IAR Embedded Workbench®
MISRA C:2004

Reference Guide

EDITION NOTICE
Third edition: March 2008

Part number: EWMISRAC:2004-1

This guide describes version 1.0 of the IAR Systems implementation of The Motor Industry Software Reliability Association's *Guidelines for the Use of the C Language in Critical Systems* (the MISRA-C:2004 standard), including the *MISRA-C:2004 Technical Corrigendum 1*, dated 17 July, 2007.

Internal reference: IJOA

Contents

Preface

Welcome to the IAR Embedded Workbench® MISRA C:2004 Reference Guide. This guide includes gives reference information about the IAR Systems implementation of The Motor Industry Software Reliability Association's *Guidelines for the Use of the C Language in Critical Systems*.

Who should read this guide

You should read this guide if you are developing a software product using the MISRA-C:2004 rules. In addition, you should have a working knowledge of:

- The C programming language
- The MISRA C subset of the C language
- Application development for safety-critical embedded systems
- The architecture and instruction set of your microcontroller (refer to the chip manufacturer's documentation)
- The operating system of your host machine.

What this guide contains

Below is a brief outline and summary of the chapters in this guide.

- *Introduction* explains the benefits of using MISRA C and gives an overview of the IAR Systems implementation.
- *General IDE options* describes the general MISRA C options in the IAR Embedded Workbench IDE.
- *Compiler IDE options* describes the MISRA C compiler options in the IAR Embedded Workbench IDE.
- *Command line options* explains how to set the options from the command line.
- *MISRA C:2004 rules reference* describes how IAR Systems has interpreted and implemented the rules given in *Guidelines for the Use of the C Language in Critical Systems*, including the *MISRA-C:2004 Technical Corrigendum 1*, dated 17 July, 2007.

Other documentation

The complete set of IAR development tools are described in a series of guides. For information about:

- Using the IAR Embedded Workbench® and the IAR C-SPY® Debugger, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the IAR C/C++ Compiler, refer to the *IAR C/C++ Compiler Reference Guide* or the *IAR C/C++ Development Guide*
- Programming for the IAR Assembler, refer to the *IAR Assembler Reference Guide*
- Using the IAR linker and library tools, refer to the *IAR Linker and Library Tools Reference Guide* or the *IAR C/C++ Development Guide*
- Using the MISRA C 1998 rules, refer to the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*
- Using the runtime library, refer to the *Library Reference information*, available in the IAR Embedded Workbench IDE online help system.

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them are also delivered as printed books.

Recommended websites:

- The MISRA website, **www.misra.org.uk**, contains information and news about the MISRA C rules.
- The IAR website, **www.iar.com**, holds application notes and other product information.

Document conventions

This book uses the following typographic conventions:

Style	Used for
`computer`	Text that you type or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
`{option}`	A mandatory part of a command.
`a|b|c`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.

Table 1: Typographic conventions used in this guide

Style	Used for
reference	A cross-reference within this guide or to another guide.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.

Table 1: Typographic conventions used in this guide (Continued)

Introduction

The Motor Industry Software Reliability Association's *Guidelines for the Use of the C Language in Critical Systems* describe a subset of C intended for developing safety-critical systems.

This chapter describes the IAR Systems implementation for checking that a software project complies with the MISRA C rules.

Using MISRA C

C is arguably the most popular high-level programming language for embedded systems, but when it comes to developing code for safety-critical systems, the language has many drawbacks. There are several unspecified, implementation-defined, and undefined aspects of the C language that make it unsuited for use when developing safety-critical systems.

The MISRA C guidelines are intended to help you to overcome these weaknesses in the C language.

CLAIMING COMPLIANCE

To claim compliance with the MISRA C guidelines for your product, you must demonstrate that:

- A compliance matrix has been completed demonstrating how each rule is enforced.
- All C code in the product is compliant with the MISRA C rules or subject to documented deviations.
- A list of all instances where rules are not being followed is maintained, and for each instance there is an appropriately signed-off documented deviation.
- You have taken appropriate measures in the areas of training, style guide, compiler selection and validation, checking tool validation, metrics, and test coverage, as described in section 4.2 of *Guidelines for the Use of the C Language in Critical Systems*.

Implementation and interpretation of the MISRA C rules

The implementation of the MISRA C rules does not affect code generation, and has no significant effect on the performance of IAR Embedded Workbench. No changes have been made to the IAR CLIB or DLIB runtime libraries.

Note: The rules apply to the source code of the applications that you write and not to the code generated by the compiler. For example, rule 17.4 is interpreted to mean that you as a programmer cannot explicitly use any other pointer arithmetic than array indexing, but the resulting compiler-generated arithmetic is not considered to be a deviation from the rule.

CHECKING THE RULES

The compiler and linker only generate error messages, they do not actually prevent you from breaking the rules you are checking for. You can enable or disable individual rules for the entire project or at file level. A log is produced at compile and link time, and displayed in the Build Message window of the IAR Embedded Workbench IDE. This log can be saved to a file, as described in the *IAR Embedded Workbench User Guide*.

A message is generated for every deviation from a required or advisory rule, unless you have disabled it. Each message contains a reference to the MISRA C rule deviated from. The format of the reference is as in the following error message:

```
Error[Pm088]: pointer arithmetics should not be used
(MISRA C 2004 rule 17.4)
```

Note: The numbering of the messages does not match the rule numbering.

For each file being checked with MISRA C enabled, you can generate a full report containing a list of:

- All enabled MISRA C rules
- All MISRA C rules that are actually checked.

Manual checking

There are several rules that require manual checking. These are, for example, rules requiring knowledge of your intentions as a programmer or rules that are impractical to check statically, requiring excessive computations.

Note: The fact that rule 3.6 is not enforced means that standard header files in a project are not checked for compliance.

Documenting deviations

A deviation from a MISRA C rule is an instance where your application does not follow the rule. If you document a deviation from a rule, you can disable the warning for violations of that particular rule.

Note: Your source code can deviate from a rule as long as the reason is clearly documented. Because breaking rules in a controlled fashion is permitted according to the MISRA C guidelines, error messages can be explicitly disabled using the `#pragma diag_xxx` directives.

In addition, each rule is checked in its own right; no assumptions are made regarding what other rules are in effect, as these may have been disabled for this particular piece of code.

Enabling MISRA C rules

In the IAR Embedded Workbench IDE, you enable the MISRA C rules checking by choosing **Project>Options>General Options** and using the options on the **MISRA C 2004** page.

From the command line, use the option `--misrac2004` to enable the MISRA C 2004 rules checking.

General IDE options

This chapter describes the general MISRA C 2004 options in the IAR Embedded Workbench® IDE.

For information about how options can be set, see the *IAR Embedded Workbench® IDE User Guide*.

