REJ10J1314-0100

Everywhere you imagine. RENESAS

MISRA C Rule Checker
SQMlint V.1.03
User's Manual

Add-in to Renesas' compilers

User's Manual

Rev.1.00
Aug. 1, 2006

Renesas Technology
www.renesas.com

Keep safety first in your circuit designs!

- Renesas Technology Corporation and Renesas Solutions Corporation put the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation, Renesas Solutions Corporation or a third party.
- Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation and Renesas Solutions Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor for the latest product information before purchasing a product listed herein. The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors. Please also pay attention to information published by Renesas Technology Corporation and Renesas Solutions Corporation by various means, including the Renesas home page (http://www.renesas.com).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation and Renesas Solutions Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation or Renesas Solutions Corporation for further details on these materials or the products contained therein.

For inquiries about the contents of this document or product, fill in the text file the installer generates in the following directory and email to your local distributor.

\SUPPORT\Product-name\SUPPORT.TXT

Renesas Tools Homepage http://www.renesas.com/en/tools

1. Introduction

SQMlint statically checks C source codes to find codes that deviate from any of MISRA C[1] rules.

The MISRA C rules are guidelines and can be used as inspection items of source code review. Some deviations from the rules can be detected by inspection tools. SQMlint checks C source codes against each of the MISRA C rules, and reports where codes deviate from any of the rules, thereby assisting you in source code review.

The titles of MISRA C rules described in this manual, as well as the MISRA C rule inspection items for the MISRA C Rule Checker are based on the "Guidelines for the Use of the C Language in Vehicle Based Software" issued by MISRA.

[1] MISRA C refers to the guidelines for the use of the C language in developing vehicle-based software that are issued by the Motor Industry Software Reliability Association (MISRA) of the U.K.

2. Overview

SQMlint checks to see if C source codes deviate from any of the MISRA C rules.

If a deviation is detected in your C source codes, SQMlint outputs a report message. (See Chapter 4, “Report Specifications.”)

Example:

```
typedef unsigned short UINT16;
extern volatile UINT16 port1 = 0;
extern volatile UINT16 port2 = 0;

void func(void);
void func(void)
{
   while(port1 != 0) {
       if (port2 == 0) {
           break;
       }
   }
}
```

When the above program is inspected by SQMlint, a report message similar to the following will be output. It indicates that the use of the break statement on the 10th line of the program deviates from rule 58.

[MISRA(58) Complaining : test.c, 10] 'break' statement shall not be used (except in a 'switch')

Note

Not all rules defined in MISRA C can be inspected by SQMlint. For details about the MISRA C rules that can be inspected by SQMlint, please see Chapter 6, “List of MISRA C Rule Inspection Items.”

2.1. Position

SQMlint is started from a compile driver as part of compile operation. Therefore, the source code is processed by the compiler even after being checked against the MISRA C rules.

Supplement

The code generated by the compiler is unaffected by MISRA C rule checking.

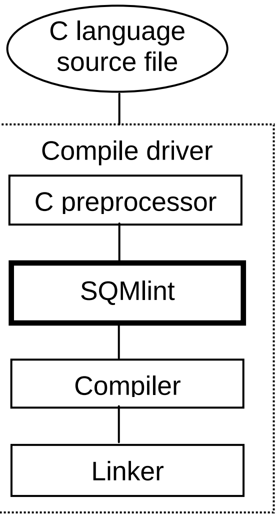


Figure 2.1 Position of SQMlint

2.2. Input/Output Files

SQMlint checks the C language source file against the MISRA C rules and outputs the inspection result to a report file, or to standard error as a report message (see Figure 2.2). For details about report file and report message specifications, see Chapter 4, "Report Specifications."

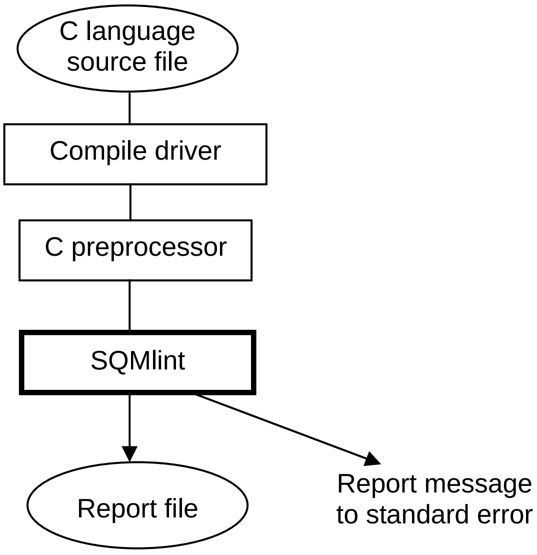


Figure 2.2 Input/Output Files of SQMlint

3. How to Use SQMlint

3.1. When you’re using C Compiler Package for M16C family

To inspect your C source codes against the MISRA C rules, specify the options in the “C compiler for M16C Family” that are shown in Table.

Example:

```
nc30 test.c -c –misra_all –misra_report report.csv
nc308 test.c -c –misra_all –misra_report report.csv
```

SQMlint inspects the *test.c* for all of the supported MISRA C rules and outputs the result to *report.csv*. The *test.c* also is processed by the compiler.

Table 3.1.1 Compiler Options for MISRA C Inspection

Option	Description
`-misra_all`	The source code is checked against all of the supported MISRA C rules. In this case, no rule numbers can be specified.
`-misra_apply` *rule-number* `[,`*rule-number*`,...]`	The source code is checked against only of the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0, equal to or greater than 128, or not included among the rule numbers to be inspected, an option error is assumed. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_ignore` *rule-number* `[,`*rule-number*`,...]`	The source code is not checked against the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. The source code is inspected for all of the MISRA C rules, except the rule numbers specified here. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*

`-misra_required`	The source code is checked against all of the supported MISRA C rules that are classified as “required.” In this case, no rule numbers can be specified.
`-misra_required_add` *rule-number* `[,`*rule-number*`,...]`	The source code is checked against all of the supported MISRA C rules that are classified as “required,” as well as against those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_required_remove` *rule-number* `[,`*rule-number*`,...]`	The source code is checked against all of the supported MISRA C rules that are classified as “required,” not including those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_report` *output-report-file-name*	Specify the output report file name to which the result of MISRA C rule checking is to be saved. If this option is specified, the output report file name cannot be omitted. For the file extension, specify “.csv.” If this option is omitted, inspection results are output to standard error.
`-ignore_files_misra` <*file-name*>`[,`<*file-name*>`]`	The specified file is not inspected. - In the form of both a "file name" and "a file name with a directory" can describe a file name. (file, dir/file, drive:/dir/file) - In directory separator '/' and '¥' are available.
`-check_language_extension`	It reports on the enhancing key word (near/far etc.) and extended specifications (one byte enum) as violation of the rule-1.
`-check_no_prototype_extension`	It reports as a violation to rule-71 when there is no prototype declaration of the function modified by __entry or __interrupt.

Note

The options `-misra_all`, `-misra_apply`, `-misra_ignore`, `-misra_required`, `-misra_required_add` and `-misra_required_remove` cannot be specified at the same time.

3.2. When you're using M32R family

To inspect your C source codes against the MISRA C rules, specify the options in the "C compile for M32R Family" driver that are shown in Table.

Example:

```
cc32r test.c -c -misra_all -misra_report report.csv
```

SQMlint inspect the *test.c* for all of the supported MISRA C rules and outputs the result to *report.csv*. The *test.c* also is processed by the compiler.

Table 3.2.1 M3T-CC32R Compiler Options for MISRA C Inspection

Option	Description
`-misra_all`	The source code is checked against all of the supported MISRA C rules. In this case, no rule numbers can be specified.
`-misra_apply` *rule-number* `[,`*rule-number*`,...]`	The source code is checked against only of the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0, equal to or greater than 128, or not included among the rule numbers to be inspected, an option error is assumed. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_ignore` *rule-number* `[,`*rule-number*`,...]`	The source code is not checked against the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. The source code is inspected for all of the MISRA C rules, except the rule numbers specified here. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_required`	The source code is checked against all of the supported MISRA C rules that are classified as "required." In this case, no rule numbers can be specified.

`-misra_required_add` *rule-number* `[,`*rule-number*`,...]`	The source code is checked against all of the supported MISRA C rules that are classified as "required," as well as against those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_required_remove` *rule-number* `[,`*rule-number*`,...]`	The source code is checked against all of the supported MISRA C rules that are classified as "required," not including those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`-misra_report` *output-report-file-name*	Specify the output report file name to which the result of MISRA C rule checking is to be saved. If this option is specified, the output report file name cannot be omitted. For the file extension, specify ".csv." If this option is omitted, inspection results are output to standard error.

Note

The options `-misra_all`, `-misra_apply`, `-misra_ignore`, `-misra_required`, `-misra_required_add` and `-misra_required_remove` cannot be specified at the same time.

3.3. When you're using SuperH RISC engine family

3.3.1. Options

To inspect your C source codes against the MISRA C rules, specify the options in the "C/C++ compiler for SuperH RISC engine" that are shown in Table. Please use only the capital letter in the option character string when you use the compilation option by the abbreviated form.

Example:

```
shc test.c –misra=all –report_misra=report.csv
```

SQMlint inspect the *test.c* for all of the supported MISRA C rules and outputs the result to *report.csv*. The *test.c* also is processed by the compiler.

Table 3.3.1.1 Options for MISRA C Inspection

Option	Description

`MIsra={`
`ALL`
`APply=`*rule-number* `[,`*rule-number*`,...]` |
`IGnore=`*rule-number* `[,`*rule-number*`,...]` |
`REQUIRED` |
`REQUIRED_Add=`*rule-number* `[,`*rule-number*`,...]` |
`REQUIRED_Remove=`*rule-number* `[,`*rule-number*`,...]` |
`}`
	(1) misra=all
	The source code is checked against all of the supported MISRA C rules.
	In this case, no rule numbers can be specified.
	(2) misra=apply
	The source code is checked against only of the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0, equal to or greater than 128, or not included among the rule numbers to be inspected, an option error is assumed.
	(3) misra=ignore
	The source code is not checked against the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. The source code is inspected for all of the MISRA C rules, except the rule numbers specified here. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed.
	(4) misra=required
	The source code is checked against all of the supported MISRA C rules that are classified as "required."
	In this case, no rule numbers can be specified.

