used for representing each character varies with each character coding system. Therefore, close attention is required on how the compiler processes the characters when the character coding system used in the translation environment (environment where the source program is processed) differs from the character coding system used in the execution environment (environment where the execution files are processed for operation).

In case of Japanese characters (hiragana, kanji, etc.), a character code composed of 2 bytes or more is assigned to each character. There are multiple character coding systems that handle Japanese characters. At present, many personal computers in Japan use the character coding system called Shift_JIS that expresses each Japanese character as a double-byte character. Since the value assigned to each Japanese character varies with each character coding system, close attention is necessary on how the compiler handles the Japanese characters.

Another character coding system that began to be widely used in recent years is Unicode, which has been developed to handle the characters of different languages in the world including Japanese in a unified manner. In Unicode, one character may be expressed with 1 byte or multiple bytes (up to 6 bytes). There are mainly three coding systems in Unicode that define the method of expressing each character (encoding method), which are as follows:

1. UTF-8 : All the characters covered in ASCII are expressed with 1 byte. The rest are expressed in variable length (from 2 bytes to 6 bytes).
2. UTF-16 : Uses 16-bit as a unit to express each character. All the characters are expressed either in one unit (16-bit) or two units (32-bit).
3. UTF-32 : All the characters are expressed in a fixed length of 32-bit only.

C language has been extending its features by using “wide character”, which has been introduced to handle the characters expressed in multiple bytes in specific character coding systems respectively with an integer of a fixed bit length. For example, in C99, wchar_t has been introduced as a type for wide character, and libraries that supports wide characters have been added.

Example of a library that supports wide characters:

```
int vwprintf (const wchar_t * restrict format, va_list arg);
    // A version of printf that supports wide character
```

In C11, `char16_t` (2-byte length) and `char32_t` (4-byte length) have been added as types that support wide characters.

The size of wchar_t and the character codes that correspond respectively to wide character types are implementation-defined. In C11, however, two macros, `__STDC_UTF_16__` and `__STDC_UTF_32__` have been introduced, and when these macros are defined, `char16_t` and `char32_t` are encoded respectively according to UTF-16 and UTF-32.

Beside the behavioral differences expected with the character coding system used, attention is also necessary, for example, when kanji characters are expressed in Shift_JIS. In Shift_JIS, each Japanese character is encoded in two bytes. But there are some Japanese characters whose second byte is the same as “\” (back slash or “¥”) in ASCII. For example, the character code that corresponds to the kanji character “表” is “0x955c” in Shift_JIS. The second byte “0x5c” in Shift_JIS corresponds to “\” in ASCII. (See Table 1).

If the compiler used does not support Shift_JIS, double-byte characters will be recognized as single-byte characters. In case of double-byte kanji character “ 表 ”, it will be processed as escape sequence since the compiler will recognize this character as “\”. As a result, the character may be displayed differently from the intended representation (making it garbled, etc.)

Example:

Source code:
```
printf("表\n"); // Byte sequence: 0x95 0x5c 0x5c 0x6e
                //              → 0x5c 0x5c(\\) becomes 0x5c(\).
```

Output:
```
表n             // Byte sequence: 0x95 0x5c 0x6e
                // Actually intended to begin a new line after "表"
```

Representative implementation-defined behavior 3: Pointer and address

Address with an absolute value is often written in the program for embedded software. A pointer is used to access a specific address. In the following case, for example, the calculation requires an integer to be assigned to a pointer (or vice versa).

```
unsigned char *addrp = (unsigned char *)0xffff0123L;
```

The execution code actually used for such conversion between an integer and pointer is implementation-defined. Moreover, the size of the address value is also implementation-defined. These behaviors not only depend on the compiler used, but are also largely dependent on the actual architecture of the processor used for program execution.

Representative implementation-defined behavior 4: Array

The size resulting after subtraction of two pointers to the element in the same array is not necessarily guaranteed as the size (bit length) applied to the address. It is implementation-defined. C99 language standard ISO/IEC9899:1999 defines `ptrdiff_t` in `<stddef.h>` as the type of the result of subtracting two pointers.

Representative implementation-defined behavior 5: Integer

Whether the signed integer type will be expressed as sign and magnitude representation, ones' complement representation or two's complement representation is implementation-defined. Therefore, in the following example,

```
if (( intVal & 0x80000000 ) == 0x80000000 ) { // if the most significant bit is 1
```

expected behavior will occur only if the compiler processes the most significant bit of the signed integer type as representation of the sign bit ('1' if the value is negative).

Moreover, whether the extraordinary value is a trap representation or an ordinary value is also implementation-defined. Extraordinary value refers to the calculation result that is a value that does not fit in the size of the variable. If the variable is unsigned, and the calculation result exceeds the variable representation range, the actual result in that variable will be the remainder of the calculation result divided by the maximum representable value of that variable + 1. Take an unsigned 8-bit variable for example. If the calculated value is 257, the remainder of the value 257 divided by 256 (the maximum representable value of 8-bit variable +1) that is 1 will be the calculation result. This behavior is called "wrap around".