MISRA C 2004

Use the options on the **MISRA C 2004** page to control how the IAR Embedded Workbench IDE checks the source code for deviations from the MISRA C rules. The settings will be used for both the compiler and the linker.

If you want the compiler to check a different set of rules than the linker, you can override these settings in the **C/C++ Compiler** category of options.

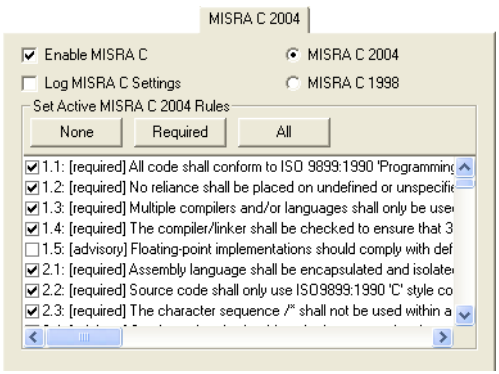

Figure 1: MISRA C 2004 general options

ENABLE MISRA C

Select this option to enable checking the source code for deviations from the MISRA C rules during compilation and linking. Only the rules you select in the scroll list will be checked.

LOG MISRA C SETTINGS

Select this option to generate a log during compilation and linking. This log is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

MISRA C 2004

Select this option to check for compliance with the MISRA-C:2004 standard.

MISRA C 1998

If you want to check for compliance with the older MISRA-C:1998 standard, select this option and use the settings on the **MISRA C 1998** page instead. When this option is selected, the list of MISRA-C:2004 rules becomes unavailable.

SET ACTIVE MISRA C 2004 RULES

Select the checkboxes for the rules in the scroll list that you want the compiler and linker to check during compilation and linking. You can use the buttons **None**, **Required**, or **All** to select or deselect several rules with one click:

None	Deselects all rules.
Required	Selects all rules that are categorized by the *Guidelines for the Use of the C Language in Critical Systems* as *required* and deselects the rules that are categorized as *advisory*
All	Selects all rules.

Compiler IDE options

This chapter describes the MISRA C:2004 compiler options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see the *IAR Embedded Workbench® IDE User Guide*.

MISRA C 2004

Use these options to override the options set on the **General Options>MISRA C 2004** page. This means that the compiler will check for a different set of rules than the linker.

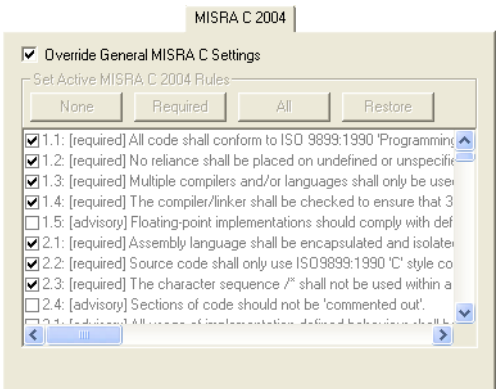

Figure 2: MISRA C 2004 compiler options

OVERRIDE GENERAL MISRA C SETTINGS

Select this option if you want the compiler to check a different selection of rules than the rules selected in the **General Options** category.

SET ACTIVE MISRA C 2004 RULES

Select the checkboxes for the rules in the scroll list that you want the compiler to check during compilation. You can use the buttons **None**, **Required**, **All**, or **Restore** to select or deselect several rules with one click:

None Deselects all rules.

Required Selects all rules that are categorized by the *Guidelines for the Use of the C Language in Critical Systems* as *required* and deselects the rules that are categorized as *advisory*

All Selects all rules.

Restore Restores the MISRA C 2004 settings used in the **General Options** category.

Note: This list is only available when both the options **Enable MISRA C** and **MISRA C 2004** have been selected on the **MISRA C 2004** page of the **General Options** category.

Command line options

This chapter describes how to set the MISRA C options from the command line, and gives reference information about each option.

Options summary

The following table summarizes the command line options:

Command line option	Description
`--misrac2004`	Enables error messages specific to MISRA C 2004
`--misrac_verbose`	Enables verbose logging of MISRA C checking

Table 2: Command line options summary

Descriptions of options

This section gives detailed reference information about each command line option.

--misrac2004

Syntax

`--misrac2004={`*`range1`*`,[~]`*`range2`*`,[~]`*`range3`*`,…}`

Parameters

`range` *`range`* can be one of:

- `all` = all MISRA-C:2004 rules
- `required` = all MISRA-C:2004 rules categorized as *required*
- the number of a group of rules
- the number of a rule
- a continuous sequence starting with a rule or a group and ending with a rule or a group, separated by a `-` (dash)

`~` excludes the following *`range`* from checking

Description

Use this option to enable checking for deviations from the rules described in the MISRA *Guidelines for the Use of the C Language in Critical Systems*.

If a rule cannot be checked, specifying the option for that rule has no effect. For instance, MISRA-C:2004 rule 3.2 is a documentation issue, and the rule is not checked. As a consequence, specifying `--misrac2004=3.2` has no effect.

Note: MISRA-C:2004 is not supported by all IAR Systems products. If MISRA-C:2004 checking is not supported, using this option will generate an error.

Examples

This command line checks all rules from rule 1.1 to rule 6.3, and all rules from rule 7.1 to rule 20.11:

```
--misrac2004=1-6.3,7-20.11
```

This command line checks rule 3.3 and all *required* rules except rule 10.3 and the rule in group 21:

```
--misrac2004=required,3.3,~10.3,~21
```

This command line checks *all* rules except rule 20.8:

```
--misrac2004=all,~20.8
```

To set the equivalent option in the IAR Embedded Workbench IDE, select **Project>Options>General Options>MISRA C 2004** or **Project>Options>C/C++ Compiler>MISRA C 2004**.

--misrac_verbose

Syntax

```
--misrac_verbose
```

Description

Use this option to generate a MISRA C log during compilation and linking. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

If this option is enabled, a text is displayed at sign-on that shows both enabled and checked MISRA C rules.

To set the equivalent option in the IAR Embedded Workbench IDE, select **Project>Options>General Options>MISRA C 2004**.

MISRA C:2004 rules reference

This chapter describes how IAR Systems has interpreted and implemented the rules given in *Guidelines for the Use of the C Language in Critical Systems* to enforce measures for stricter safety in the ISO standard for the C programming language [ISO/IEC 9899:1990].

The IAR Systems implementation is based on the standard MISRA-C:2004, dated October 2004, with the clarifications of the MISRA-C:2004 Technical Corrigendum 1, dated 17 July, 2007.

Summary of rules

These tables list all MISRA-C:2004 rules.