	(5) misra= required_add The source code is checked against all of the supported MISRA C rules that are classified as “required,” as well as against those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed. (6) misra= required_remove The source code is checked against all of the supported MISRA C rules that are classified as “required,” not including those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`REPort_misra[=`<*output-report-file-name*>`]`	Specify the output report file name to which the result of MISRA C rule checking is to be saved. When you omit the output-report-file-name, the file of the name of "C-source-file-name + **.csv**" is generated. If this option is omitted, inspection results are output to standard error.

Note

The options `-misra=all`, `-misra=apply`, `-misra=ignore`, `-misra=required`, `-misra=required_add` and `-misra=required_remove` cannot be specified at the same time.

3.3.2. Error messages

When SQMlint is installed, the following error messages are added to the compiler.

Error levels are classified into the following five types:

- (I): Information error (Continues compiling processing and outputs the object program.)
- (W): Warning error (Continues compiling processing and outputs the object program.)
- (E): Error (Continues compiling processing but does not output the object program.)

- (F): Fatal error (Aborts compiling processing.)
- (–): Internal error (Aborts compiling processing.)

C3330 (F) MISRA Error

An invalid option is specified.

C3331 (F) MISRA Internal Error

It does not normally occur. (This is an internal error.) Make a note of the internal error number, file name, line number, and comment in the message, and contact the support department of the vendor.

C1340 (W) 'MISRA' option ignored

The options(-prep and -misra) that cannot be specified simultaneously are specified. "-misra" option is ignored.

C1340 (W) 'MISRA' option ignored

The options(-lang=C++ and -misra) that cannot be specified simultaneously are specified. "-misra" option is ignored.

C3327 (F) Cannot find "sqmlint.exe"

The execution file(sqmlint.exe) is not found.

3.4. When you’re using H8S, H8/300 series

3.4.1. Options

To inspect your C source codes against the MISRA C rules, specify the options in the “C/C++ compiler for H8S, H8/300 series” that are shown in Table. Please use only the capital letter in the option character string when you use the compilation option by the abbreviated form.

Example:

```
ch38 test.c –misra=all –report_misra=report.csv
```

SQMlint inspect the *test.c* for all of the supported MISRA C rules and outputs the result to *report.csv*. The *test.c* also is processed by the compiler.

Table 3.4.1.1 Options for MISRA C Inspection

Option	Description

`MIsra={`
`ALL`
`APply=`*rule-number* `[,`*rule-number*`,...]` |
`IGnore=`*rule-number* `[,`*rule-number*`,...]` |
`REQUIRED` |
`REQUIRED_Add=`*rule-number* `[,`*rule-number*`,...]` |
`REQUIRED_Remove=`*rule-number* `[,`*rule-number*`,...]` |
`}`
	(1) misra=all
	The source code is checked against all of the supported MISRA C rules.
	In this case, no rule numbers can be specified.
	(2) misra=apply
	The source code is checked against only of the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0, equal to or greater than 128, or not included among the rule numbers to be inspected, an option error is assumed.
	(3) misra=ignore
	The source code is not checked against the MISRA C rules of the specified rule numbers. One or more rule numbers always in decimal must be specified. The source code is inspected for all of the MISRA C rules, except the rule numbers specified here. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed.

	(4) misra=required The source code is checked against all of the supported MISRA C rules that are classified as “required.” In this case, no rule numbers can be specified. (5) misra= required_add The source code is checked against all of the supported MISRA C rules that are classified as “required,” as well as against those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. If any specified rule number is equal to or less than 0 or equal to or greater than 128, an option error is assumed. (6) misra= required_remove The source code is checked against all of the supported MISRA C rules that are classified as “required,” not including those that are specified by rule numbers. One or more rule numbers always in decimal must be specified. *Do not insert spaces between a rule number and the comma (,) or between the comma (,) and a rule number.*
`REPort_misra[=<`*output-report-file-name*`>]`	Specify the output report file name to which the result of MISRA C rule checking is to be saved. When you omit the output-report-file-name, the file of the name of "C-source-file-name + **.csv**" is generated. If this option is omitted, inspection results are output to standard error.
`IGnore_files_misra=<` *file-name*`>[,<` *file-name*`>]`	The specified file is not inspected. - In the form of both a "file name" and "a file name with a directory" can describe a file name. (file, dir/file, drive:/dir/file) - In directory separator '/' and '¥' are available.
`CHECK_Language_extension`	It reports on the enhancing key word (__abs8 etc.) and extended specifications (one byte enum) as violation of the rule-1.

`CHECK_No_prototype_extension`	It reports as a violation to rule-71 when there is no prototype declaration of the function modified by __entry or __interrupt.

Note

The options `-misra=all`, `-misra=apply`, `-misra=ignore`, `-misra=required`, `-misra=required_add` and `-misra=required_remove` cannot be specified at the same time.

3.4.2. Error messages

When SQMlint is installed, the following error messages are added to the compiler.

Error levels are classified into the following five types:

- (I): Information error (Continues compiling processing and outputs the object program.)
- (W): Warning error (Continues compiling processing and outputs the object program.)
- (E): Error (Continues compiling processing but does not output the object program.)
- (F): Fatal error (Aborts compiling processing.)
- (–): Internal error (Aborts compiling processing.)

C3330 (F) MISRA Error

An invalid option is specified.

C3331 (F) MISRA Internal Error

It does not normally occur. (This is an internal error.) Make a note of the internal error number, file name, line number, and comment in the message, and contact the support department of the vendor.

C1340 (W) 'MISRA' option ignored

The options(-prep and -misra) that cannot be specified simultaneously are specified. "-misra" option is ignored.

C1340 (W) 'MISRA' option ignored

The options(-lang=C++ and -misra) that cannot be specified simultaneously are specified. "-misra" option is ignored.

C3327 (F) Cannot find "sqmlint.exe"

The execution file(sqmlint.exe) is not found.

4. Report Specifications

The result of MISRA C rule checking consists of the following two levels:

1. Complaining
   When any part of the source code deviates from MISRA C rules
2. Warning
   When any part of the source code is likely to deviate from MISRA C rules

4.1. Report Message

Messages that are output to standard error as the result of MISRA C rule checking are referred to as the report message.

If a complaining or warning against any MISRA C rules is detected, SQMlint outputs a message in the format shown below.

```
[MISRA (rule-number) Complaining: file-name, line-number] message
[MISRA (rule-number) Warning: file-name, line-number] message
```

for complainings
for warnings

4.2. Report File

When the result of MISRA C rule checking is output to a report file, the report is output in the CSV (Comma Separated Value) format that can be available in most spreadsheet applications.

If a complaining or warning against any MISRA C rules is detected in the source code, SQMlint outputs particulars in a report file that are comprised of MISRA C rule numbers, classification of complaining or warning, file name, line numbers and report messages in that order, with each entry separated by a comma (,). Each report message is followed by a new-line code.

Example of an output report file:

```
Rule,Level,File,Line,Message<new line>
57,Complaining,"test.c",6 ,"the 'continue' ..."<new line>

58,Warning,"test.c",10 ,"the 'break' ..."<new line>
```

Header
for a complaining
(6th line of test.c)
for a warning
(10th line of test.c)

Each header for each column is output to the first line.
The inspection results are output to the second and subsequent lines.

4.3. Compile Errors

The compile errors detected by SQMlint are reported as deviations from MISRA C rule 1.

If 500 or more compile errors are detected, the compiler aborts the inspection and stops running SQMlint.

Such a limitation does not apply to the report messages for MISRA C rules that are irrelevant to compile errors, so that any number of report messages will be output.

Note

The compile errors detected by the compiler that is run after MISRA C rule checking by SQMlint are not output as report messages. (They are output to standard error as ordinary compile errors.) Nor are they output to report files.

5. Confirming the Result

There are following three methods for confirming the result of MISRA C rule checking. Choose the appropriate method of confirmation depending on the purposes for which you're going to verify the inspection result.

5.1. Referring to Report Files to Confirm

If you wish to make use of inspection results in source code review, save the results in report files (CSV format) and refer to the saved files for confirmation.

[Making the most of a report file]

1. Read the report files into your computer using a spreadsheet application, and the file can be manipulated by, for example, sorting inspection results by rule.
2. Use the SQMmerger utility attached to SQMlint to create a mixed text file that is produced by merging the inspection result into its C source file. For details on how to make use of SQMmerger, see Section 5.3 in the later part of this manual.

5.2. Referring to Report Messages to Confirm

This method may be used to temporarily confirm the inspection result while you compile the program. You can check the result as if you checked the ordinary compile errors.

5.3. Using the SQMmerger to Confirm

Use of SQMmerger can merge a report file into a corresponding C source file and produce a mixed text file. In the mixed text file, a report message for a complaining or warning is inserted after a corresponding line that has the complaining or warning. You can make use of the mixed text file in source code review, etc.

For details on how to use SQMmerger, see the "SQMmerger User's Manual" included with your SQMlint.

Example of a mixed text file:

```
1 : void  func(void);
2 : void  func(void)
3 : {
4 : LABEL:
[MISRA(55) Complaining] label ('LABEL') should not be used

5 :
6 :     goto LABEL;
[MISRA(56) Complaining] the 'goto' statement shall not be used

7 :}
```

6. List of Supported MISRA C Rules

Table 6.1 lists each specific MISRA C rule for which C source code can be or cannot be inspected by SQMlint.

