

IAR Embedded Workbench® MISRA C:1998

Reference Guide

COPYRIGHT NOTICE

© Copyright 2004–2008 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE

Third edition: February 2008

Part number: EWMISRAC1998-3

This guide describes version 1.0 of the IAR Systems implementation of The Motor Industry Software Reliability Association's legacy *Guidelines for the Use of the C Language in Vehicle Based Software*, also known as the MISRA C:1998 standard.

Contents

Preface	1
Who should read this guide	1
What this guide contains	1
Other documentation	2
Document conventions	2
Introduction	5
Using MISRA C	5
Claiming compliance	5
Implementation and interpretation of the MISRA C rules	6
Checking the rules	6
Enabling MISRA C rules	7
General IDE options	9
MISRA C 1998	9
Compiler IDE options	11
MISRA C 1998	11
Command line options	13
Options summary	13
Descriptions of options	13
MISRA C:1998 rules reference	15
Summary of rules	15
Environment rules	23
Character sets	24
Comments	26
Identifiers	26
Types	27
Constants	29
Declarations and definitions	29
Initialization	32

Operators	33
Conversions	36
Expressions	37
Control flow	39
Functions	43
Preprocessing directives	47
Pointers and arrays	51
Structures and unions	53
Standard libraries	54

Preface

Welcome to the IAR Embedded Workbench® MISRA C:1998 Reference Guide. This guide includes gives reference information about the IAR Systems implementation of The Motor Industry Software Reliability Association's legacy *Guidelines for the Use of the C Language in Vehicle Based Software*, also known as the MISRA C:1998 standard.

Who should read this guide

You should read this guide if you are developing a software product using the MISRA C:1998 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.

Refer to the *IAR C/EC++ Compiler Reference Guide* or the *IAR C/C++ Development Guide*, the *IAR Assembler Reference Guide*, and *IAR Linker and Library Tools Reference Guide* for more information about the other development tools incorporated in the IAR Embedded Workbench IDE.

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:1998 rules reference* describes how IAR Systems has interpreted and implemented the rules given in *Guidelines for the Use of the C Language in Vehicle Based Software*.

Other documentation

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

- Using the IAR Embedded Workbench® IDE 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/EC++ 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 2004 rules, refer to the *IAR Embedded Workbench® MISRA C:2004 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.
<i>parameter</i>	A label representing the actual value you should type as part of a command.
[option]	An optional part of a command.

Table 1: Typographic conventions used in this guide



Style	Used for
{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.
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 Vehicle Based Software* describe a subset of C intended for developing safety-critical systems, also called MISRA C:1998.

This chapter describes the IAR Systems implementation for checking that a software project complies with the MISRA C:1998 rules. IAR Systems also supports the later MISRA C 2004 standard.

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 5.2 of *Guidelines for the Use of the C Language in Vehicle Based Software*.