GROUP 1: ENVIRONMENT

No	Rule	Type	Category
1.1	All code shall conform to ISO 9899:1990 Programming languages – C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	Environment	Required
1.2	No reliance shall be placed on undefined or unspecified behavior.	Environment	Required
1.3	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.	Environment	Required
1.4	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	Environment	Required
1.5	Floating-point implementations should comply with a defined floating-point standard.	Environment	Advisory

Table 3: MISRA C 2004 Environment rules summary

GROUP 2: LANGUAGE EXTENSIONS

No	Rule	Type	Category
2.1	Assembler language shall be encapsulated and isolated.	Language extensions	Required
2.2	Source code shall only use /* ... */ style comments.	Language extensions	Required
2.3	The character sequence /* shall not be used within a comment.	Language extensions	Required
2.4	Sections of code should not be commented out.	Language extensions	Advisory

Table 4: MISRA C 2004 Language extensions rules summary

GROUP 3: DOCUMENTATION

No	Rule	Type	Category
3.1	All usage of implementation-defined behavior shall be documented.	Documentation	Required
3.2	The character set and the corresponding encoding shall be documented.	Documentation	Required
3.3	The implementation of integer division in the chosen compiler should be determined, documented, and taken into account.	Documentation	Advisory
3.4	All uses of the #pragma directive shall be documented and explained.	Documentation	Required
3.5	If it is being relied upon, the implementation-defined behavior and packing of bitfields shall be documented.	Documentation	Required
3.6	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	Documentation	Required

Table 5: MISRA C 2004 Documentation rules summary

GROUP 4: CHARACTER SETS

No	Rule	Type	Category
4.1	Only those escape sequences that are defined in the ISO C standard shall be used.	Character sets	Required
4.2	Trigraphs shall not be used.	Character sets	Required

Table 6: MISRA C 2004 Character sets rules summary

GROUP 5: IDENTIFIERS

No	Rule	Type	Category
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	Identifiers	Required
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Identifiers	Required
5.3	A typedef name shall be a unique identifier.	Identifiers	Required
5.4	A tag name shall be a unique identifier.	Identifiers	Required
5.5	No object or function identifier with static storage duration should be reused.	Identifiers	Advisory
5.6	No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names.	Identifiers	Advisory
5.7	No identifier name should be reused.	Identifiers	Advisory

Table 7: MISRA C 2004 Identifiers rules summary

GROUP 6: TYPES

No	Rule	Type	Category
6.1	The plain char type shall be used only for the storage and use of character values.	Types	Required
6.2	signed and unsigned char type shall be used only for the storage and use of numeric values.	Types	Required
6.3	`typedef`s that indicate size and signedness should be used in place of the basic types.	Types	Advisory
6.4	Bitfields shall only be defined to be of type unsigned int or signed int.	Types	Required
6.5	Bitfields of signed type shall be at least 2 bits long.	Types	Required

Table 8: MISRA C 2004 Types rules summary

GROUP 7: CONSTANTS

No	Rule	Type	Category
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	Constants	Required

Table 9: MISRA C 2004 Constants rules summary

GROUP 8: DECLARATIONS AND DEFINITIONS

No	Rule	Type	Category
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	Declarations and definitions	Required
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.	Declarations and definitions	Required
8.3	For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical.	Declarations and definitions	Required
8.4	If objects or functions are declared more than once, their types shall be compatible.	Declarations and definitions	Required
8.5	There shall be no definitions of objects or functions in a header file.	Declarations and definitions	Required
8.6	Functions shall be declared at file scope.	Declarations and definitions	Required
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.	Declarations and definitions	Required
8.8	An external object or function shall be declared in one and only one file.	Declarations and definitions	Required
8.9	An identifier with external linkage shall have exactly one external definition.	Declarations and definitions	Required
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	Declarations and definitions	Required
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	Declarations and definitions	Required
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.	Declarations and definitions	Required

Table 10: MISRA C 2004 Declarations and definitions rules summary

GROUP 9: INITIALIZATION

No	Rule	Type	Category
9.1	All automatic variables shall have been assigned a value before being used.	Initialization	Required
9.2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	Initialization	Required
9.3	In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Initialization	Required

Table 11: MISRA C 2004 Initialization rules summary

GROUP 10: ARITHMETIC TYPE CONVERSIONS

No	Rule	Type	Category
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.	Arithmetic type conversions	Required
10.2	The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.	Arithmetic type conversions	Required
10.3	The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression.	Arithmetic type conversions	Required
10.4	The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.	Arithmetic type conversions	Required

Table 12: MISRA C 2004 Arithmetic type conversions rules summary

No	Rule	Type	Category
10.5	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	Arithmetic type conversions	Required
10.6	A U suffix shall be applied to all constants of unsigned type.	Arithmetic type conversions	Required

Table 12: MISRA C 2004 Arithmetic type conversions rules summary (Continued)

GROUP 11: POINTER TYPE CONVERSIONS

No	Rule	Type	Category
11.1	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	Pointer type conversions	Required
11.2	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type, or a pointer to void.	Pointer type conversions	Required
11.3	A cast should not be performed between a pointer type and an integral type.	Pointer type conversions	Advisory
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	Pointer type conversions	Advisory
11.5	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	Pointer type conversions	Required

Table 13: MISRA C 2004 Pointer type conversions rules summary

GROUP 12: EXPRESSIONS

No	Rule	Type	Category
12.1	Limited dependence should be placed on the C operator precedence rules in expressions.	Expressions	Advisory
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	Expressions	Required
12.3	The sizeof operator shall not be used on expressions that contain side effects.	Expressions	Required

Table 14: MISRA C 2004 Expressions rules summary

No	Rule	Type	Category
12.4	The right-hand operand of a logical && or || operator shall not contain side effects.	Expressions	Required
12.5	The operands of a logical && or || shall be primary expressions.	Expressions	Required
12.6	The operands of logical operators (&&, ||, and !) should be effectively boolean. Expressions that are effectively boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).	Expressions	Advisory
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.	Expressions	Required
12.8	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	Expressions	Required
12.9	Trigraphs shall not be used.The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	Expressions	Required
12.10	The comma operator shall not be used.	Expressions	Required
12.11	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	Expressions	Advisory
12.12	The underlying bit representations of floating-point values shall not be used.	Expressions	Required
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	Expressions	Advisory

Table 14: MISRA C 2004 Expressions rules summary (Continued)

GROUP 13: CONTROL STATEMENT EXPRESSIONS

No	Rule	Type	Category
13.1	Assignment operators shall not be used in expressions that yield a boolean value.	Control statement expressions	Required
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively boolean.	Control statement expressions	Advisory
13.3	Floating-point expressions shall not be tested for equality or inequality.	Control statement expressions	Required
13.4	The controlling expression of a for statement shall not contain any objects of floating type.	Control statement expressions	Required

Table 15: MISRA C 2004 Control statement expressions rules summary

No	Rule	Type	Category
13.5	The three expressions of a for statement shall be concerned only with loop control.	Control statement expressions	Required
13.6	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	Control statement expressions	Required
13.7	Boolean operations whose results are invariant shall not be permitted.	Control statement expressions	Required