On the other hand, if the variable is signed, and the calculation result exceeds the variable representation range, an overflow will occur. In this case, the compiler may either represent the calculation result in the same way as the case with unsigned variable (as the value left in the variable) or process it as trap representation, which is a bit pattern specially defined in the system for internal processing. If the system is using two's complement, the most significant bit of the pattern defined as trap representation will be 1 and the rest will all be 0.

Representative implementation-defined behavior 6: Bit field

The so-called embedded C compiler can use the bit field of a size of unsigned 8-bit. It is frequently used to access the processor registers where bits are assigned to microcontroller functions.

However, how the bit field is used is implementation-defined. There is no guarantee that a behavior that was normal with a specific compiler will be the same when a different compiler is used. Moreover, whether the bit sequence will be in the ascending order from 0 set as the most significant bit or as the least significant bit is also implementation-defined.

Furthermore, even when the bit field is used, whether the actual execution code will command bit access to, such as, internal registers or not is also implementation-defined. The execution code may command a read-modify-write operation that accesses the byte that includes that bit, and cause an unexpected failure.

Representative implementation-defined behavior 7: Access to volatile qualified type object

`volatile` qualifier is used to suppress the optimization of the compiler. For example, to wait for interrupt, there is a code somewhere in the interrupt handler that sets `InterruptFlag` to 1, as shown below in the while loop, which does the polling.

```
while ( InterruptFlag == 0 ) { ; }
```

In this case, the process to make the variable, `InterruptFlag`, a value of 1 is not in this loop. The compiler may optimize and transform the loop into a simple infinite loop. `volatile` qualifier can prevent the compiler from optimizing in this way.

`volatile` qualified object implicitly indicates that it may be processed without being recognized by the compiler. How the execution code configures the access to `volatile` qualified object is implementation-defined.

Representative implementation-defined behavior 8: Preprocessing directives

There are various preprocessing directives that are implementation-defined, as outlined below.

- Method of corresponding each of the header names specified in series by < > or "" to either the header or the name of the external source file
- Whether the value of the character constant of the constant equation that controls the conditional include matches with the value of the same character constant in the execution character set
- Whether the character constant of a single character of the constant equation controls the conditional include takes a negative value or not
- Method of forming the header name from the preprocessing token in the #include directive (which may also be generated from macro expansion)
- Nesting limitation when processing `#include`
- Whether "\" is inserted in front of "\" that is the first character of a universal character name or not when # operator is in the character constant or string constant
- Behavior of non-STDC `#pragma`
  From C99 (ISO/IEC9899:1999), a specific `#pragma` directive was defined additionally as a standard directive in C language. This is called "STDC (standard C) `#pragma` directive". Any `#pragma` that is not STDC is implementation-defined.
- How the `__DATE__` and `__TIME__` are processed when the translated date and time are not known.

Representative implementation-defined behavior 9: Others

Even when inline instruction or register qualifier is specified, whether it will be forced or not is implementation-defined.

To learn about other implementation-defined behaviors that have not been mentioned above, refer to the manual of the compiler (of the precise version) used in the development.

Column: Unspecified behavior and undefined behavior

In C language, there are four kinds of behavior that require particular attention.

1. **Unspecified behavior**
2. **Undefined behavior**
3. **Implementation-defined behavior**
4. **Locale-specific behavior**

(For details, refer to C99 language standard "ISO/IEC 9899:1999 Programming Language C" Annex J.)

Unspecified behavior and undefined behavior are alike, but do not mean the same, as explained below.

■ Unspecified behavior

There are some behaviors that are grammatically correct (and therefore will not be processed as error) but have alternative execution results depending on which alternative the compiler chooses to process. These behaviors are collectively referred to as "unspecified behavior". For example, the order of evaluating the actual argument to a function may differ depending on the choice made by the compiler.

```
printf("%d %d ¥n", i, i++);
```

In case of the above code, the result displayed will differ depending on whether `i` or `i++` is evaluated first. To gain an overall knowledge about what kind of behavior is unspecified, refer to the list under J.1 in ISO/IEC9899:1999. Descriptions that will cause unspecified behavior should be avoided as much as possible.

■ Undefined behavior

Undefined behavior refers to a set of behaviors that are not defined in C language standard.

For example, the behavior of division by zero is undefined. To gain an overall knowledge about what kind of behavior is unspecified, refer to the list under J.2 in ISO/IEC9899:1999.

Any descriptions that cause undefined behavior must be avoided by all means, since the behavior resulting from any of such descriptions is not defined in the language standard. There is a need to know beforehand which undefined behavior can be detected or not by the static analysis tool that is going to be used for the development.