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 101 is interpreted to mean that you as a programmer may not explicitly use pointer arithmetic, but the compiler-generated arithmetic resulting from, e.g., `a[3]` 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 1998 rule 101)
```

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 116 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** and using the options on the **MISRA C 2004** page in the **General Options** category.



From the command line, use the option `--misrac1998` to enable the MISRA C rules checking.

General IDE options

This chapter describes the general MISRA C 1998 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 1998

Use the options on the **MISRA C 1998** page to control how IAR Embedded Workbench checks the source code for deviations from the MISRA C rules. The settings will be used for both the compiler and the linker.

Note: This list is only available when both the options **Enable MISRA C** and **MISRA C 1998** have been selected on the **MISRA C 2004** page. If you want a verbose log of the check, you must also select the option **Log MISRA C settings** on the **MISRA C 2004** page. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.

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

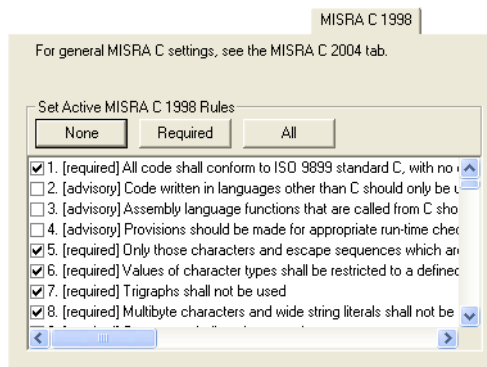


Figure 1: MISRA C 1998 general options

SET ACTIVE MISRA C 1998 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 Vehicle Based Software* as *required* and deselects the rules that are categorized as *advisory*

All Selects all rules.

Note: This list is only available when both the options **Enable MISRA C** and **MISRA C 1998** have been selected on the **MISRA C 2004** page.

Compiler IDE options

This chapter describes the MISRA C 1998 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 1998

Use these options to override the options set on the **General Options>MISRA C 1998** page.

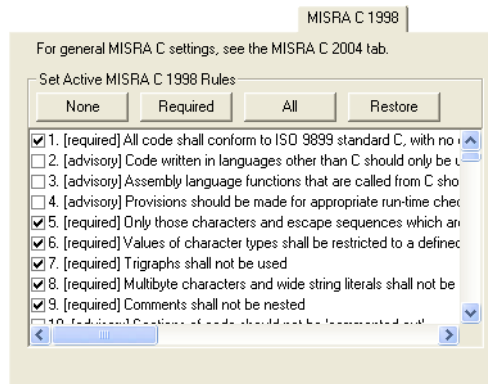


Figure 2: MISRA C 1998 compiler options

To make the compiler to check a different set of rules than the rules selected on the **General Options>MISRA C 1998** page, select the option **Override general MISRA C settings** on the **C/C++ Compiler>MISRA C 2004** page.

SET ACTIVE MISRA C 1998 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 Vehicle Based Software* as *required* and deselects the rules that are categorized as *advisory*
- All** Selects all rules.
- Restore** Restores the MISRA C 1998 settings used in the **General Options** category.

Note: This list is only available when both the options **Enable MISRA C** and **MISRA C 1998** have been selected on the **MISRA C 2004** page of the **General Options** category.

Command line options

This chapter describes how to set the 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
<code>--misrac1998</code>	Enables error messages specific to MISRA C:1998
<code>--misrac_verbose</code>	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.

`--misrac1998`

Syntax