Table 15: MISRA C 2004 Control statement expressions rules summary (Continued)

GROUP 14: CONTROL FLOW

No	Rule	Type	Category
14.1	There shall be no unreachable code.	Control flow	Required
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change.	Control flow	Required
14.3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character.	Control flow	Required
14.4	The goto statement shall not be used.	Control flow	Required
14.5	The continue statement shall not be used.	Control flow	Required
14.6	For any iteration statement, there shall be at most one break statement used for loop termination.	Control flow	Required
14.7	A function shall have a single point of exit at the end of the function.	Control flow	Required
14.8	The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.	Control flow	Required
14.9	An if expression construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement or another if statement.	Control flow	Required
14.10	All if ... else if constructs shall be terminated with an else clause.	Control flow	Required

Table 16: MISRA C 2004 Control flow rules summary

GROUP 15: SWITCH STATEMENTS

No	Rule	Type	Category
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Switch statements	Required
15.2	An unconditional break statement shall terminate every non-empty switch clause.	Switch statements	Required
15.3	The final clause of a switch statement shall be the default clause.	Switch statements	Required
15.4	A switch expression shall not represent a value that is effectively boolean.	Switch statements	Required
15.5	Every switch statement shall have at least one case clause.	Switch statements	Required

Table 17: MISRA C 2004 Switch statements rules summary

GROUP 16: FUNCTIONS

No	Rule	Type	Category
16.1	Functions shall not be defined with a variable number of arguments.	Functions	Required
16.2	Functions shall not call themselves, either directly or indirectly.	Functions	Required
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Functions	Required
16.4	The identifiers used in the declaration and definition of a function shall be identical.	Functions	Required
16.5	Functions with no parameters shall be declared and defined with the parameter list void.	Functions	Required
16.6	The number of arguments passed to a function shall match the number of parameters.	Functions	Required
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	Functions	Advisory
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Functions	Required

Table 18: MISRA C 2004 Functions rules summary

No	Rule	Type	Category
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Functions	Required
16.10	If a function returns error information, then that error information shall be tested.	Functions	Required

Table 18: MISRA C 2004 Functions rules summary (Continued)

GROUP 17: POINTERS AND ARRAYS

No	Rule	Type	Category
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointers and arrays	Required
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Pointers and arrays	Required
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Pointers and arrays	Required
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Pointers and arrays	Required
17.5	The declaration of objects should contain no more than two levels of pointer indirection.	Pointers and arrays	Advisory
17.6	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	Pointers and arrays	Required

Table 19: MISRA C 2004 Pointers and arrays rules summary

GROUP 18: STRUCTURES AND UNIONS

No	Rule	Type	Category
18.1	All structure and union types shall be complete at the end of the translation unit.	Structures and unions	Required
18.2	An object shall not be assigned to an overlapping object.	Structures and unions	Required
18.3	An area of memory shall not be used for unrelated purposes.	Structures and unions	Required
18.4	Unions shall not be used.	Structures and unions	Required

Table 20: MISRA C 2004 Structures and unions rules summary

GROUP 19: PREPROCESSING DIRECTIVES

No	Rule	Type	Category
19.1	`#include` statements in a file should only be preceded by other preprocessor directives or comments.	Preprocessing directives	Advisory
19.2	Non-standard characters should not occur in header file names in #include directives.	Preprocessing directives	Advisory
19.3	The #include directive shall be followed by either a <filename> or "filename" sequence.	Preprocessing directives	Required
19.4	C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Preprocessing directives	Required
19.5	Macros shall not be #define'd or #undef'd within a block.	Preprocessing directives	Required
19.6	#undef shall not be used.	Preprocessing directives	Required
19.7	A function should be used in preference to a function-like macro.	Preprocessing directives	Advisory
19.8	A function-like macro shall not be invoked without all of its arguments.	Preprocessing directives	Required
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Preprocessing directives	Required
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Preprocessing directives	Required
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	Preprocessing directives	Required
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	Preprocessing directives	Required
19.13	The # and ## preprocessor operators should not be used.	Preprocessing directives	Advisory
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	Preprocessing directives	Required

Table 21: MISRA C 2004 Preprocessing directives rules summary

No	Rule	Type	Category
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Preprocessing directives	Required
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	Preprocessing directives	Required
19.17	All #else, #elif, and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	Preprocessing directives	Required

Table 21: MISRA C 2004 Preprocessing directives rules summary (Continued)

GROUP 20: STANDARD LIBRARIES

No	Rule	Type	Category
20.1	Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined.	Standard libraries	Required
20.2	The names of Standard Library macros, objects, and functions shall not be reused.	Standard libraries	Required
20.3	The validity of values passed to library functions shall be checked.	Standard libraries	Required
20.4	Dynamic heap memory allocation shall not be used.	Standard libraries	Required
20.5	The error indicator errno shall not be used.	Standard libraries	Required
20.6	The macro offsetof in the stddef.h library shall not be used.	Standard libraries	Required
20.7	The setjmp macro and the longjmp function shall not be used.	Standard libraries	Required
20.8	The signal handling facilities of signal.h shall not be used.	Standard libraries	Required
20.9	The input/output library stdio.h shall not be used in production code.	Standard libraries	Required
20.10	The functions atof, atoi, and atol from the library stdlib.h shall not be used.	Standard libraries	Required
20.11	The functions abort, exit, getenv, and system from the library stdlib.h shall not be used.	Standard libraries	Required
20.12	The time handling functions of time.h shall not be used.	Standard libraries	Required

Table 22: MISRA C 2004 Standard libraries rules summary

GROUP 21: RUNTIME FAILURES

No	Rule	Type	Category
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: a. static analysis tools/techniques b. dynamic analysis tools/techniques c. explicit coding of checks to handle runtime faults.	Runtime failures	Required

Table 23: MISRA C 2004 Runtime failures rules summary

Environment rules

The rules in this section are concerned with the language environment.

Rule 1.1 (required)

All code shall conform to ISO 9899:1990 *Programming languages – C*, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the compiler is configured (using command line options or IDE options) to:

- compile with IAR extensions
- compile C++ code.

Note: The compiler does not generate this error if you use IAR extensions in your source code by means of a pragma directive.

Examples of rule violations

```
int16_t __far my_far_variable;
int16_t port @ 0xBEEF;
```

Example of correct code

```
#pragma location=0xBEEF
int16_t port;
```

Rule 1.2 (required) No reliance shall be placed on undefined or unspecified behavior.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 1.3 (required) Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 1.4 (required) The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

How the rule is checked

All IAR Systems compilers and linkers adhere to this rule. This rule is always followed.