Citations and References

[1] ISO/IEC 25010:2011, Systems and software enginieering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models.

[2] ISO/IEC 9899:1990, Programming languages – C, ISO/IEC 9899:1990/Cor 1:1994, ISO/IEC 9899:1990/Cor 2:1996, ISO/IEC 9899:1990/Amd 1:1995, C Integrity

[3] ISO/IEC 9899:1999, Programming languages – C, ISO/IEC 9899/Cor1:2001

[4] ISO/IEC 14882:2003, Programming languages – C++

[5] "MISRA Guidelines for the Use of the C Language in Vehicle Based Software", The Motor Industry Software Reliability Association, ISBN 0-9524156-9-0, Apr. 1998, www.misra-c.com

[6] "MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems", The Motor Industry Software Reliability Association, ISBN 0-9524156-2-3, Oct. 2004, www.misra.org.uk

[7] "MISRA C:2012 Guidelines for the Use of the C Language in Critical Systems",March. 2013, ISBN 978-1-906400-10-1, www.misra.org.uk

[8] "Indian Hill Style and Coding Standards", ftp://ftp.cs.utoronto.ca/doc/programming/ihstyle.ps

[9] "comp.lang.c Frequently Asked Questions", http://www.eskimo.com/~scs/C-faq/top.html

[10] "GNU coding standards", Free Software Foundation, http://www.gnu.org/prep/standards/

[11] "The C Programming Language, Second Edition", Brian W. Kernighan and Dennis M. Ritchie, ISBN 0-13-110362-8, Prentice Hall PTR, Mar. 1988

[12] "Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs", Steve Maguire, ISBN 1-55615-551-4, Microsoft Press, May. 1993

[13] "The Practice of Programming", Brian W. Kernighan and Rob Pike, ISBN 0-201-61586-X, Addison-Wesley Professional, Feb. 1999

[14] "Linux kernel coding style", http://www.kernel.org/doc/Documentation/CodingStyle

[15] "C Style: Standards and Guidelines: Defining Programming Standards for Professional C Programmers", David Straker, ISBN 0-1311-6898-3, Prentice Hall, Jan. 1992

[16] "C Programming FAQs:Frequently Asked Questions", Addison-Wesley Professional, Nov. 1995. ISBN 9780201845198, Steve Summit

[17] "C STYLE GUIDE (SOFTWARE ENGINEERING LABORATORY SERIES SEL-94-003)", NASA, Aug. 1994, http://sel.gsfc. nasa.gov/website/documents/online-doc/94-003.pdf

[18] "The CERT® C Secure Coding Standard", Robert C. Seacord, ISBN 978-0321563217, Addison-Wesley Professional, Oct. 2008

Ver.1.1 Authors and editorss

AOKI Nao	IPA/SEC
ENDO Arisa	IPA/SEC
ENDOU Ryuji	Mitsubishi Space Software Co., Ltd.
FURUYAMA Hisaki	Matsushita Electric Industrial Co., Ltd.
FUTAGAMI Takao	TOYO Corporation
HACHIYA Shouichi	GAIA System Solutions Inc.
HAYASHIDA Seiji	TOSHIBA CORPORATION
HIRAYAMA Masayuki	IPA/SEC
MITSUHASHI Fusako	NEC Corporation
MURO Shuji	IPA/SEC
NAMIKI Rieko	OGIS-RI Co., Ltd.
OHNO Katsumi	IPA/SEC
OHSHIMA Kenji	Ricoh Company, Ltd.
SHISHIDO Fumio	eSOL Co., Ltd.
UEDA Naoko	Fujitsu Limited
UNO Musubi	Matsushita Electric Industrial Co., Ltd.

Ver. 2.0 Authors and editors

FUTAGAMI Takao	TOYO Corporation
ITOH Masako	Fujitsu Limited
MIHARA Yukihiro	IPA/SEC
MITSUHASHI Fusako	NEC Corporation
NISHIYAMA HIroyasu	Hitachi, Ltd.
SHUKUGUCHI Masahiro	eSOL Co., Ltd.
TACHI Nobuyuki	Nagoya University
TOYAMA Keisuke	IPA/SEC

(Organizational affiliations are as of the publication of Japanese edition.)

Contributers to English translation version

OBATA Hiromi	IPA/SEC
OOIZUMI Yuga	IPA/SEC
SHIMIZU Tatsuo	Shimizu International
TAKEICHI Sachie	IPA/SEC
TOYAMA Keisuke	IPA/SEC

ESCR

【Revised edition】

Embedded System development Coding Reference guide [C language edition]

Ver.2.0

July 24, 2014
March 24, 2017

Written and edited by Software Reliability Enhancement Center,
Technology Headquarters,Information-technology Promotion Agency, Japan

http://www.ipa.go.jp/english/sec/