Table 6.1 List of Supported MISRA C Rules (v: Can be inspected; *: Can be inspected with some restrictions; (empty): Excluded from those inspected)

Rule No.	Support	Rule No.	Support	Rule No.	Support	Rule No.	Support
1	v	33	v	65	v	97	
2		34	v	66		98	
3		35	v	67		99	v
4		36	v	68	v	100	
5	v	37	v	69	v	101	v
6		38	v	70	*	102	v
7		39	v	71	v	103	v
8	v	40	v	72	*	104	v
9		41		73	v	105	v
10		42	v	74	v	106	*
11		43	v	75	v	107	
12	v	44	v	76	v	108	v
13	v	45	v	77	v	109	
14	v	46	*	78	v	110	v
15		47		79	v	111	v
16		48	v	80	v	112	v
17	*	49	v	81		113	v
18	v	50	v	82	v	114	
19	v	51	*	83	v	115	v
20	v	52		84	v	116	
21	*	53	v	85	v	117	
22	*	54	*	86		118	v
23		55	v	87		119	v
24	v	56	v	88		120	
25		57	v	89		121	v
26		58	v	90		122	v
27		59	v	91		123	v
28	v	60	v	92		124	v
29	v	61	v	93		125	*
30		62	v	94		126	v
31	v	63	v	95		127	v
32	v	64	v	96			

7. Handling of Each MISRA C Rule

This chapter describes, in order of rule numbers, how each of the MISRA C rules will be handled by SQMlint. The following shows how to read the explanation of each rule.

Section-number **Rule *number***
(Category of rule): requirement text on MISRA C

- **Interpretation**
  Describes how the rule is interpreted at Renesas.
- **Functional specification**
  Describes the content of inspection in detail.
- **Precaution**
- **Limitation**

*Requirement texts are based on the "Guidelines for the Use of the C Language in Vehicle Based Software" issued by MISRA.

- The following describes the meaning of the terms used in the explanation of each rule.

Term	Meaning of the term
Identifier	Refers to any of the label name, tag name, typedef name, variable name, function name, member name or enumerator name.
Translation unit	Refers to the include files written in one source file and the source file.
Object	Refers to the data storage that has the value and size corresponding to a specified type. For example, variables are an object.

- About the *typedef* names used in the explanation of each rule
  The following *typedef* names are used in the explanation of each rule to represent the types of variables, etc.

typedef **name**	**Meaning of *typedef* name**
`SCHAR`	1-byte signed character type (*signed char* type)
`UCHAR`	1-byte unsigned character type (*unsigned char* type)
`SINT16`	2-byte signed integer type (*signed short* type)
`UINT16`	2-byte unsigned integer type (*unsigned short* type)
`SINT32`	4-byte signed integer type (*signed int* type)
`UINT32`	4-byte unsigned integer type (*unsigned int* type)
`SLONG`	4-byte signed integer type (*signed long* type)
`ULONG`	4-byte unsigned integer type (*unsigned long* type)
`FLOAT`	4-byte floating-point type (*float* type)
`DOUBLE`	8-byte floating-point type (*double* type)

7.1. Rule 1

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

- **Interpretation**

  No language extensions that are not compliant with ISO9899:1990 can be used. All codes must conform to ISO9899:1990.

- **Functional specification**

  If any language extensions that are not compliant with ISO9899:1990 are used, SQMlint reports them as a complaining.

  If any code does not conform to ISO9899:1990, SQMlint reports it as a complaining.

[note]

Even if the ***-misra ignore 1*** option is specified, the compile errors detected by SQMlint are reported as deviations from MISRA C rule 1.

7.2. Rule 2 (Not supported)

(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 compilers/assemblers for both languages conform.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

  For our supported C compilers, there is a defined standard for interfacing with our supported assemblers.

7.3. Rule 3 (Not supported)

(required): Assembly language functions that are called from C should be written as C functions containing only in-line assembly language, and in-line assembly language should not be embedded in normal C code.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.4. Rule 4 (Not supported)

(advisory): Provision should be made for appropriate run-time checking.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.5. Rule 5

(required): Only those characters and escape sequences which are defined in the ISO C standard shall be used.

- **Interpretation**

  Only those characters and escape sequences that are compliant with ISO9899:1990 can be used.

- **Functional specification**

  If kanji code, katakana or undefined escape sequence is used in any character constant or string literal, SQMlint reports it as a complaining.

7.6. Rule 6 (Not supported)

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

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.7. Rule 7 (Not supported)

(required): Trigraphs shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.8. Rule 8

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

- **Interpretation**
  Multibyte characters, wide string literals and wide character constants cannot be used.

- **Functional specification**
  The multibyte character refers to a description like `‘ab’`.
  The wide character constant refers to a description like `L‘AB’`.
  The wide string literal refers to a description like `L“ABCD”`.
  If any of these characters or strings is detected, SQMlint reports it as a complaining.

7.9. Rule 9 (Not supported)

(required): Comments shall not be nested.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.10. Rule 10 (Not supported)

(advisory): Sections of code should not be 'commented out'.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.11. Rule 11 (Not supported)

(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 characters significance and case sensitivity are supported for external identifiers.

- **Interpretation**

  Any identifier cannot have the same name in up to 31 characters.

- **Functional specification**

  This rule is not supported.

  Our supported C compilers and linkers support identifiers consisting of more than 31 characters. They also discriminate between the uppercase and lowercase letters.

7.12. Rule 12

(advisory): No identifier in one name space shall have the same spelling as an identifier in another name space.

- **Interpretation**

  The name space can be classified into four: labels, tags, members and other identifiers (function name, variable name other than member, enumerator name and typedef name). This rule means that any name space, irrespective of its kind, cannot have the same spelling (same name).

  The situation where the same spelling is used in name spaces of the same kind is not the subject of this rule. For example, this applies to the case where two or more different structures have members with the same spelling.

  This rule may be interpreted as accepting use of the same identifier in two or more name spaces providing it has different scopes in each.

- **Functional specification**

  If any label, tag, member or other identifier is found that has the same spelling as an identifier in another name space in the same scope, SQMlint reports it as a complaining.

Example:

```
void func(void)
{
    struct tag { SINT32 m; }s;
    SINT32 tag; /* Although duplicate tag and variable names are accepted in
                   ISO9899:1990 specifications, they deviate from rule 12.
*/
}
```

No warnings are output even when members with the same name are used in different structures in the same scope.

Example:

```
struct tag1 { SINT32 m; }s1;
struct tag2 { SINT32 m; }s2;
 /* The member variable m in the structure tag2 and the member variable m
    in the structure tag1 may have the same name.  */
```

7.13. 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.

- **Interpretation**

  To allow for compiler-independent C source code, this rule is intended to clarify the data size. Therefore, all basic types, not just the above-captioned types, but also including *signed int*, are checked to see if they are *typedef*'d.

- **Functional specification**

  If any names of basic types are used in declarations or cast expressions directly without being typedef'd, SQMlint reports them as a complaining.

  Reported type names are detailed below.

  (1) *char*, *int*, *short*, *long*, *float*, *double*, *long double*, *_Bool*(NC only), *long long*(NC only)

  Example:

  ```
  int  i;
  int* p;
  int  ary[5];
  ```

  (2) *signed* and *unsigned* types of (1)

  Example:

  ```
  signed int  si;
  ```

  (3) Statements where types are implicitly declared to be *int*

  Example:

  ```
  const   i;
  signed  func(void);
  ```

7.14. Rule 14

(required): The type char shall always be declared as unsigned char or signed char.

- **Interpretation**

  As captioned.

- **Functional specification**

  If a plain-char (without *unsigned* or *signed*) is used in a declaration or a cast expression, SQMlint reports it as a complaining. Similarly, if a plain-char, is used for the object pointed to by a pointer, SQMlint reports it as a complaining.

- **Precaution**

  If the type *char* is used after being *typedef*'d as described below, SQMlint reports only where the typedef name is defined but not where the typedef name is used.

Example:

```
typedef  char  CHAR;  /* Complaining */
CHAR  c;              /* No Complaining */
char*  p;             /* Complaining */
```

7.15. Rule 15 (Not Supported)

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

- **Interpretation**

  The implementations of our supported C compilers are compliant with IEEE754.

- **Functional specification**

  This rule is not supported.

7.16. Rule 16 (Not Supported)

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

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.17. Rule 17

(required): *typedef* names shall not be reused.

- **Interpretation**

  The *typedef* name cannot be reused in any code or file.

- **Functional specification**

  If the *typedef* name is found the same as any one of the identifiers in the entire scope of a translation unit, SQMlint reports it as a complaining.

- **Limitation**

  Inspection is not performed across the translation units.

7.18. Rule 18

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

- **Interpretation**

  There will be a case where constants are implicitly converted when used in expressions. The captioned appropriate suffix (e.g., `u`, `l`, `ul` or `f`) refers to the one that can represent the type to which constants are converted.

- **Functional specification**

  If any constant used in an expression meets all of the conditions described below, SQMlint reports it as a complaining.
  - The constant is implicitly converted.
  - Although its converted type can be specified by a suffix, No suffix or a suffix that is different from its type is used for the constant.
  - Operand constant of the unary + operator, operand constant of the unary – operator, constant enclosed with round bracket, and those combinations.

Example 1:

```
unsigned int ui;
ui = ui + 3;  /* Complaining, because 3 (int type) is converted
                   to the unsigned int type */
```

This should be correct to

```
ui = ui + 3u;
```

Example 2:

```
long l;
l = -3;  /* Complaining, because -3 (int type) is converted
                   to the long type */
```

This should be correct to

```
l = -3L;
```

- **Precaution**

Example 1:

```
FLOAT f;
f = f + 1.0;
```

Because the value 1.0 has type *double*, f is extended to *double*. The value 1.0 is not converted and operated on directly as being *double*, this is not considered as a deviation.

Example 2:

```
UINT32 l;
l = l + 'a';  /* Because The letter 'a' is not numeric, it is excluded from
                 those inspected. */
```

7.19. Rule 19

(required): Octal constants (other than zero) shall not used.

- **Interpretation**
  Octal constants often cause a descriptive error when writing constants because they are not easily distinguishable from decimal constants. Therefore, octal constants other than zero cannot be used.

- **Functional specification**
  If any octal constants other than zero are used, SQMlint reports them as a complaining.

7.20. Rule 20

(required): All object and function identifiers shall be declared before use.

- **Interpretation**
  This rule prohibits the following:
  - To call functions while they are not declared or before they are declared
  - To use undeclared variables
    (Use of undeclared variables results in generating a compile error in ISO9899:1990-compliant C compilers.)

- **Functional specification**
  If any undeclared variables or functions are used, SQMlint reports them as a complaining.