Rule 1.5 (advisory) Floating-point implementations should comply with a defined floating-point standard.

How the rule is checked

All IAR Systems compilers and runtime libraries comply with the IEEE 754 floating-point standard. This rule is always followed.

Language extensions

The rules in this section are concerned with how extensions to the C language can be used.

Rule 2.1 (required) Assembler language shall be encapsulated and isolated.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 2.2 (required) Source code shall only use `/* ... */` style comments.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if C++ style comments (`//`) are used in your source code.

Rule 2.3 (required) The character sequence `/*` shall not be used within a comment.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if `/*` is used inside a comment.

Rule 2.4 (advisory) Sections of code should not be commented out.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever a comment ends with `;`, `{`, or `}`.

Note: This rule is checked in such a manner that code samples inside comments are allowed and do not generate an error.

Documentation

The rules in this section are concerned with documentation issues.

Rule 3.1 (advisory) All usage of implementation-defined behavior shall be documented.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 3.2 (required) The character set and the corresponding encoding shall be documented.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker.

Rule 3.3 (advisory) The implementation of integer division in the chosen compiler should be determined, documented, and taken into account.

How the rule is checked

This is implementation-defined behavior. For all IAR Systems compilers, the sign of the remainder on integer division is the same as the sign of the dividend, as documented in the *IAR C/C++ Compiler Reference Guide*.

Rule 3.4 (required) All uses of the `#pragma` directive shall be documented and explained.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 3.5 (required) If it is being relied upon, the implementation-defined behavior and packing of bitfields shall be documented.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker.

Note: See the *IAR C/C++ Compiler Reference Guide* or *IAR C/C++ Development Guide* for a description of how bitfields are stored in memory.

Rule 3.6 (required) All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Character sets

The rules in this section are concerned with how character sets can be used.

Rule 4.1 (required)

Only those escape sequences that are defined in the ISO C standard shall be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if any of the following are read inside a string or character literal:

- A character with an ASCII code outside the ranges 32–35, 37–63, 65–95, and 97–126
- An escape sequence that is not one of: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\'`, `\"`, `\\`, or `\0`.

Note: `$` (dollar), `@` (at), and `` ` `` (backquote) are not part of the source character set.

Examples of rule violations

```
"Just my $0.02"
"Just my £0.02"
```

Examples of correct code

```
"Hello world!\n"
'\n'
```

Note: This rule aims to restrict undefined behavior and implementation-defined behavior. The implementation-defined behavior applies only when characters are converted to internal representation, which only applies to character constants and string literals. For that reason, the IAR Systems implementation restricts the usage of characters only within character literals and string literals; characters within comments are not restricted.

Rule 4.2 (required)

Trigraphs shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a trigraph is used.

Examples of rule violations

```
SI_16 a ??( 3 ??);
STRING sic = "??(sic??)";
```

Example of correct code

```
STRING str = "What???";
```

Identifiers

The rules in this section are concerned with identifiers used in the code.

Rule 5.1 (required) Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, in a declaration or definition of an identifier if it has the same 31 initial characters as a previously declared or defined identifier.

The linker will generate an error, indicating a violation of this rule, if any identifiers have the same 31 initial characters.

Rule 5.2 (required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever a declaration or definition hides the name of another identifier.

Rule 5.3 (required) A `typedef` name shall be a unique identifier.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for:

- any declaration or definition that uses a name previously used as a `typedef`
- any `typedef` name previously used in a declaration or definition.

Rule 5.4 (required) A tag name shall be a unique identifier.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 5.5 (advisory) No object or function identifier with static storage duration should be reused.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 5.6 (advisory) No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 5.7 (advisory) No identifier name should be reused.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Types

The rules in this section are concerned with how data types are allowed to be declared.

Rule 6.1 (required) The plain `char` type shall be used only for the storage and use of character values.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 6.2 (required) `signed` and `unsigned char` type shall be used only for the storage and use of numeric values.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 6.3 (advisory) `typedefs` that indicate size and signedness should be used in place of the basic types.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if any of the basic types are used in a declaration or definition that is not a `typedef`.

Example of a rule violation

```
int x;
```

Example of correct code

```
typedef int SI_16
SI_16 x;
```

Note: The basic types are allowed in bitfields.

Rule 6.4 (required) Bitfields shall only be defined to be of type `unsigned int` or `signed int`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a bitfield is declared to have any type other than `unsigned int` or `signed int`.

Note: An error is given if a bitfield is declared to be of type `int` without using a `signed` or `unsigned` specifier.

Rule 6.5 (required) Bitfields of signed type shall be at least 2 bits long.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a bitfield of type `signed int` is declared to have size `0` or `1`.

Constants

The rule in this section is concerned with the use of constants.

Rule 7.1 (required) Octal constants (other than zero) and octal escape sequences shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever a non-zero constant starts with a 0.

Declarations and definitions

The rules in this section are concerned with declarations and definitions.

Rule 8.1 (required) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever:

- A non-static function is defined but there is no prototype visible at the point of definition
- A function pointer type with no prototype is used
- A non-prototype function is declared.

Example of a rule violation

```
void func();       /* Not a prototype */
```

Example of correct code

```
void func(void);
void func(void) { … }
```

Rule 8.2 (required) Whenever an object or function is declared or defined, its type shall be explicitly stated.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the type is missing.

Rule 8.3 (required) For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any function definition where the type given in the definition is not identical with the return type and the type of the parameters in the declaration. In particular, `typedef` types with different names are not considered identical and will generate an error.

Rule 8.4 (required) If objects or functions are declared more than once, their types shall be compatible.

How the rule is checked

The linker always checks for this, also when the MISRA C rules are disabled, and issues a warning. When the MISRA C rules are enabled, an error is issued instead.

The linker checks that declarations and definitions have compatible types, with these exceptions:

- `bool` and `wchar_t` are compatible with all `int` types of the same size.
- For parameters to Kernighan & Ritchie functions:
  - `int` and `unsigned int` are considered compatible
  - `long` and `unsigned long` are considered compatible.
- Incomplete types are considered compatible if they have the same name.
- Complete types are considered compatible if they have fields with compatible types.

Rule 8.5 (required) There shall be no definitions of objects or functions in a header file.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a function or variable is defined in a header file.

Note: The compiler will not generate an error when a variable is placed at an absolute address using the `@` operator or the `#pragma location` directive.

Rule 8.6 (required) Functions shall be declared at file scope.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, on encountering a function declaration at block scope.

Rule 8.7 (required) Objects shall be defined at block scope if they are only accessed from within a single function.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 8.8 (required) An external object or function shall be declared in one and only one file.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 8.9 (required) An identifier with external linkage shall have exactly one external definition.

How the rule is checked

The linker always checks for this, also when the MISRA C rules are disabled.