```
--misrac1998 [= { tag1, tag2-tag3, ... | all | required }]
```

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 Vehicle Based Software*. By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C:1998 rules.

If a rule cannot be checked, specifying the option for that rule has no effect. For instance, MISRA C rule 15 is a documentation issue, and the rule is not checked. As a consequence, specifying `--misrac1998=15` has no effect.

MISRA C:1998 is not supported by all IAR Systems products. If MISRA C:1998 checking is not supported, using this option will generate an error.

Note: In some IAR Systems products, you must specify this option as `--misrac` instead, for reasons of backwards compatibility.

Parameters

<code>--misrac1998</code>	Enables checking for all MISRA C:1998 rules
<code>--misrac1998=n</code>	Enables checking for the MISRA C:1998 rule with number <i>n</i>

<code>--misrac1998=<i>m,n</i></code>	Enables checking for the MISRA C:1998 rules with numbers <i>m</i> and <i>n</i>
<code>--misrac1998=<i>k-n</i></code>	Enables checking for all MISRA C:1998 rules with numbers from <i>k</i> to <i>n</i>
<code>--misrac1998=<i>k,m,r-t</i></code>	Enables checking for MISRA C:1998 rules with numbers <i>k,m</i> , and from <i>r</i> to <i>t</i>
<code>--misrac1998=all</code>	Enables checking for all MISRA C:1998 rules
<code>--misrac1998=required</code>	Enables checking for all MISRA C:1998 rules categorized as <i>required</i>



To set related options in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>MISRA C 1998** or **Project>Options>C/C++ Compiler>MISRA C 1998**.

--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>Log MISRA C Settings**.

MISRA C:1998 rules reference

This chapter describes how IAR Systems has interpreted and implemented the rules given in the legacy *Guidelines for the Use of the C Language in Vehicle Based Software* (MISRA C:1998) 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 version 1 of the MISRA C rules, dated April 1998.

Summary of rules

The table below lists all MISRA C:1998 rules.

No	Rule	Type	Category
1	All code shall conform to ISO 9899 standard C, with no extensions permitted.	Environment	Required
2	Code written in languages other than C should only be used if there is a defined interface standard for object code to which the compiler/assemblers for both languages conform.	Environment	Advisory
3	Assembler language functions that are called from C should be written as C functions containing only inline assembler language, and inline assembler language should not be embedded in normal code.	Environment	Advisory
4	Provisions should be made for appropriate runtime checking.	Environment	Advisory
5	Only those characters and escape sequences which are defined in the ISO C standard shall be used.	Character sets	Required
6	Values of character types shall be restricted to a defined and documented subset of ISO 10646-1.	Character sets	Required
7	Trigraphs shall not be used.	Character sets	Required
8	Multibyte characters and wide string literals shall not be used.	Character sets	Required
9	Comments shall not be nested.	Comments	Required

Table 3: MISRA C 1998 rules summary

No	Rule	Type	Category
10	Sections of code should not be 'commented out'.	Comments	Advisory
11	Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore, the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	Identifiers	Required
12	No identifier in one namespace shall have the same spelling as an identifier in another namespace.	Identifiers	Advisory
13	The basic types of char, int, short, long, float, and double should not be used, but specific-length equivalents should be typedef'd for the specific compiler, and these type names used in the code.	Types	Advisory
14	The type char shall always be declared as unsigned char or signed char.	Types	Required
15	Floating point implementations should comply with a defined floating-point standard.	Types	Advisory
16	The underlying bit representation of floating-point numbers shall not be used in any way by the programmer..	Types	Required
17	typedef names shall not be reused.	Types	Required
18	Numeric constants should be suffixed to indicate type, where an appropriate suffix is available.	Constants	Advisory
19	Octal constants (other than zero) shall not be used.	Constants	Required
20	All object and function identifiers shall be declared before use.	Declarations and definitions	Required
21	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide the identifier.	Declarations and definitions	Required
22	Declaration of objects should be at function scope unless a wider scope is necessary.	Declarations and definitions	Advisory
23	All declarations at file scope should be static where possible.	Declarations and definitions	Advisory
24	Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	Declarations and definitions	Required
25	An identifier with external linkage shall have exactly one external definition.	Declarations and definitions	Required

Table 3: MISRA C 1998 rules summary (Continued)

No	Rule	Type	Category
26	If objects or functions are declared more than once, they shall have compatible declarations.	Declarations and definitions	Required
27	External objects should not be declared in more than one file.	Declarations and definitions	Advisory
28	The register storage class specifier should not be used.	Declarations and definitions	Advisory
29	The use of a tag shall agree with its declaration.	Declarations and definitions	Required
30	All automatic variables shall be assigned a value before being used.	Initialization	Required
31	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	Initialization	Required
32	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
33	The right-hand operand of an && or operator shall not contain side effects.	Operators	Required
34	The operands of a logical && or shall be primary expressions.	Operators	Required
35	Assignment operators shall not be used in expressions which return Boolean values.	Operators	Required
36	Logical operators should not be confused with bitwise operators.	Operators	Advisory
37	Bitwise operations shall not be performed on signed integer types.	Operators	Required
38	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).	Operators	Required
39	The unary minus operator shall not be applied to an unsigned expression.	Operators	Required
40	The sizeof operator should not be used on expressions that contain side effects.	Operators	Advisory
41	The implementation of integer division in the chosen compiler should be determined, documented, and taken into account.	Operators	Advisory

Table 3: MISRA C 1998 rules summary (Continued)

No	Rule	Type	Category
42	The comma operator shall not be used, except in the control expression of a for loop.	Operators	Required
43	Implicit conversions which may result in a loss of information shall not be used.	Conversions	Required
44	Redundant explicit casts should not be used.	Conversions	Advisory
45	Type casting from any type to or from pointers shall not be used.	Conversions	Required
46	The value of an expression shall be the same under any order of evaluation that the standard permits.	Expressions	Required
47	No dependence should be placed on C's operator precedence rules in expressions.	Expressions	Advisory
48	Mixed precision arithmetic should use explicit casting to generate the desired result.	Expressions	Advisory
49	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.	Expressions	Advisory
50	Floating-point variables shall not be tested for exact equality or inequality.	Expressions	Required
51	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	Expressions	Advisory
52	There shall be no unreachable code.	Control flow	Required
53	All non-null statements shall have a side-effect.	Control flow	Required
54	A null statement shall only occur on a line by itself, and shall not have any other text on the same line.	Control flow	Required
55	Labels should not be used, except in switch statements.	Control flow	Advisory
56	The goto statement shall not be used.	Control flow	Required
57	The continue statement shall not be used.	Control flow	Required
58	The break statement shall not be used (except to terminate the cases of a switch statement).	Control flow	Required