7.21. Rule 21

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

- **Interpretation**

  If a scope has another scope in it, the scopes inside and outside of another are interpreted as an inner scope and an outer scope, respectively. This rule prohibits declaring, for example, variables in a function that have the same name as one of variables declared at file scope. However, identifiers and members may have the same name.

- **Functional specification**

  If any identifier in a function has the same name as another identifier in an outer scope, SQMlint reports it as a complaining.

Example:

```
SINT32 val;
void func(void)
{
    SINT32 val;  /* Because the name of this variable val and that of
                    the global variable val are the same,  this deviates from
                    rule 21. */
}
```

- **Precaution**

  The following case is excluded from those to be inspected, because the inner variable name is not hidden.

Example:

```
void func(void)
{
    SINT32 val;
}
SINT32 val;  /* The global variable val does not hide
                the upper local variable val.  */
```

- **Limitation**

  Inspection is not performed across the translation units.

7.22. Rule 22

(advisory): Declarations of objects should be at function scope unless a wider scope is necessary.

- **Interpretation**
  If the variables defined in file scope (not including *extern*-declared variables) are not going to be used in more than one function, they should be defined in function scope.

- **Functional specification**
  If the variables defined in file scope (not including *extern*-declared variables) are being used in more than one function, SQMlint reports them as a warning.

- **Limitation**
  Inspection is not performed across the translation units.

7.23. Rule 23 (Not Supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.24. Rule 24

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

- **Interpretation**
  Identifiers with the same name cannot have a different linkage (e.g., *extern* or *static*) irrespective of their scope.

- **Functional specification**
  If any identifiers with the same name have a different linkage, SQMlint reports them as a complaining.

Example:

```
static SINT32 a;
SINT32 a;  /* Because the variable a is tentatively defined after being
                    static-declared, it deviates from rule 24. */

static SINT32 b;
extern SINT32 b; /* Because the static declaration is followed by
                         an extern declaration,  it deviates from rule 24. */

static SINT32 s1;
static SINT32 s2;
void func(void)
{
   extern SINT32 s1; /* Because the static declaration is followed by an
                             extern declaration,  it deviates from rule 24.  */
   SINT32 s2; /* Although the variable 's2' does not deviate from rule 24,
                    it deviates from rule 21. */
```

7.25. Rule 25 (Not supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.
  If there is a deviation from rule 25, an error is assumed in the linker.

7.26. Rule 26 (Not Supported)

(required): If objects or functions are declared more than once they shall have compatible declarations.

- **Interpretation**
  Incompatible declarations between two different files such as “`extern int a;`” in one file and “`extern float a;`” in another are not accepted.

- **Functional specification**
  This rule is not supported.
  If variables declared in the same file have different types, SQMlint reports in complain level.

7.27. Rule 27 (Not Supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.28. Rule 28

(advisory): The register storage class specifier should not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  If any *register* storage class specifier is used, SQMlint reports it as a complaining.

Example:

```
void  func( register  SINT32  arg)    /* Complaining */
{
    register  SINT32  i;   /* Complaining */
}
```

7.29. Rule 29

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

- **Interpretation**

(1) If when initializing the structure or union that is given a tag, the expressions to initialize its members cannot be assigned to its members, a deviation from this rule is considered. This is to prevent unintended use of a structure or union in cases where a different structure or union with the same tag name as in one file is defined in another file.

Note that this causes a compile error in ISO9899:1990-compliant C compilers.

Example:

```
  struct S1 { SINT32 i; } s1 = { "abc" }; /* Complaining
                                    and at the same time a compile error */
  struct { SINT32; } s2 = { "abc" }; /* Compile error only,
                                          because no tags are given  */
  struct S2 { SLONG l; } s3 = { 0ul };  /* Excluded from those
                                                to   be   inspected
*/
  struct S3 { ULONG ul; } s4 = { (ULONG)"abc" }; /* Excluded
                                          from those to be inspected */
  struct T t;
  void func(void)
  {
     struct S1 s = t;       /* Complaining and at the same time
                               a compile error  */
  }
```

(2) Initializing variables of an enumerated type with constants or expressions that are not an enumerator is interpreted as deviating from this rule.

Example:

```
  enum NUM { ONE = 1, TWO = 2 };
  enum NUM n = 1;  /* Complaining */
```

(3) Casting constants or variables of an integer type with an enumerated type is also interpreted as deviating from this rule.

Example:

```
  enum NUM { ONE = 1, TWO = 2 };
  SINT32 n = 2;
  (enum NUM)1;  /* Complaining */
  (enum NUM)n;  /* Complaining */
```

- **Functional specification**

  If variables of an enumerated type are initialized with constant expressions that are not an enumerator, SQMlint reports them as a complaining.

  If constants or variables of an integer type are cast with an enumerated type, SQMlint reports them as a complaining.

  If when initializing the structure or union that is given a tag, the expressions to initialize its members cannot be assigned to its members, SQMlint reports them as a complaining.

  If structures or unions are initialized with a different structure or union, SQMlint reports them as a complaining.

7.30. Rule 30 (Not supported)

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

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.31. Rule 31

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

- **Interpretation**

  To initialize arrays or structures with an initializer list (a list of expressions separated by a comma, each representing the initial value), braces must be used to indicate that array or structure.

  Example:

```
struct S {
     struct T {  SINT32  i;  SINT32  j;  }  t;
     SINT32  x;
} s = {  { 1,  2 },   3  };  /* Braces {1,  2} are required */
```

- **Functional specification**

  If any initializer list does not include enclosing braces, indicating the initializer list part to initialize structures and that to initialize arrays, SQMlint reports it as a complaining.

  However, this rule does not apply to the initializers (expressions representing the initialize value) that initialize unions. Nor does this rule apply to the case where the initializers are a string literal (string enclosed in double-quotes).

  Example:

```
  SINT32 arr[2][3] = { 1,2,3, 4,5,6+1 };  /* Complaining */
```

  This should be correct to

```
  SINT32 arr[2][3] = { {1,2,3}, {4,5,6+1} };
```

7.32. Rule 32

(required): In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.

- **Interpretation**

  As captioned.

- **Functional specification**

  It is in only one of the following cases that the '=' can be used in an enumerator. If any enumeration declaration is detected where neither cases hold true, SQMlint reports it as a complaining.
  - Only the first enumerator is initialized with '='
  - All enumerators are initialized with '='

Example:

```
/* No complaining */
enum E1 { a1, b1, c1 };
enum E2 { a2 = 1, b2, c2 };
enum E3 { a3 = 1, b3 = 3, c3 = 5 };

/* Complaining */
enum E4 { a4, b4 = 3, c4 };
enum E5 { a5 = 1, b5, c5 = 5 };
```

7.33. Rule 33

(required): The right hand operator of a && or || operator shall not contain side effects.

- **Interpretation**

  The operators && and || are characteristic in that the evaluation result of the left-hand side expression determines whether or not to evaluate the right-hand side expression. Writing an expression that may cause side effects when the right-hand side expression is evaluated will result in global variables becoming rewritten or not rewritten depending on the evaluation result of the left-hand side expression. The purpose of this rule is to prevent such indeterminate behaviors.

  Following operations are side effects:

  – Modifying a abject(as when modifying a variable or the object pointed to by a pointer)
  – Accessing a volatile object(as when referencing or modifying the volatile object)
  – Modifying a file
  – Calling a function that does any of the above operations

- **Functional specification**

  If any expression is detected where side effects are likely to occur on the right-hand side of the operator && or ||, SQMlint outputs a report. In this case, if side effects on the right-hand side expression are caused by only a function call, SQMlint reports it as a warning. Otherwise, it reports it as a complaining.

Example:

```
volatile UINT16 us;
SINT16 n;
if ( (n == 0) || is_empty() ) { /* Warning */
if ( (n == 0) && (us == 0) ) {  /* Complaining */
```

7.34. Rule 34

(required): The operands of a logical && or || shall be primary expressions.

- **Interpretation**

  Here, the primary expression refers to variables, functions, enumerators and constants that are enclosed in parentheses. By limiting the left-hand and right-hand sides of the operators && and || to only the primary expressions, this aims to guarantee that they are logical expressions intended and make them easily to read.

- **Functional specification**

  Unless either side of the operator && or || is a primary expression, SQMlint reports it as a complaining.

Example 1:

```
 if ( a==0 && b==0)  {          /* Complaining */
```

This should be correct to

```
 if ((a==0) && (b==0)) {
```

Example 2:

```
 if (is_empty() || is_zero) {    /* No complaining because both function
                                       and variable are primary expressions */
```

Example 3:

```
 if ((a==0) && (b==0) || (c==0)) { /* Complaining because the expression
                                        on the left-hand side of ||
                                        is not a primary expression */
```

This should be correct to

```
 if (((a==0) && (b==0)) || (c==0)) {
```

7.35. Rule 35

(required): Assignment operators shall not be used in expressions which return Boolean values.

- **Interpretation**

  The following expressions are interpreted as an expression that returns Boolean values:

  - ◆ Conditional expression
    - Expression in ( ) of *if* or *else if*
    - Expression in ( ) of *while*
    - Second expression in parentheses of *for*
    - Expression in the first operand of ?:
  - ◆ Expressions on the left-hand and right-hand sides of a logical combination operator (&& or ||)
  - ◆ Expression on the right-hand side of the operator !

- **Functional specification**

  If an assignment operator is included in any conditional expression, in the left-hand or right-hand side expression of the logical combination operator (&& or ||), or in the right-hand side expression of the logical negation operator (!), SQMlint reports it as a complaining.

7.36. Rule 36

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

- **Interpretation**

  A statement `if(a & b)`, for example, is regarded as an abbreviated version of `if((a & b)!=0)`, but it may perhaps be intended to mean `if(a&&b)` by misusing the operators & and && or perhaps erroneously written. This rule detects such misused statements in the following expressions:

  - ◆ Conditional expression
    - Expression in ( ) of *if* or *else if*
    - Expression in ( ) of *while*
    - Second expression in parentheses of *for*
    - Expression in the first operand of ?:
  - ◆ Expressions on the left-hand and right-hand sides of a logical combination operator (&& or ||)
  - ◆ Expression on the right-hand side of the operator !