Note: Multiple definitions of global symbols are considered to be errors by the linker. The use of a symbol with no definition available is also considered to be a linker error.

Rule 8.10 (required) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

How the rule is checked

The linker will generate an error, indicating a violation of this rule, if a symbol is used in—and exported from—a module but not referenced from any other module.

Rule 8.11 (required) The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the `static` keyword is used in some but not all declarations and the definition.

Rule 8.12 (required) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the size of an array cannot be determined.

Example of a rule violation

```
extern int16_t array[];
```

Examples of correct code

```
int16_t array2[10];
int16_t array2[] = { 1, 2, 3 };
```

Initialization

The rules in this section are concerned with the initialization of variables.

Rule 9.1 (required) All automatic variables shall have been assigned a value before being used.

How the rule is checked

Partial support for checking this rule is available.

The compiler will generate an error, indicating a violation of this rule, if a variable is used but not previously assigned a value, but only if no execution path contains an assignment.

Rule 9.2 (required) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any initializer that does not have the correct brace structure and number of elements. The compiler will not generate an error if the initializer `{ 0 }` is used.

Examples of rule violations

```
struct { int16_t a,b; } a_struct = { 1 };
struct { int16_t a[3]; } a_struct = { 1, 2 };
```

Examples of correct code

```
struct { int16_t a,b; } a_struct = { 1, 2 };
struct { int16_t a,b; } a_struct = { 0 };
struct { int16_t a[3]; } a_struct = { 0 };
```

Rule 9.3 (required) In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if there are initializers for at least one of the enumeration constants, but:

- the first enumeration constant does not have an initializer, or
- the number of initializers is more than one but fewer than the number of enumeration constants.

Arithmetic type conversions

The rules in this section are concerned with type conversions and casts.

Internally, the compiler tracks the underlying type of all expressions, as described in section 6.10.4 of the *Guidelines for the Use of the C Language in Critical Systems*. The definition of a *complex expression* can be found in section 6.10.5.

Rule 10.1 (required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:

a. it is not a conversion to a wider integer type of the same signedness, or

b. the expression is complex, or

c. the expression is not constant and is a function argument, or

d. the expression is not constant and is a return expression.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for implicit integer conversions that do not comply with rule 10.1.

Rule 10.2 (required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if:

a. it is not a conversion to a wider floating type, or

b. the expression is complex, or

c. the expression is a function argument, or

d. the expression is a return expression.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for implicit floating-point conversions that do not comply with rule 10.2.

Rule 10.3 (required) The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for integer casts that do not comply with rule 10.3.

Rule 10.4 (required) The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for floating-point casts that do not comply with rule 10.4.

Rule 10.5 (required) If the bitwise operators ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the result of the ~ and << operators, when applied to the specified types, is not immediately cast to the underlying type.

Rule 10.6 (required) A `U` suffix shall be applied to all constants of `unsigned` type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a numeric integer constant without the `U` suffix is used in an unsigned operation.

Example of a rule violation

```
uint8_t uc = 10;
```

Examples of correct code

```
uint16_t ui = 0U;
...
ui = ui + 10U;

uint8_t uc;
...
int16_t i = uc + 1;      /* The + operation is performed with
                            type int. */

char ch = ´a´;           /* Not a numeric constant */
```

Pointer type conversions

The rules in this section are concerned with pointer type conversions and casts.

Rule 11.1 (required) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a pointer to a function is converted to any type other than an integral type.

Rule 11.2 (required) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type, or a pointer to `void`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a pointer to an object type is converted to any type other than the types specified in rule 11.2.

Rule 11.3 (advisory) A cast should not be performed between a pointer type and an integral type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a cast is performed between a pointer type and an integral type.

Rule 11.4 (advisory) A cast should not be performed between a pointer to object type and a different pointer to object type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a cast is performed between two different pointer to object types.

Rule 11.5 (required) A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a cast removes any `const` or `volatile` qualifications from a pointer type.

Expressions

The rules in this section are concerned with expressions.

Rule 12.1 (advisory) Limited dependence should be placed on the C operator precedence rules in expressions.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 12.2 (required) The value of an expression shall be the same under any order of evaluation that the standard permits.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for an expression if there are:

- multiple writes to a location without an intervening sequence point
- unordered reads and writes to or from the same location
- unordered accesses to a `volatile` location.

Note: An error is not generated for the expression `f() + f()`.

Rule 12.3 (required) The `sizeof` operator shall not be used on expressions that contain side effects.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the `sizeof` operator is applied to an expression containing either `++`, `--`, an assignment operator, or a function call.

Rule 12.4 (required) The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the right-hand side expression of an `&&` or `||` operator contains either `++`, `--`, an assignment operator, or a function call.

Rule 12.5 (required) The operands of a logical `&&` or `||` shall be *primary expressions*.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, unless both the left- and right-hand sides of a binary logical operator are either a single variable, a constant, or an expression in parentheses.

Note: No error is generated when the left- or right-hand expression is using the same logical operator. These are safe with respect to evaluation order and readability.

Examples of rule violations

```
a && b || c
a || b && c
a == 3 || b > 5
```

Examples of correct code

```
a && b && c
a || b || c
(a == 3) || (b > 5)
```

Rule 12.6 (advisory) The operands of logical operators (`&&`, `||`, and `!`) should be effectively boolean. Expressions that are effectively boolean should not be used as operands to operators other than (`&&`, `||`, `!`, `=`, `==`, `!=`, and `?:`).

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, in the following situations:

- If a bitwise operator is used in a boolean context
- If a logical operator is used in a non-boolean context.

A boolean context is:

- The top level of the controlling expression in an `if`, `while`, or `for` statement
- The top level of the first expression of an `?:` operator
- The top level of the left- or right-hand side of an `&&` or `||` operator.

Examples of rule violations

```
d = ( c & a ) && b;
d = a && b << c;
if ( ga & 1 ) { … }
```

Examples of correct code

```
d = a && b ? a : c;
d = ~a & b;
if ( (ga & 1) == 0 ) { … }
```

Rule 12.7 (required) Bitwise operators shall not be applied to operands whose underlying type is `signed`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a bitwise operator is applied to an operand whose underlying type is `signed`.

Rule 12.8 (required) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the right-hand side of a shift operator is an integer constant with a value exceeding the width of the left-hand type after integer promotion.

Specifically, for a signed 8-bit integer variable `i8`, the compiler will not generate an error when shifting 8 positions because the value of `i8` is promoted to `int` before the left-shift operator is applied, and therefore has a well-defined behavior.

Example of correct code

```
i8 = i8 >> 8;   /* i8 promoted to int */
```

Rule 12.9 (required) The unary minus operator shall not be applied to an expression whose underlying type is `unsigned`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if unary minus is applied to an expression with an `unsigned` type.

Rule 12.10 (required) The comma operator shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the comma operator is used.