59	The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces.	Control flow	Required
60	All if, else if constructs should contain a final else clause.	Control flow	Advisory
61	Every non-empty case clause in a switch statement shall be terminated with a break statement.	Control flow	Required
62	All switch statements should contain a final default clause.	Control flow	Required

Table 3: MISRA C 1998 rules summary (Continued)

No	Rule	Type	Category
63	A switch expression should not represent a Boolean value.	Control flow	Advisory
64	Every switch statement shall have at least one case.	Control flow	Required
65	Floating-point variables shall not be used as loop counters.	Control flow	Required
66	Only expressions concerned with loop control should appear within a for statement.	Control flow	Advisory
67	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Control flow	Advisory
68	Functions shall always be declared at file scope.	Functions	Required
69	Functions with variable number of arguments shall not be used.	Functions	Required
70	Functions shall not call themselves, either directly or indirectly.	Functions	Required
71	Functions shall always have prototype declarations and the prototype shall be visible at both the function definition and call.	Functions	Required
72	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Functions	Required
73	Identifiers shall either be given for all the parameters in a function prototype declaration, or for none.	Functions	Required
74	If identifiers are given for any of the parameters, then the identifiers used in the declaration and definition shall be identical.	Functions	Required
75	Every function shall have an explicit return type.	Functions	Required
76	Functions with no parameters shall be declared with parameter type void.	Functions	Required
77	The unqualified type of parameters passed to a function shall be compatible with the unqualified expected types defined in the function prototype.	Functions	Required
78	The number of parameters passed to a function shall match the function prototype.	Functions	Required
79	The values returned by void functions shall not be used.	Functions	Required
80	void expressions shall not be passed as function parameters.	Functions	Required

Table 3: MISRA C 1998 rules summary (Continued)

No	Rule	Type	Category
81	<code>const</code> qualification should be used on function parameters which are passed by reference, where it is intended that the function will not modify the parameter.	Functions	Advisory
82	A function should have a single point of exit.	Functions	Advisory
83	For functions with non-void return types:...	Functions	Required
84	For functions with void return type, return statements shall not have an expression.	Functions	Required
85	Function calls with no parameters should have empty parentheses.	Functions	Advisory
86	If a function returns error information, then that error information should be tested.	Functions	Advisory
87	<code>#include</code> statements in a file shall only be preceded by other preprocessor directives or comments.	Preprocessing directives	Required
88	Non-standard characters shall not occur in header file names in <code>#include</code> directives.	Preprocessing directives	Required
89	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.	Preprocessing directives	Required
80	C macros shall only be used for symbolic constants, function-like macros, type qualifiers, and storage class specifiers.	Preprocessing directives	Required
91	Macros shall not be <code>#define</code> 'd and <code>#undef</code> 'd within a block.	Preprocessing directives	Required
92	<code>#undef</code> should not be used.	Preprocessing directives	Advisory
93	A function should be used in preference to a function-like macro.	Preprocessing directives	Advisory
94	A function-like macro shall not be 'called' without all of its arguments.	Preprocessing directives	Required
95	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Preprocessing directives	Required
96	In the definition of a function-like macro the whole definition, and each instance of a parameter, shall be enclosed in parentheses.	Preprocessing directives	Required
97	Identifiers in preprocessor directives should be defined before use.	Preprocessing directives	Advisory

Table 3: MISRA C 1998 rules summary (Continued)

No	Rule	Type	Category
98	There shall be at most one occurrence of the # or ## preprocessor operator in a single macro definition.	Preprocessing directives	Required
99	All uses of the #pragma directive shall be documented and explained.	Preprocessing directives	Required
100	The defined preprocessor operator shall only be used in one of the two standard forms.	Preprocessing directives	Required
101	Pointer arithmetic should not be used.	Pointers and arrays	Advisory
102	No more than 2 levels of pointer indirection should be used.	Pointers and arrays	Advisory
103	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to the same array, structure, or union.	Pointers and arrays	Required
104	Non-constant pointers to functions shall not be used.	Pointers and arrays	Required
105	All the functions pointed to by a single pointer to function shall be identical in the number and type of parameters and the return type.	Pointers and arrays	Required
106	The address of an object with automatic storage shall not be assigned to an object which may persist after the object has ceased to exist.	Pointers and arrays	Required
107	The null pointer shall not be de-referenced.	Pointers and arrays	Required
108	In the specification of a structure or union type, all members of the structure or union shall be fully specified.	Structures and unions	Required
109	Overlapping storage shall not be used.	Structures and unions	Required
110	Unions shall not be used to access subparts of larger data types.	Structures and unions	Required
111	Bitfields shall only be defined to be of type unsigned int or signed int.	Structures and unions	Required
112	Bitfields of type signed int shall be at least 2 bits long.	Structures and unions	Required
113	All the members of a structure (or union) shall be named and shall only be accessed via their name.	Structures and unions	Required

Table 3: MISRA C 1998 rules summary (Continued)

No	Rule	Type	Category
I14	Reserved words and standard library function names shall not be redefined or undefined.	Standard libraries	Required
I15	Standard library function names shall not be reused.	Standard libraries	Required
I16	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.	Standard libraries	Required
I17	The validity of values passed to library functions shall be checked.	Standard libraries	Required
I18	Dynamic heap memory allocation shall not be used.	Standard libraries	Required
I19	The error indicator <code>errno</code> shall not be used.	Standard libraries	Required
I20	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	Standard libraries	Required
I21	<code><locale.h></code> and the <code>setlocale</code> function shall not be used.	Standard libraries	Required
I22	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	Standard libraries	Required
I23	The signal handling facilities of <code><signal.h></code> shall not be used.	Standard libraries	Required
I24	The input/output library <code><stdio.h></code> shall not be used in production code.	Standard libraries	Required
I25	The library functions <code>atof</code> , <code>atoi</code> , and <code>atol</code> from library <code><stdlib.h></code> shall not be used.	Standard libraries	Required
I26	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> , and <code>system</code> from library <code><stdlib.h></code> shall not be used.	Standard libraries	Required
I27	The time handling functions of library <code><time.h></code> shall not be used.	Standard libraries	Required

Table 3: MISRA C 1998 rules summary (Continued)

Environment rules

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

Rule 1 (required) All code shall conform to ISO 9899 standard C, with no extensions permitted.

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 GUI options) to:

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

Note: The compiler does not generate this error if you use IAR extensions from within the code by using a pragma directive.

Examples of rule violations