- **Functional specification**

  If the last operator to be evaluated in any conditional expression, in the left-hand or right-hand side expression of the logical combination operator (&& or ||), or in the right-hand side expression of the logical negation operator (!) is &, | or ~ of a bitwise operator, SQMlint reports it as a complaining.

7.37. Rule 37

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

- **Interpretation**

  As captioned.

  Performing ^ operations on signed integer types is also thought to be prohibited.

- **Functional specification**

  If the types of the following expressions are signed integer types, SQMlint reports them as a complaining.

  - The right-hand side expression of a bitwise operator (~)
  - The left-hand and/or right-hand side expression of a bitwise operator (& , ^ or |)
  - The left-hand side expression of a bitwise shift operator(<< or >>)
  - The left-hand and/or right-hand side expression the compound assignment operator of a bitwise operator (&=, ^= or |=)
  - The left-hand side expression of the compound assignment operator of a bitwise shift operator (<<= or >>=)

  Example:

  ```
  SINT32 si;
  si = si >> 1u;  /* Shift operations performed on signed int types constitute
                     a deviation from this rule */
  ```

- **Precaution**

  If the expression to be inspected is a signed constant value, SQMlint also reports it as a complaining

  Example:

  ```
  0xFFFF << 4u;
  ```

  Here, because the `0xFFFF` to be inspected for this rule is the *int* type and a signed constant value, SQMlint reports it as a complaining. To avoid this deviation, the expression should be corrected to the one shown below.

  ```
  0xFFFFU << 4u;
  ```

7.38. 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).

- **Interpretation**

  As captioned.

- **Functional specification**

  If the right-hand side operand of a shift operator (<<, >>, <<= or >>=), the shift count, is a constant expression and the value of that expression is equal to or greater than the width in bits represented by the type of the left-side expression, SQMlint reports it as a complaining.

7.39. Rule 39

(required): The unary minus operator shall not be applied to an unsigned expression.

- **Interpretation**

  As captioned.

- **Functional specification**

  If the right-hand side expression of the unary operator '-' is an unsigned type of expression, SQMlint reports it as a complaining.

Example:

```
UINT16 us;
-us;  /* Complaining */
```

7.40. Rule 40

(advisory): The sizeof operator should not be used to expressions that contain side effect.

- **Interpretation**

  Following operations are side effects:
  - Modifying a abject(as when modifying a variable or the object pointed to by a pointer)
  - Accessing a volatile object(as when referencing or modifying the volatile object)
  - Modifying a file
  - Calling a function that does any of the above operations

- **Functional specification**

  If the operand of the *sizeof* operator (`sizeof(`this expression)) contains any expression that is likely to cause side effects, SQMlint outputs a report. If no function calls are included in the relevant expression, SQMlint reports them as a complaining. If function calls are included, SQMlint reports them as a warning.

7.41. Rule 41 (Not supported)

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

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.42. Rule 42

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

- **Interpretation**

  As captioned.

- **Functional specification**

  If a comma operator is used in expressions other than those in ( ) of *for* statements, SQMlint reports it as a complaining.

7.43. Rule 43

(required): Implicit conversion which may result in a loss of information shall not be used.

- **Interpretation**

  The conversions that result in a loss of information refer to the case where the converted type cannot represent the pre-conversion value. If some value is assigned to, or has parameters or returns passed to, or is initialized to a type lower in precision or smaller in size than that, a loss of information may be incurred by conversion to such a type.

  Conversions in the following cases will cause a loss of information:

  - When a variable is converted to a type smaller in size than that
  - When a signed type is converted to an unsigned type
  - When an unsigned type is converted to a signed type
  - When a floating-point type is converted to an integer type
  - When a constant expression is converted to a type that cannot hold the value of the constant

- **Functional specification**

  If a conversion that will cause a loss of information is detected in the following context, SQMlint reports it as a complaining:

  - When a value is assigned
  - When passed as a parameter
  - When returned as a return value
  - When initialized

  If a constant expression is assigned to or initialized to a bit-field that cannot hold the value of the constant, because in this case too a loss of information is incurred, SQMlint also reports it as a complaining.

  This rule is inspected for integer, floating-point and enumerated types and pointers.

7.44. Rule 44

(advisory): Redundant explicit casts should not be used.

- **Interpretation**

  The following two cases are interpreted as redundant explicit casts:
  - When cast to the same type as before
  - When cast to the same type as will be assumed after an implicit conversion

- **Functional specification**

  A value may be cast to the same type as before, either directly or through multiple casts. In the former case, SQMlint reports it as a complaining, whereas in the latter case, SQMlint reports it as a warning.

  When a value is cast to the same type as will be assumed for it after an implicit conversion, SQMlint reports it as a complaining.

Example:

```
SCHAR c;
SINT16 s;
SINT32 n;
SLONG l;
ULONG ul;
DOUBLE d;
(SINT32)n;            /* Complaining */
(SLONG)(SLONG)n;      /* Complaining */
(SINT16)(SLONG)s;     /* Warning */
(SINT16)(SCHAR)s;     /* Warning */
       /* If the upper digits needs to be set to 0, a mask should be used */
n = (SINT32)c + (SINT32)c;   /* Complaining (same as will be assumed
                                after implicit conversion)*/
n = (SINT32)c + c;           /* Complaining (same as will be assumed
                                after implicit conversion) */
n = c + (SINT32)c;           /* Complaining (same as will be assumed
                                after implicit conversion) */
d = (DOUBLE)1 / (DOUBLE)3;   /* Complaining for the second cast (same as
                             will be assumed after implicit conversion) */
d = (DOUBLE)1 / 3;    /* No complaining */
d = 1 / (DOUBLE)3;    /* No complaining */
ul = (ULONG)1;        /* Complaining */
```

7.45. Rule 45

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

- **Interpretation**
  As captioned.

- **Functional specification**
  If a cast by a pointer type or, a cast on a pointer or on a function call that returns a pointer is detected, SQMlint reports it as a complaining.

7.46. Rule 46

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

- **Interpretation**

  The orders of evaluation for expressions from one sequence point (function call ( ), &&, ||, ?:, ;, or comma operator) to the next sequence point is unspecified. If expressions that will cause the same side effect are included in between, they will be evaluated in varying orders depending on the compilers used and the intended result may not be obtained. This rule prohibits statements that will cause such a problem.

  Following operations are side effects:
  - Modifying a abject(as when modifying a variable or the object pointed to by a pointer)
  - Accessing a volatile object(as when referencing or modifying the volatile object)
  - Modifying a file
  - Calling a function that does any of the above operations

- **Functional specification**

  If the following expressions are included between one sequence point to the next sequence point, SQMlint outputs a report:

  (1) Expressions that modify the same variable (assignment operation, ++, and -- operation)
  (2) Expressions that contain the same *volatile* variable
  (3) More than one function call

  However, if one of the following cases holds true, SQMlint reports it as a warning. Otherwise, SQMlint reports it as a complaining.
  The cases of (1) and (2) above, where the object is a structure or union member variable
  The cases of (1) and (2) above, where the object is an array element
  The case of (3) above

  Example:

```
ULONG ul = 10UL;
volatile ULONG v;
ul = (ul++) + ul;     /* Complaining */
ul = v + v;           /* Complaining */
ul = func (pop(), push());  /* Warning */
```

- **Precaution**

  The following example is not considered to be a deviation from the rule, because arguments are always evaluated prior to the relative function calls.:

```
x = func( y = z / 3 );
```

- **Limitation**

  Inspection is not performed in cases where the same object is pointed to by pointers and it has side effects. An example of this case is shown below.

  Example:

  ```
  *p = *p++ + *p++;
  ```

7.47. Rule 47 (Not supported)

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

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.48. Rule 48

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

- **Interpretation**

  This rule is provided to prevent a loss of information as will be caused by arithmetic operations performed with an unintended low precision due to implicit conversions.
  If the result of an operation is operated on with a higher precision than that next, the final result may not be thought to have been derived by calculations with the expected precision.

- **Functional specification**

  If the result of an arithmetic four-rules operation (+, -, * or /) is used in one of the operations shown below with a higher precision than that, SQMlint reports it as a complaining.
  - Arithmetic operation (+, -, *, / or %)
  - Assignment operation (=, +=, -=, *=, /=, or %=)
  - Cast

  If the result of an arithmetic four-rules operation is used to initialize a variable whose type has a higher precision than that, SQMlint also reports it as a complaining.

Example:

```
UINT16 x = 65535u;
UINT16 y = 10u;
UINT32 ul1 = (UINT32) (x + y);  /* Complaining (x + y equals 9) */
```

To change the above statement as intended, correct it to

```
UINT32 ul2 = (UINT32)x + (UINT32)y;  /* x + y equals 65545 */
```

7.49. Rule 49

(advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

- **Interpretation**

  Expressions whose types are effectively Boolean are thought to be such expressions that the last operator to be evaluated is &&, | |, !, <, >, <=, >=, == or !=.
  If the following expressions are not an expression whose type is effectively Boolean, comparisons with zero are considered necessary:
  - ◆ Conditional expression
    - Expression in ( ) of *if* or *else if*
    - Expression in ( ) of *while*
    - Second expression in parentheses of *for*
    - Expression in the first operand of ?:
  - ◆ Expressions on the left-hand and right-hand sides of a logical combination operator (&& or | |)
  - ◆ Expression on the right-hand side of the operator !

- **Functional specification**

  If the last operator to be evaluated in any conditional expression, in the left-hand and right-hand side expressions of the logical combination operator (&& or | |), or in the right-hand side expression of the logical negation operator (!) is none of the operators &&, | |, !, <, >, <=, >=, == or !=, SQMlint reports it as a complaining.

Example:

```
ULONG *pl;
if (pl) {     /* Complaining */
```

This should be corrected to

```
if (pl != 0) {
```

7.50. Rule 50

(required): Floating-point variables shall not be tested for exact equality or inequality.

- **Interpretation**

  As captioned.

- **Functional specification**

  If the either side expression of the operator == or != is a floating-point type, SQMlint reports it as a complaining.

Example:

```
if (d == 1.0) {   /* Complaining */
```

7.51. Rule 51

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

- **Interpretation**
  If the evaluation of a constant unsigned integer expression results in overflowing or underflowing the range of values that representable for the type of the result, wrap-around is considered for the result.