Rule 12.11 (advisory) Evaluation of constant `unsigned` integer expressions should not lead to wrap-around.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the evaluation of a constant unsigned integer expression leads to wrap-around.

Rule 12.12 (required) The underlying bit representations of floating-point values shall not be used.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 12.13 (advisory) The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the `++` and `--` operators are mixed with other operators that perform side effects.

Note: Reading or writing `volatile` variables is not considered to be a side effect in this context, because it is not considered a violation to apply `++` and `--` to a `volatile` variable.

Control statement expressions

The rules in this section are concerned with expressions of control flow statements.

Rule 13.1 (required)

Assignment operators shall not be used in expressions that yield a boolean value.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any assignment operator appearing in a boolean context, that is:

- On the top level of the controlling expression in an `if`, `while`, or `for` statement
- In the first part of an `?:` operator
- On the top level of the left- or right-hand side of an `&&` or `||` operator.

Example of a rule violation

```
if (a = func()) {
  …
}
```

Rule 13.2 (advisory)

Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 13.3 (required)

Floating-point expressions shall not be tested for equality or inequality.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if `==` or `!=` is applied to a floating-point value.

Note: The compiler does not check the correctness of indirect equality tests such as `((x <=y) && (x >= y))`.

Rule 13.4 (required) The controlling expression of a `for` statement shall not contain any objects of floating type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the controlling expression of a `for` statement contains any floating-point expressions.

Rule 13.5 (required) The three expressions of a `for` statement shall be concerned only with loop control.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a `for` statement is not of this form:

- The first expression is a simple assignment, and
- The second expression is a simple test, and
- The third expression is some kind of update expression. This includes applying a variable to the pre- or post-increment and -decrement operation, any operation-assign operators, or the plain assign operator (provided that the right-hand side is not a constant).

In addition, these combinations are allowed:

- All three expressions exist
- The second and third expressions exist
- None of the expressions exist, indicating an infinite loop.

Rule 13.6 (required) Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 13.7 (required) Boolean operations whose results are invariant shall not be permitted.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, when a comparison cannot be true or is always true. However, the compiler does not detect when two or more expressions together cause an expression to be true or false.

Control flow

The rules in this section are concerned with the flow of the application code.

Rule 14.1 (required) There shall be no unreachable code.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, in any of the following cases:

- Code after a `goto` or `return`
- Code in a `switch` body, before the first label
- Code after an infinite loop (a loop with a constant controlling expression that evaluates to `true`)
- Code after a function call of a function that is known not to return
- Code after `break` in a `switch` clause
- Code after an `if` statement that is always taken where the end of the dependent statement is unreachable
- Code after an `if` statement where the ends of both dependent statements are unreachable
- Code after a `switch` statement where the ends of all clauses are unreachable.

Rule 14.2 (required) All non-null statements shall either have at least one side effect however executed, or cause control flow to change.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a statement does not contain a function call, an assignment, an operator with a side-effect (`++` and `--`), or an access to a volatile variable.

Example of a rule violation

```
v;      /* If 'v' is non-volatile */
```

Examples of correct code

```
do_stuff();
;       /* A null statement */
v;      /* If 'v' is volatile */
```

Rule 14.3 (required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for a null statement if the last *physical* line contains anything else than a single semicolon surrounded by whitespace. A comment may follow the semicolon as long as there is at least one whitespace character between the semicolon and the comment.

Rule 14.4 (required) The `goto` statement shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a `goto` statement is used.

Rule 14.5 (required) The `continue` statement shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a `continue` statement is used.

Rule 14.6 (required) For any iteration statement, there shall be at most one `break` statement used for loop termination.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if there are more than one `break` statement in a loop.

Rule 14.7 (required) A function shall have a single point of exit at the end of the function.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if there is a return statement anywhere else than at the end of a function.

Rule 14.8 (required) The statement forming the body of a `switch`, `while`, `do ... while`, or `for` statement shall be a compound statement.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the statements forming the body of the constructions in rule 14.8 is not a block.

Rule 14.9 (required) An `if` *`expression`* construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement or another `if` statement.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the statements forming the body of the constructions in rule 14.9 is not a block.

Rule 14.10 (required) All `if ... else if` constructs shall be terminated with an `else` clause.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an `if ... else if` construct is not terminated by an `else` clause.

Switch statements

The rules in this section are concerned with the allowed syntax of `switch` statements.

Rule 15.1 (required) A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a `case` label is not at the outermost block in the `switch` statement.

Rule 15.2 (required) An unconditional `break` statement shall terminate every non-empty `switch` clause.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any `case` clause that is not terminated by a `break` statement.

Note: An error will be generated even if the `case` statement is terminated with a `return` statement.

Rule 15.3 (required) The final clause of a `switch` statement shall be the `default` clause.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever a `switch` statement does not have a `default` label or the `default` label is not last in the `switch` statement.

Rule 15.4 (required) A `switch` expression shall not represent a value that is effectively boolean.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, in the following two cases:

- The controlling expression of a `switch` is the result of a comparison operator (equality or relational operator) or a logical operator (`&&`, `||`, or `!`)
- There is only one `case` label in the `switch` body.

Rule 15.5 (required) Every `switch` statement shall have at least one `case` clause.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a `switch` statement does not contain at least one `case` clause.

Functions

The rules in this section are concerned with the declaration and use of functions.

Rule 16.1 (required) Functions shall not be defined with a variable number of arguments.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever a function is declared, defined, or called using the ellipsis notation.

Note: No error is given for using `va_start`, `va_end`, or `va_arg` macros, because it is pointless to use them without using the ellipsis notation.

Rule 16.2 (required) Functions shall not call themselves, either directly or indirectly.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 16.3 (required) Identifiers shall be given for all of the parameters in a function prototype declaration.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, unless identifiers are given for all parameters.

Rule 16.4 (required) The identifiers used in the declaration and definition of a function shall be identical.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, unless the parameter identifiers of the declaration and the definition are equal.

Rule 16.5 (required) Functions with no parameters shall be declared and defined with the parameter list `void`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a function is declared or defined without a parameter list.

Example of a rule violation

```
void myfunc1();
```

Example of correct code

```
void myfunc1(void);
```

Rule 16.6 (required) The number of arguments passed to a function shall match the number of parameters.

How the rule is checked

The compiler always checks for this, also when the MISRA C rules are disabled.

Rule 16.7 (advisory) A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 16.8 (required) All exit paths from a function with non-`void` return type shall have an explicit `return` statement with an expression.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for non-`void` functions if:

- the end of a function can be reached and it does not contain a `return` statement
- a `return` statement does not have an expression.

Rule 16.9 (required) A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if:

- a function designator (a function name without parentheses) is used in the controlling expression of an `if`, `while`, or `for` statement
- a function designator is compared with `0` using either `==` or `!=`
- a function designator is used in a `void` expression.