```
int __far my_far_variable;  
int port @ 0xbeef;
```

Example of correct code

```
#pragma location=0xbeef  
int port;
```

Rule 2 (advisory) Code written in languages other than C should only be used if there is a defined interface standard for object code to which the compiler/assemblers for both languages 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 3 (advisory) Assembler language functions that are called from C should be written as C functions containing only inline assembler language, and inline assembler language should not be embedded in normal code.

How the rule is checked

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

Rule 4 (advisory) Provisions should be made for appropriate runtime checking.

How the rule is checked

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

Character sets

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

Rule 5 (required) Only those characters and escape sequences which 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 `\octal`.

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 6 (required) Values of character types shall be restricted to a defined and documented subset of ISO 10646-1.

How the rule is checked

This restriction is implemented according to the information in the section about characters in the chapter *Implementation-defined behavior* in the *IAR C/EC++ Compiler Reference Guide*.

Rule 7 (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???";
```

Rule 8 (required) Multibyte characters and wide string literals shall not be used.

How the rule is checked

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

- any multibyte character occurs in a character literal, a string literal, a comment, or a header file name
- any of the functions `mblen`, `mbtowc`, `wctomb`, `mbstowcs`, or `wcstombs` (declared in the header file `stdlib.h`) are called
- a wide string literal is used.

Note: The compiler will only generate an error for using `mblen`, `mbtowc`, `wctomb`, `mbstowcs`, or `wcstombs` when the correct header file is included. Using any other function with the same name will not generate an error.

Comments

The rules in this section are concerned with the use of comments in the code.

Rule 9 (required) Comments shall not be nested.

How the rule is checked

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

Rule 10 (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.

Identifiers

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

Rule 11 (required) Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore, 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

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

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.

Rule 12 (required) No identifier in one namespace shall have the same spelling as an identifier in another namespace.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a declaration or definition would hide an identifier if they were in the same namespace. For example, fields of different structures will not generate an error.