- **Functional specification**
  If the evaluation of a constant expression whose result type is an unsigned integer constant results in an overflow or underflow, SQMlint reports it as a complaining.

  Example:
  ```
  ULONG ul = 3UL – 8UL;  /* Complaining (underflow) */
  ```

- **Limitation**
  Inspection is not performed on constant expressions in macros.

  Example:
  ```
  #if (1u - 2u)
  ```

7.52. Rule 52 (Not supported)

(required): There shall be no unreachable code.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.53. Rule 53

(required): All non-null statements shall have a side-effect.

- **Interpretation**

  As captioned.

  Following operations are side effects:
  - Modifying a abject(as when modifying a variable or the object pointed to by a pointer)
  - Accessing a volatile object(as when referencing or modifying the volatile object)
  - Modifying a file
  - Calling a function that does any of the above operations

- **Functional specification**

  If any expression which is not a function call and does not produce a side effect is detected, SQMlint reports it as a complaining. However, no reports are output when function calls that do not produce a side effect are encountered.

7.54. 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.

- **Interpretation**

  Neither comments nor expressions can be written before and after a null statement (;) on the same line as the statement.

```
a = 1; ;  /* Complaining */
; a = 1;  /* Complaining */
if (1);   /* Complaining */
;;        /* Complaining */
```

- **Functional specification**

  If any expression is written before or after a null statement, SQMlint reports it as a complaining.

- **Limitation**

  Inspection cannot be performed in cases where a null statement is preceded or followed by a comment or macro description.

Example:

```
#define TEXT
void func(void)
{
   TEXT;  /* In this case, inspection cannot be performed. */
}
```

7.55. Rule 55

(advisory): Labels should not be used, except in switch statements.

- **Interpretation**
  As captioned.

- **Functional specification**
  If labels are defined, SQMlint reports them as a complaining.

7.56. Rule 56

(required): The goto statement shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  If *goto* statements are used, SQMlint reports them as a complaining.

7.57. Rule 57

(required): The continue statement shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  If *continue* statements are used, SQMlint reports them as a complaining.

7.58. Rule 58

(required): The break statement shall not be used (except to terminate the cases of a switch statement).

- **Interpretation**

  As captioned.

- **Functional specification**

  If *break* statements are used, except to terminate the conditions of a *switch* statement, SQMlint reports them as a complaining.

  If *break* statements are used in local blocks of a *case* clause, SQMlint reports them as a complaining.

  Example:

```
case 1:
{
    a = 0;
    break;  /* Complaining */
}
```

7.59. 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.

- **Interpretation**

  As captioned.

  A similar rule applies to *switch* statements too.

- **Functional specification**

  Unless said statements are enclosed in { }, SQMlint reports them as a complaining.

7.60. Rule 60

(advisory): All if, else if constructs should contain a final else clause.

- **Interpretation**
  This rule is interpreted as shown by examples below.

Example:

```
if (a == 1) {
} else if (a < 1) {
}  /* An else statement is always required here.  */

if (x < 0) {
}  /* An else statement may be omitted here.  */
```

- **Functional specification**
  If the matching *else* for *else if* is not found, SQMlint reports it as a complaining.

7.61. Rule 61

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

- **Interpretation**
  As captioned.

- **Functional specification**
  If any non-empty *case* clause is not followed by a *break* statement at the end of it, SQMlint reports it as a complaining.

7.62. Rule 62

(required): All switch statements should contain a final default clause.

- **Interpretation**
  As captioned.

- **Functional specification**
  If any *switch* statement is not followed by a *default* clause at the end of it, SQMlint reports it as a complaining.

7.63. Rule 63

(advisory): A switch expression should not represent a Boolean value.

- **Interpretation**

  Expressions that represent a Boolean value are assumed to be such an expression that the last operator to be evaluated is &&, ||, !, <, >, <=, >=, == or !=.

  The *switch* statement shown below is interpreted as deviating from this rule, because its processing involves determining whether the variable 'a' has the value 1 or not.

  Example:

  ```
  switch (a) {
  case 1:
       break;
  default:
       break;
  }
  ```

- **Functional specification**

  If the last operator to be evaluated in a *switch* statement is one of the operators &&, ||, !, <, >, <=, >=, == or !=, SQMlint reports it as a complaining.

  If a *switch* statement consists only of one *case* clause and a *default* clause, SQMlint reports it as a complaining.

7.64. Rule 64

(required): Every switch statements shall have at least one case.

- **Interpretation**

  As captioned.

- **Functional specification**

  If a *switch* statement does not have any *case*, SQMlint reports it as a complaining.

7.65. Rule 65

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

- **Interpretation**
  Loop counters are interpreted as a variable used in a conditional expression that terminates the relevant loop.

- **Functional specification**
  In all cases where any variable or member variable of a floating-point type is used in the conditional expression of a *while* statement or in the second expression of a *for* statement, SQMlint reports it as a warning.

7.66. Rule 66 (Not supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.67. Rule 67 (Not supported)

(advisory): Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.68. Rule 68

(required): Functions shall always be declared at file scope.

- **Interpretation**

  As captioned.

- **Functional specification**

  If functions are declared at other than file scope, SQMlint reports them as a complaining.

  If a function is called while no prototypes are declared for the function, SQMlint also reports it as a complaining.

Example:

```
void main(void)
{
   func(); /* Complaining */
}
```

7.69. Rule 69

(required): Functions with variable numbers of arguments shall not be used.

- **Interpretation**

  As captioned.

- **Functional specification**

  If "..." is included in function declarations or function definitions, SQMlint reports it as a complaining.

Example:

```
void  func1( const SCHAR* fmt, ... );   /* Complaining */
void  func2( SINT32 i, ... )            /* Complaining */
{
}
```

7.70. Rule 70

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

- **Interpretation**

  As captioned.

- **Functional specification**

  Only when functions are called from themselves directly, SQMlint reports them as a complaining.

- **Limitation**

  Inspection is not performed in cases when functions are called from themselves indirectly.

7.71. Rule 71

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

- **Interpretation**

  Prototype declarations are required before functions can be defined.
  Prototype declarations are required before functions can be called, and it deviates from rule 20 also.

- **Functional specification**

  If no prototype is declared for a function before defining or calling the function, SQMlint reports it as a complaining.

  Example:

  ```
  void func(void) { }  /* Complaining */
  ```

  This should be corrected to

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

  If no prototype is declared before calling a function, SQMlint reports it as a complaining for this rule as well as for rule 20.

7.72. 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.

- **Interpretation**

  Parameter types and return types in prototype declarations and those in function definitions must be completely identical.

- **Functional specification**

  Unless parameter types and return types in prototype declarations and those in function definitions are completely identical, SQMlint reports them as a complaining.

- **Limitation**

  Inspection is not performed across the translation units.

7.73. Rule 73

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

- **Interpretation**

  The identifiers here mean the names of parameters. In cases where multiple parameters are present in one prototype declaration, this rule prohibits coexistence of "parameters without a name" and "parameters with a name."

- **Functional specification**

  If "parameters without a name" and "parameters with a name" coexist in a prototype declaration, SQMlint reports them as a complaining.

  Because ellipsises (...) have no name, they are not inspected for this rule.

Example:

```
/* Complaining */
 void func1( UINT32 n, UINT32 );  /* only the first parameter
                                                       has the name */
 void func2( UINT32, UINT32 i );  /* only the second parameter
                                                       has the name */

/* No complaining */
 void func3( UINT32, UINT32 );      /* all parameters have no name */
 void func4( UINT32 n, UINT32 i );  /* all parameters have the names */
 void func5( const UINT32* fmt, ... );  /* No Complaining */
```

7.74. Rule 74

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

- **Interpretation**
  If identifiers (parameter names) are given for any parameters in a prototype declaration, they must have the same name as identifiers for parameters in a function definition.

- **Functional specification**
  If any identifier has different names in prototype declaration and function definition, SQMlint reports it as a complaining.

7.75. Rule 75

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

- **Interpretation**
  In ISO9899:1990, if type names such as *char*, *int* or *short* that represent the return type are omitted in function declarations, type *int* is implicitly assumed for that return type. Declarations of such an implicit return type are interpreted as deviation this rule.

- **Functional specification**
  If no return type is found in a function declaration or definition or in a function type cast, SQMlint reports it as a complaining.
  In the following cases where no type names are given and only a type qualifier (*const* or *volatile*) or a storage class (*static* or *extern*) is written for return type, the statement is not regarded as explicitly specifying a return type, and therefore SQMlint reports it as a complaining.

Example:

```
/* Complaining */
func1() { }             /* Return type implicitly becomes int */
const func2() { }       /* Return type implicitly becomes const int */
const func3();          /* Return type implicitly becomes const int */
static func4();         /* Return type implicitly becomes int */
```

7.76. Rule 76

(required): Function with no parameters shall be declared with parameter type void.

- **Interpretation**

  A function with no parameters refers to a function type whose parameter list is empty as in the case of "`void func();`". Because compilers do not check the parameter types of such a function, this rule stipulates that *void* be written for parameters.

- **Functional specification**

  If no parameter list is found in a function declaration or definition or in a cast with a type including function types, SQMlint reports it as a complaining.

7.77. 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.

- **Interpretation**

  This rule is interpreted as requesting that argument types specified when calling a function be compatible with the unqualified expected types of parameters (not including *const* and *volatile*) that are defined in the function prototype. Accordingly, arguments that occur implicit conversions are interpreted as deviation from the rule.

- **Functional specification**

  If the argument types specified when calling a function and the corresponding parameter types defined in the function prototype, except qualifiers (*const* and *volatile*), do not match when they are compared, SQMlint reports it as a complaining.

Example:

```
void  func(const SLONG);
void  xxx(void)
{
    SINT32  i;
    SLONG  l;
     func( l );   /*  No complaining  */
     func( i );   /*  Complaining (int and signed long are incompatible)  */
```

7.78. Rule 78

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

- **Interpretation**

  As captioned.

- **Functional specification**

  If the number of arguments that is given when calling a function does not match the number of parameters defined in the function prototype, SQMlint reports it as a complaining. Note, however, that this causes a compile error in ISO9899:1990-compliant C compilers.