Example of a rule violation

```
extern int func(void);
if ( func ) { … }
```

Example of correct code

```
extern int func(void);
if ( func() ) { … }
```

Rule 16.10 (required) If a function returns error information, then that error information shall be tested.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Pointers and arrays

The rules in this section are concerned with pointers and arrays.

Rule 17.1 (required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 17.2 (required) Pointer subtraction shall only be applied to pointers that address elements of the same array.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 17.3 (required) >, >=, <, <= shall not be applied to pointer types except where they point to the same array.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 17.4 (required) Array indexing shall be the only allowed form of pointer arithmetic.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a value is added to or subtracted from a pointer. An error is not issued if the `pointer[index]` notation is used.

Rule 17.5 (advisory) The declaration of objects should contain no more than two levels of pointer indirection.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if any type with more than two levels of indirection is used in a declaration or definition of an object or function.

Rule 17.6 (required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Structures and unions

The rules in this section are concerned with the specification and use of structures and unions.

Rule 18.1 (required) All structure and union types shall be complete at the end of the translation unit.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a field in a structure or a union is declared as an array without a size.

Rule 18.2 (required) An object shall not be assigned to an overlapping object.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 18.3 (required) An area of memory shall not be used for unrelated purposes.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 18.4 (required) Unions shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for a definition or declaration of a union.

Preprocessing directives

The rules in this section are concerned with include files and preprocessor directives.

Rule 19.1 (advisory) `#include` statements in a file should only be preceded by other preprocessor directives or comments.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an include directive is preceded by anything that is not a preprocessor directive or a comment.

Rule 19.2 (advisory) Non-standard characters should not occur in header file names in `#include` directives.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a header file name contains any non-standard character.

Rule 19.3 (required) The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an include directive is not followed by either `"` or <.

Rule 19.4 (required) C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 19.5 (required) Macros shall not be `#define`'d or `#undef`'d within a block.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a `#define` or `#undef` directive is used outside file-level scope.

Rule 19.6 (required) `#undef` shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an `#undef` directive is used.

Rule 19.7 (advisory) A function should be used in preference to a function-like macro.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 19.8 (required) A function-like macro shall not be invoked without all of its arguments.

How the rule is checked

The compiler always checks to see that the correct number of arguments are used, also when the MISRA C rules are disabled. The compiler will generate an error, indicating a violation of this rule, for a macro call where one or more arguments do not contain any tokens.

Example of a rule violation

```
MACRO(,)
```

Example of correct code

```
#define EMPTY
MACRO(EMPTY,EMPTY)
```

Rule 19.9 (required) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a preprocessing token with an initial `#` is used.

Note: No error is given for macros that are never expanded.

Rule 19.10 (required) In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a macro parameter is not enclosed in parentheses, unless the parameter is used as an operand of `#` or `##`.

Example of a rule violation

```
#define MY_MACRO_1(x) x + 2
```

Example of correct code

```
#define MY_MACRO_1(x) (x) + 2
#define MY_MACRO_2(x,y) x##y
```

Rule 19.11 (required) All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an undefined preprocessor symbol is used in an `#if` or `#elif` directive.

Rule 19.12 (required) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if more than one of # or ## is used in combination. For example, the occurrence of # and ## in the same macro definition will trigger an error.

Example of a rule violation

```
#define MY_MACRO(x) BAR(#x) ## _var
```

Examples of correct code

```
#define MY_MACRO(x) #x
#define MY_MACRO(x) my_ ## x
```

Rule 19.13 (advisory) The # and ## preprocessor operators should not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if # or ## is part of a macro definition.

Rule 19.14 (required) The `defined` preprocessor operator shall only be used in one of the two standard forms.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the result of expanding a macro in an expression controlling conditional inclusion, results in the `defined` unary operator.

Rule 19.15 (required) Precautions shall be taken in order to prevent the contents of a header file being included twice.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a construction similar to this one is not found in a header file:

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

Rule 19.16 (required) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an unknown preprocessor directive is found or if a standard preprocessor directive is used incorrectly.

Examples of rule violations

```
#ifdef FOO BAR

#else1

#else FOO

#endif FOO
```

Rule 19.17 (required) All `#else`, `#elif`, and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

How the rule is checked

The compiler always checks for this, also when the MISRA C rules are disabled.

Standard libraries

The rules in this section are concerned with the use of standard library functions.

Rule 20.1 (required) Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any `#define` (or `#undef`) used to define (or undefine) an object- or function-like macro with a name that is:

- a compiler predefined macro
- an object- or function-like macro defined in any standard header
- an object or function declared in any standard header.

Rule 20.2 (required) The names of Standard Library macros, objects, and functions shall not be reused.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any definition used for defining a macro, object, or function with a name that is already declared in a standard header. This regardless of whether the correct header file has been included or not.

Rule 20.3 (required) The validity of values passed to library functions shall be checked.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.

Rule 20.4 (required) Dynamic heap memory allocation shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any reference to functions named `malloc`, `realloc`, `calloc`, or `free`, even if the header file `stdlib.h` has not been included.

Rule 20.5 (required) The error indicator `errno` shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any reference to an object named `errno`, even if the header file `errno.h` has been included.

Rule 20.6 (required) The macro `offsetof` in the `stddef.h` library shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a macro with the name `offsetof` is expanded.

Note: Including the header file `stddef.h` does not, in itself, generate an error.

Rule 20.7 (required) The `setjmp` macro and the `longjmp` function shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any reference to a function named `setjmp` or `longjmp`; regardless of whether the header file `setjmp.h` is included.

Rule 20.8 (required) The signal handling facilities of `signal.h` shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the header file `signal.h` is included.

Rule 20.9 (required) The input/output library `stdio.h` shall not be used in production code.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the header file `stdio.h` has been included when `NDEBUG` is defined.

Rule 20.10 (required) The functions `atof`, `atoi`, and `atol` from the library `stdlib.h` shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any reference to a function named `atof`, `atoi`, or `atol`; regardless of whether the header file `stdlib.h` is included.

Rule 20.11 (required) The functions `abort`, `exit`, `getenv`, and `system` from the library `stdlib.h` shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any reference to a function named `abort`, `exit`, `getenv`, and `system`; regardless of whether the header file `stdlib.h` is included.

Rule 20.12 (required) The time handling functions of `time.h` shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the header file `time.h` has been included.

Runtime failures

The rule in this section is concerned with the minimization of runtime failures.

Rule 21.1 (required) Minimization of runtime failures shall be ensured by the use of at least one of:

a. static analysis tools/techniques

b. dynamic analysis tools/techniques

c. explicit coding of checks to handle runtime faults.

How the rule is checked

Violations of this rule are not checked for by the compiler or linker. This rule requires manual checking.