Example of rule violations

```
struct an_ident { int an_ident; } an_ident;
```

Example of correct code

```
struct a_struct { int a_field; } a_variable;
```

Types

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

Rule 13 (advisory) The basic types of `char`, `int`, `short`, `long`, `float`, and `double` should not be used, but specific-length equivalents should be `typedef`'d for the specific compiler, and these type names used in the code.

How the rule is checked

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

Example of rule violations

```
int x;
```

Example of correct code

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

Rule 14 (required) The type `char` shall always be declared as `unsigned char` or `signed char`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the basic type `char` is used without explicitly having either a `signed` or `unsigned` specifier.

Rule 15 (advisory) Floating point implementations should comply with a defined floating-point standard.

How the rule is checked

The floating-point standard of the IAR C/C++ Compiler is documented in the *IAR C/EC++ Compiler Reference Guide*.

Rule 16 (required) The underlying bit representation of floating-point numbers shall not be used in any way by the programmer.

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 (required) `typedef` names shall not be reused.

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` using a name previously used in a declaration or definition.

Example of correct code

```
/* No error for this widely used coding idiom */
typedef struct a_struct {
    ...
} a_struct;
```

Constants

The rules in this section are concerned with the use of constants.

Rule 18 (advisory) Numeric constants should be suffixed to indicate type, where an appropriate suffix is available.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any integer constant whose type is not the same in any standard-conforming implementation.

Example of rule violations

```
100000
```

Examples of correct code

```
30000
100000L
100000UL
```

Rule 19 (required) Octal constants (other than zero) 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 20 (required) All object and function identifiers shall be declared before use.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any implicit declaration of a function.

Note: This rule still permits Kernighan & Ritchie functions since their behavior is well-defined.

Rule 21 (required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide the 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 22 (advisory) Declaration of objects should be at function scope unless a wider scope is necessary.

How the rule is checked

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

Rule 23 (advisory) All declarations at file scope should be static where possible.

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 24 (required) Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.

How the rule is checked

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

- with *external* linkage and there already exists an identical symbol in the current scope with *internal* linkage, or
- with *internal* linkage and there already exists an identical symbol in the current scope with *external* linkage.

Rule 25 (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 26 (required) If objects or functions are declared more than once, they shall have compatible declarations.

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 27 (advisory) External objects should not be declared in more than one file.

How the rule is checked

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

Rule 28 (advisory) The `register` storage class specifier should not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the `register` keyword is used.

Rule 29 (required) The use of a tag shall agree with its declaration.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an enumeration constant is assigned to variable of the wrong enumeration type.

The linker will generate an error, indicating a violation of this rule, if the same structure or enumeration tag is used in several different translation units.

Initialization

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

Rule 30 (required) All automatic variables shall be assigned a value before being used.

How the rule is checked

Partial support for checking this rule is available in the implementation.

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 31 (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 { int a,b; } a_struct = { 1 };  
struct { int a[3]; } a_struct = { 1, 2 };
```

Examples of correct code

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

Rule 32 (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.

Operators

The rules in this section are concerned with the behavior of operators and operands.

Rule 33 (required) The right-hand operand of an && 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 34 (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: There is an exception: 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 35 (required) Assignment operators shall not be used in expressions which return Boolean values.

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 rule violations

```
if (a = func()) {
    ...
}
```

Example of correct code

```
if ((a = func()) != 0) {
    ...
}
```

Rule 36 (advisory) Logical operators should not be confused with bitwise operators.

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.

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 ) { ... }
```

Note: The following are considered Boolean contexts:

- 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.

Rule 37 (required) Bitwise operations shall not be performed on signed integer types.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the type of the operation is a signed integer, with an exception if the expression is:

- a positive constant
- directly converted from an integer type strictly smaller than `int`
- a Boolean operation.

Rule 38 (required) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).

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 since the value of `i8` will be 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 39 (required) The unary minus operator shall not be applied to an unsigned expression.

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 40 (advisory) The `sizeof` operator should 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 41 (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 the IAR C/C++ Compiler, the sign of the remainder on integer division is the same as the sign of the dividend, as documented in the *IAR C/EC++ Compiler Reference Guide*.