- **Precaution**

  Function calls that make use of default arguments which is an extended feature are interpreted as deviating from this rule.

7.79. Rule 79

(required): The values returned by void functions shall not be used.

- **Interpretation**

  As captioned.

- **Functional specification**

  If calls to functions whose return type is *void* are used in a subexpression, SQMlint reports it as a complaining. Note, however, that this causes a compile error in ISO9899:1990-compliant C compilers.

7.80. Rule 80

(required): Void expression shall not be passed as function parameters.

- **Interpretation**

  As captioned.

- **Functional specification**

  If calls to functions whose return type is *void* are used as parameters to another function, or a variable declared as `void* p`; is passed to `func(*p)` as an argument, SQMlint reports it as a complaining. Note, however, that this causes a compile error in ISO9899:1990-compliant C compilers. It also deviates from rule 79.

7.81. Rule 81 (Not supported)

(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.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.82. Rule 82

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

- **Interpretation**

  If a function has a return value, the function can have only one *return* statement at the end of it. If the return type of the function is *void*, the function can either have no *return* statements at all or have only one *return* statement at the end of it.

- **Functional specification**

  (1) When the function has a return value

  If the function has multiple *return* statements or does not have a *return* statement at the end of it, SQMlint reports it as a complaining.

  (2) When the return type of the function is *void*

  If the function has multiple *return* statements or has a *return* statement at other than the end of it, SQMlint reports it as a complaining.

7.83. Rule 83

(required): For functions with non-void return type: i) there shall be one return statement for every exit branch (including the end of the program), ii) each return shall have an expression, iii) the return expression shall match the declared return type.

- **Interpretation**

  If a function has return type, the *return* statement must have an expression and the return expression must match the declared return type. Furthermore, there can be no statements after the last *return* statement of the function.

- **Functional specification**

  If the *return* statement does not appear at the end of the function, SQMlint reports it as a complaining.

  If the *return* statement does not have a return value, SQMlint reports it as a complaining.

  If the return value of the *return* statement does not match the declared return type, SQMlint reports it as a complaining.

7.84. Rule 84

(required): For function with void return type, return statements shall not have an expression.

- **Interpretation**

  As captioned.

- **Functional specification**

  If the *return* statement has a return value, SQMlint reports it as a complaining.

7.85. Rule 85

(advisory): Function called with no parameters should have empty parentheses.

- **Interpretation**

  Function names written without parentheses ( ) represent a function address. Such a description cannot be discriminated whether an intended function call is erroneously written or an address reference is intended. To prevent this problem, this rule is interpreted as recommending an explicit use of the unary operator “&” when function addresses are to be referenced.

  Expressions (`*func`) where the indirection operator (`*`) is operated on function names also are interpreted as deviating from the rule as does (*func* only).

  However, if such an expression is used in function calls (i.e., `(*func)()`) is not interpreted as deviating from the rule.

- **Functional specification**

  In cases when function names are included in an expression, if neither the unary operator & nor the () operator for function calls are found, SQMlint reports them as a complaining.

  However, when an expression is used as initializer(initial value of initialization) of the pointer to a function, it is not considered to be a violation.

7.86. Rule 86 (Not supported)

(advisory): If a function returns error information, then that error information should be tested.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.87. Rule 87 (Not supported)

(required): #include statements in a file shall only be preceded by other pre-processor directives or comments.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.88. Rule 88 (Not supported)

(required): Non-standard characters shall not occur in header file names in #include directives.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.89. Rule 88 (Not supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.90. Rule 90 (Not supported)

(required): C macros shall only be used for symbolic constants, function-like macros, type qualifiers and storage class specifiers.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.91. Rule 91 (Not supported)

(required): Macros shall not be #define'd and #undef'd within a block.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.92. Rule 92 (Not supported)

(advisory): A #undef should not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.93. Rule 93 (Not supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.94. Rule 94 (Not supported)

(required): A function-like macro shall not be 'called' without all of its arguments.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.95. Rule 95 (Not supported)

(required): Arguments to a function-like macro shall not contain tokes that look like pre-processing directives.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.96. Rule 96 (Not supported)

(required): In the definition of a function-like macro the whole definition, and each instance of a parameter, shall be enclosed in parentheses.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.97. Rule 97 (Not supported)

(advisory): Identifiers in pre-processor directives should be defined before use.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.98. Rule 98 (Not supported)

(required): There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.99. Rule 99

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

- **Interpretation**
  As captioned.

- **Functional specification**
  If *#pragma* is used, SQMlint reports it as a warning.

7.100. Rule 100 (Not supported)

(required): The defined pre-processor operator shall only be used in one of the two standard forms.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.101. Rule 101

(advisory): Pointer arithmetic should not be used.

- **Interpretation**
  Accessing the address result of increment(++) or decrement(--) operations to pointers or using [] to access a pointer as an array is not considered to be a problem. However, accessing the addresses obtained by any other arithmetic operations on the pointers may be considered to have a greater possibility of constituting an illegal access than the above. This rule prohibits such pointer arithmetic.

- **Functional specification**
  If any arithmetic operations on pointers other than increment(++) or decrement(--) operations are detected, SQMlint reports them as a complaining.
  However, use of the [] operator in the right-side expression of the ! operator or in a pointer is excluded from those inspected.

Example:

```
SINT32 a[10];
SINT32 *p = &(a[0]);
p++;            /* No complaining */
p[4] = 1;       /* No complaining */
*(p + 5) = 0;   /* Complain */
```

7.102. Rule 102

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

- **Interpretation**

  As captioned.

- **Functional specification**

  If pointer variables are declared in more than 2 levels, SQMlint reports them as a complaining.
  If expressions are cast to pointer types in more than 2 levels, SQMlint reports them as a complaining.
  If the indirection operator (`*`) is used more than two times successively (e.g., `***p`), SQMlint reports it as a complaining.

7.103. 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.

- **Interpretation**

  The "same array, structure or union" is interpreted as referring to the same array object, same structure object or same union object, respectively.
  The relational operators refer to <, >, <= and >=, with == and != not included.

- **Functional specification**

  Whenever relational operators used to compare pointers together are found, SQMlint reports it as a warning. This does not always mean that the reported statements deviate from rule 103. If the same array object, same structure object or same union object is pointed to, SQMlint also outputs a report.

Example:

```
SINT32* pi1, pi2;
pi1 > pi2;  /* Pointers for different objects cannot be compared */
```

7.104. Rule 104

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

- **Interpretation**

  This rule is interpreted as prohibiting use of a pointer to function whose address is not determined when compiled as part of an expression other than function calls, as well as casting with a pointer to function.

- **Functional specification**

  Unless declarations of pointer variables to functions are initialized by a constant expression or function name, SQMlint reports them as a complaining.

  If casts by pointer types to functions or casts to pointers to functions are detected, SQMlint also reports them as a complaining.

Example:

```
void func(void)
{
    void (*f1)(void) = &func;
        /*  This is correct because the pointer f1 to function is initialized by
            a pointer to function whose address is determined when compiled */
    void (*f2)(void) = f1;
       /*  Cannot be initialized by a pointer to function whose address is
           not determined when compiled  */
}
```

7.105. 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 return type.

- **Interpretation**

  This rule is interpreted as requesting that in the following cases where they respectively are pointers to functions, the function types pointed to by these two pointers should be compatible. Two function types of which the number of parameters, the corresponding types of parameters and the return types are identical are compatible.

  - Pointers from which values are assigned and pointers to which values are assigned
  - Parameters in prototype declarations and arguments when functions are called
  - Return values in function definitions and return values in *return* statements
  - Declaration of pointers to functions and expressions to initialize them

- **Functional specification**

  If function types are not compatible in the above interpretation, SQMlint reports them as a complaining. Note, however, that this causes a compile error in ISO9899:1990-compliant C compilers.

Example:

```
SINT32 func1(SINT32);
SINT32 func2();
SINT32 (fp*)(SINT32);

fp = &func1;    /* No Complaining */
fp = &func2;    /* Complaining */
```

7.106. 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.

- **Interpretation**

  This rule is interpreted as aiming to prevent the address of a local variable from being assigned to a pointer variable existing in a scope outside that scope or being used outside the scope as a return value.

- **Functional specification**

  If the address value of a local variable is assigned to a pointer variable existing in a scope outside that scope, SQMlint reports it as a complaining. If a *return* statement where the addresses of local variables are used as return values is detected, SQMlint also reports it as a complaining.

- **Limitation**

  Inspection is not performed in case of indirect assignments. The indirect assignments, for example, are the following cases.

  – when the address value of a local variable, after being assigned to a local pointer variable first, is assigned to a pointer variable existing in outer scope

  – when the address thus assigned to a pointer variable is used as a return value in a *return* statement

  Nor will inspection be performed in cases where the address value of a local variable is assigned to the object pointed to by a pointer.

Example:

```
SINT32** pp;
SINT32* p;
void func(void)
{
   SINT32 a;
   SINT32* pa=&a;

   p = &a;      /* Complaining */
   *pp = &pa;   /* Not inspected */
}
```

7.107. Rule 107 (Not supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.108. Rule 108

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

- **Interpretation**
  To "specify all members" here means assigning names (variable names) to all the members of a structure or union when defining the structure or union.
  In bit-field declarations, it is possible to declare "structures consisting only of padding, with members assigned no names." However, if this structure variable is referenced or modified, program behavior is undefined. This rule is interpreted as aiming to prevent such undefined behavior.

  Example:

```
struct  S {
   unsigned  int  :1;
   unsigned  int  :1;
   unsigned  int  :1;
   unsigned  int  :1;
}  s;     /*  Hereafter, what will result by referencing
                                   or modifying this variable s is undefined  */
```

- **Functional specification**
  If none of the structure or union members have names, SQMlint reports it as a complaining.

7.109. Rule 109 (Not supported)

(required): Overlapping variable storage shall not be used.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.110. Rule 110

(required): Unions shall not be used to access the sub-parts of larger data types.

- **Interpretation**

  In cases where the following conditions are met at the same time, unions are regarded as being used to "access the sub-parts of larger data types."
  - All members included in a union have the same size.
  - That union contains one or more structures or arrays.