Rule 42 (required) The comma operator shall not be used, except in the control expression of a `for` loop.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a comma is used anywhere except in the first or last part in the head of a `for` loop.

Conversions

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

Rule 43 (required) Implicit conversions which may result in a loss of information shall not be used.

How the rule is checked

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

Rule 44 (advisory) Redundant explicit casts should not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an explicit cast is used to convert to an identical type.

Rule 45 (required) Type casting from any type to or from pointers shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a value of object pointer type is cast to any other type or if any value is cast to an object pointer type.

Note: This includes implicit and explicit casts to or from `void` pointer types, which are otherwise allowed by the standard.

Expressions

The rules in this section are concerned with expressions.

Rule 46 (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 the a volatile location.

Note: The implementation does not generate an error for the expression $f() + f()$.

Rule 47 (advisory) No dependence should be placed on C's operator precedence rules in expressions.

How the rule is checked

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

Example of rule violations

```
x = 3 * a + b / c;
```

Example of correct code

```
x = (3 * a) + (b / c);
```

Rule 48 (advisory) Mixed precision arithmetic should use explicit casting to generate the desired result.

How the rule is checked

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

Rule 49 (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.

Rule 50 (required) Floating-point variables shall not be tested for exact 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. If a comparison is explicitly against the floating-point constant 0.0, no error message is given.

Rule 51 (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.

Control flow

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

Rule 52 (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 53 (required) All non-null statements shall have a side-effect.

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 rule violations

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

Examples of correct code

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

Rule 54 (required) A null statement shall only occur on a line by itself, and shall not have any other text on the same line.

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 white space.

Rule 55 (advisory) Labels should not be used, except in `switch` statements.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a label that is not a `case` label or `default` is used.

Rule 56 (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 57 (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 58 (required) The `break` statement shall not be used (except to terminate the cases of a `switch` statement).

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any `break` statement that is not part of a `switch` statement.

Rule 59 (required) The statements forming the body of an `if`, `else if`, `else`, `while`, `do ... while` or `for` statement shall always be enclosed in braces.

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 above is not a block.

Rule 60 (advisory) All `if`, `else if` constructs should contain a final `else` clause.

How the rule is checked

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

Rule 61 (required) Every non-empty `case` clause in a `switch` statement shall be terminated with a `break` statement.

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 62 (required) All `switch` statements should contain a final 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 63 (advisory) A `switch` expression should not represent a Boolean value.

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 64 (required) Every `switch` statement shall have at least one `case`.

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.

Rule 65 (required) Floating-point variables shall not be used as loop counters.

How the rule is checked

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

Rule 66 (advisory) Only expressions concerned with loop control should appear within a `for` statement.

How the rule is checked

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

Rule 67 (advisory) Numeric variables being used within a `for` loop for iteration counting should 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.

Functions

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

Rule 68 (required) Functions shall always 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 69 (required) Functions with variable number of arguments shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, whenever a function is declared or defined 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 70 (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.

Rule 71 (required) Functions shall always 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 rule violations

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

Example of correct code

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

Rule 72 (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 73 (required) Identifiers shall either be given for all the parameters in a function prototype declaration, or for none.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a prototype declaration has an identifier for more than one parameter, but for fewer than the number of parameters in the prototype.

Rule 74 (required) If identifiers are given for any of the parameters, then the identifiers used in the declaration and definition shall be identical.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the identifier given in the definition of a function does not match the corresponding identifier given in the prototype.

Rule 75 (required) Every function shall have an explicit return type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a function has an implicitly declared return type.

Rule 76 (required) Functions with no parameters shall be declared with parameter type `void`.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a function declaration or definition is not also a prototype.

Rule 77 (required) The unqualified type of parameters passed to a function shall be compatible with the unqualified expected types defined in the function prototype.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any function call requiring an implicit conversion of any of the parameters.

Rule 78 (required) The number of parameters passed to a function shall match the function prototype.

How the rule is checked

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

Rule 79 (required) The values returned by `void` functions shall not be used.

How the rule is checked

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

Rule 80 (required) `void` expressions shall not be passed as function parameters.

How the rule is checked

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

Rule 81 (advisory) `const` qualification should be used on function parameters which are passed by reference, where it is intended that the function will not modify the parameter.

How the rule is checked

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

Rule 82 (advisory) A function should have a single point of exit.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for the second point of exit from a function, which is either a `return` statement or the end of the function.

No error is given for points of exit that cannot be reached.

Rule 83 (required) For functions with non-void return types:

- there shall be one `return` statement for every `exit` branch (including the end of the program),
- each `return` shall have an expression,
- the `return` expression shall match the declared return type.

How the rule is checked

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

- a function with non-void return type does not have a `return` last in the function
- a `return` statement does not have an expression
- the expression given in any `return` statement is implicitly converted to match the return type.

Rule 84 (required) For functions with `void` return type, `return` statements shall not have an expression.

How the rule is checked

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

Rule 85 (advisory) Function calls with no parameters should have empty parentheses.

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 rule violations

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

Example of correct code

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

Rule 86 (advisory) If a function returns error information, then that error information should 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.

Preprocessing directives

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

Rule 87 (required) `#include` statements in a file shall 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 88 (required) Non-standard characters shall 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 89 (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 90 (required) C macros shall only be used for symbolic constants, function-like macros, type qualifiers, and storage class specifiers.

How the rule is checked

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

Rule 91 (required) Macros shall not be `#define`'d and `#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 of a file-level scope.

Rule 92 (advisory) `#undef` should 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 93 (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 94 (required) A function-like macro shall not be ‘called’ without all of its arguments.

How the rule is checked

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 rule violations

```
MACRO ( , )
```

Example of correct code

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

Rule 95 (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 96 (require) In the definition of a function-like macro the whole definition, and each instance of a parameter, shall be enclosed in parentheses.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a function-like macro is defined where either:

- a macro parameter in the replacement text of the macro is not enclosed in parentheses, or

- the replacement text is not enclosed in parentheses.

Examples of rule violations

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

Example of correct code

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

Rule 97 (advisory) Identifiers in preprocessor directives should be defined before use.

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 98 (required) There shall be at most one occurrence of the `#` or `##` preprocessor operator 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 rule violations

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

Examples of correct code

```
#define FOO(x) #x
#define FOO(x) my_ ## x
```

Rule 99 (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 100 (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.

Pointers and arrays

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

Rule 101 (advisory) Pointer arithmetic should not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if the left- or right-hand side of `+`, `-`, `+=`, or `-=` is an expression of pointer type.

Rule 102 (advisory) No more than 2 levels of pointer indirection should be used.

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 103 (required) Relational operators shall not be applied to pointer types except where both operands are of the same type and point to the same array, structure, or union.

How the rule is checked

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

Rule 104 (required) Non-constant pointers to functions shall not be used.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an explicit cast of a value to a function pointer type is made, except when casting:

- constant values
- function pointers.

Rule 105 (required) All the functions pointed to by a single pointer to function shall be identical in the number and type of parameters and the return type.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if an implicit or explicit cast of a function pointer is made to a different function pointer type.

Rule 106 (required) The address of an object with automatic storage shall not be assigned to an object which may persist after the object has ceased to exist.

How the rule is checked

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

Rule 107 (required) The null pointer shall not be de-referenced.

How the rule is checked

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

Structures and unions

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

Rule 108 (required) In the specification of a structure or union type, all members of the structure or union shall be fully specified.

How the rule is checked

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

Rule 109 (required) Overlapping storage 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.

Rule 110 (required) Unions shall not be used to access subparts of larger data types.

How the rule is checked

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

Rule 111 (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 112 (required) Bitfields of type `signed int` 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.

Rule 113 (required) All the members of a structure (or union) shall be named and shall only be accessed via their name.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if a bitfield is declared without a name or if the address of a structure field is taken.

Standard libraries

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

Rule 114 (required) Reserved words and standard library function names shall not be 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 115 (required) Standard library function names shall not be reused.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, for any function definition used to define a 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 116 (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

This rule is not enforced.

Rule 117 (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.

Rule 118 (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 119 (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 120 (required) The macro `offsetof`, in library `<stddef.h>`, 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 121 (required) `<locale.h>` and the `setlocale` function 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 `locale.h` is included.

Rule 122 (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 123 (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 124 (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 125 (required) The library functions `atof`, `atoi`, and `atol` from 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 126 (required) The library functions `abort`, `exit`, `getenv`, and `system` from 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 127 (required) The time handling functions of library `<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.