  However, even when members included in parts of a union have the same size, the union is not regarded as being used to "access the sub-parts of larger data types" unless all of the union members are identical in size.

- **Functional specification**

  If unions are regarded as being used to "access the sub-parts of larger data types" in the above interpretation, SQMlint reports them as a complaining.

Example 1:

```
union U1 {
   unsigned char arr[4];
   unsigned long ul;
}; /* Complaining, because arr and ul use areas of the same size */
```

Example 2:

```
union U2 {
  struct {
      unsigned char b0:1;
      unsigned char b1:1;
      unsigned char b2:1;
      unsigned char b3:1;
      unsigned char b4:1;
      unsigned char b5:1;
      unsigned char b6:1;
      unsigned char b7:1;
   } bits;
   unsigned char all;
   struct {
      unsigned char c1:4;
      unsigned char c2:4;
   } other;
}; /* Complaining, because bits and all, other use areas of the same size */
```

Example 3:

```
union U3 {
   unsigned char arr[2];
   unsigned long ul;
}; /* No complaining, because arr and ul sizes are different */
```

7.111. Rule 111

(required): Bit fields shall only be defined to be of type unsigned int or signed int.

- **Interpretation**

  As captioned.

- **Functional specification**

  If bit fields are detected that are neither type *unsigned int* nor type *signed int*, SQMlint reports them as a complaining.

  Example:

```
struct S {
    int           b0: 1;  /* Complaining */
    signed int    b1: 2;  /* No complaining */
 };
```

7.112. Rule 112

(required): Bit fields of type signed int shall be at least 2 bits long.

- **Interpretation**

  This rule is interpreted as requesting that bit-fields of signed type should be 2 bits or more in width. Although rule 111 stipulates that bit-fields can only be defined to be of type *unsigned int* or *signed int*, some compilers allow the use of bit-fields of type *unsigned char* or *signed char*.

  Because bit-fields consisting only of a sign bit are considered not acceptable for security, it is desirable that inspection be performed on all signed types. Therefore, SQMlint outputs a report for all signed types.

- **Functional specification**

  If bit-fields in width of 2 bits or more are defined to be of a signed type, SQMlint reports them as a complaining.

7.113. Rule 113

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

- **Interpretation**

  Defining members without names for a structure (or union) is interpreted as deviating from this rule. However, in ISO9899:1990-compliant C language, only the bit-fields for paddings can be defined as unnamed members. Therefore, it is only the case where padding for bit-fields are unnamed that is regarded as deviating from this rule.
  Taking the addresses of members of a structure (or union) is regarded as deviating from this rule, because it is possible that they will be accessed without their name.

- **Functional specification**

  If a structure or union is defined with unnamed members, SQMlint reports it as a complaining. However, no report will be output for bit-fields that are zero bits wide, because they cannot have names according to ISO9899:1990.
  If the addresses of members of a structure (or union) are taken, SQMlint also reports them as a complaining.

7.114. Rule 114 (Not supported)

(required): Reserved words and standard library function names shall not be redefined or undefined.

- **Interpretation**

  As captioned.

- **Functional specification**

  This rule is not supported.

7.115. Rule 115

(required): Standard library function names shall not be reused.

- **Interpretation**

  Functions with the same names as the standard library function names cannot be defined.

- **Functional specification**

  If functions with the same names as the standard library function names are defined, SQMlint reports them as a complaining.

7.116. Rule 116 (Not supported)

(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.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.117. Rule 117 (Not supported)

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

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.118. Rule 118

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

- **Interpretation**
  As captioned.

- **Functional specification**
  If calls to functions *calloc()*, *malloc()*, *realloc()* or *free()* are used, SQMlint reports them as a complaining.

7.119. Rule 119

(required): The error indicator errno shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  If *errno* is used, SQMlint reports it as a complaining.

7.120. Rule 120 (Not supported)

(required): The macro offsetof, in library <stddef.h>, shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  This rule is not supported.

7.121. Rule 121

(required): <locale.h> and the setlocale function shall not be used.

- **Interpretation**
  As captioned.
  Nor can the *localeconv* function be used.

- **Functional specification**
  If calls to functions *setlocale()* or *localeconv()* are used, SQMlint reports them as a complaining.

7.122. Rule 122

(required): The setjmp macro and the longjmp function shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  If calls to functions *setjmp()* or *longjmp()* are used, SQMlint reports in complaining level.

7.123. Rule 123

(required): The signal handling facilities of <signal.h> shall not be used.

- **Interpretation**
  As captioned.

- **Functional specification**
  If calls to functions *signal()* or *raise()* are used, SQMlint reports in complaining level.

7.124. Rule 124

(required): The input/output library <stdio.h> shall not be used in production code.

- **Interpretation**

  In no case can *stdio.h* be included.

- **Functional specification**

  If *stdio.h* is included, SQMlint reports it as a complaining.

  However, the function *stdio.h* is used without including *stdio.h*, SQMlint outputs a report in rule 71.

7.125. Rule 125

(required): The library functions atof, atoi, and atol from library <stdlib.h> shall not be used.

- **Interpretation**

  As captioned.

- **Functional specification**

  If calls to functions *atof()*, *atoi()* or *atol()* are used, SQMlint reports them as a complaining.

- **Limitation**

  Inspection cannot be performed in cases where the functions *atof()*, *atoi()* or *atol()* are defined in *stdlib.h* as macro.

7.126. Rule 126

(required): The library functions about, exit, getenv, and system from library <stdlib.h> shall not be used.

- **Interpretation**

  As captioned.

- **Functional specification**

  If calls to functions *abort()*, *exit()*, *getenv()* or *system()* are used, SQMlint reports them as a complaining.

7.127. Rule 127

(required): The time handling functions of library <time.h> shall not be used.

- **Interpretation**

  As captioned.

- **Functional specification**

  If calls to functions *clock()*, *difftime()*, *mktime()*, *time()*, *asctime()*, *ctime()*, *gmtime()*, *localtime()* or *strftime()* are used, SQMlint reports them as a complaining.

8. Appendix Merge Utility

SQMmerger is a utility designed to create a file that can produce a mixed text of C source lines and corresponding report messages from the report files (CSV format) generated by SQMlint and the C source files.

8.1. Outline of Processing

8.1.1. Outline

SQMmerger creates a file that can produce a mixed text of C source lines and corresponding report messages from the C language source files and the report files generated by SQMlint. Figure 8.2.1.1 shows the relationship between the input and output files of SQMmerger.

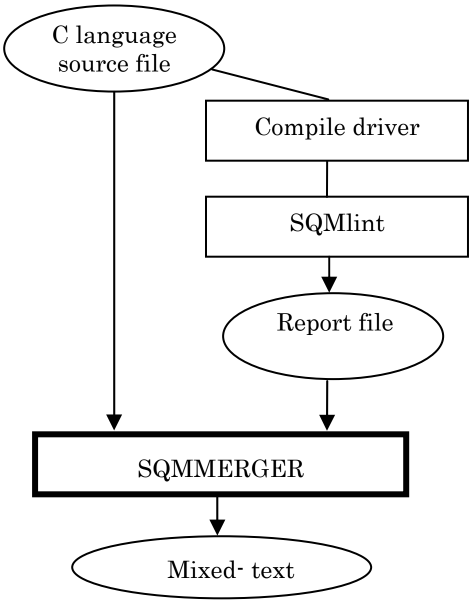


Figure 8.1.1.1. Input/output files of SQMviewer

Example of an output mixed text file:

```
1 : void  func(void);
2 : void  func(void)
3 : {
4 : LABEL:
[MISRA(55) Complain] label (LABEL) should not be used

5 :
6 :    goto LABEL;
[MISRA(56) Complain] the 'goto' statement shall not be used

7 : }
```

8.2. How to Use

8.2.1. Command Line

Command line usage:

SQMmerger -src *C-source-file-name* -r *report-file-name* -o *mixed-text-output-file-name* [-v] [-V] [-h]

Example: **sqmmerger -src test.c -r test.csv -o test.vi**

See Section 8.2.2, “Options of SQMmerger,” for details about the options of SQMmerger.

8.2.2. Options

The options of SQMmerger and their features are described below.

Table 8.2.2.1 Options

Option	Feature
-src *C-source-file-name*	Specifies the C source file name that was inspected by SQMlint.
-r *report-file-name*	Specifies the report file name generated by SQMlint.
-o *mixed-text-output-file-name*	Specifies the mixed text output file name to be generated by SQMmerger.
-v	Displays the command program name and the command line being executed.
-V	Terminates execution of SQMmerger after displaying a startup message.
-h	Displays command line options of SQMmerger.

8.3. Specification of Mixed Text Files Output

8.3.1. Output Format of C Source Lines

Line numbers and colons (:) are added to the beginning of C source lines when they are output.

Example:

```
1 : typedef signed int INT;
2 : INT glb;
```

8.3.2. Output Format of MISRA C Inspection Results

There are two formats in which the MISRA C inspection results are output.

(1) When the inspection result report message generated is for the source file name specified by the -src option

The MISRA C rule number and the content of a report message are output.

Example:

```
4 : LABEL:
[MISRA(55) Complain] label (LABEL) should not be used
```

(2) When the inspection result report message generated is for any source file name other

than the source file name specified by the -src option (i.e., a header file, etc. that has been included in the source file)
The MISRA C rule number, the source file name and line number in error, and the content of a report message are output.
Example:

```
[MISRA(124) Complain:stdio.h,line 10] the 'stdio.h' library shall not be
used in production code
```

If the inspection result report messages (errors) generated are for any source file other than the source file specified by the -src option, those report messages are output at the beginning of the mixed text file.
Example:

```
[MISRA(8) Complain:locale.h,line 15] multibyte characters shall not be used

1 : #include <locale.h>
2 :
3 : void comp1(void);
4 : void comp1(void)
5 : {
6 :        setlocale(0,0);
[MISRA(68) Complain] function (setlocale) shall always be declared at file scope
[MISRA(20) Complain] (setlocale) has not been declared yet
[MISRA(71) Complain] (setlocale) has not been declared as prototype
[MISRA(121) Complain] a (setlocale) function has been used

7 : }
```

Report messages generated for included files are output at the beginning of the file.

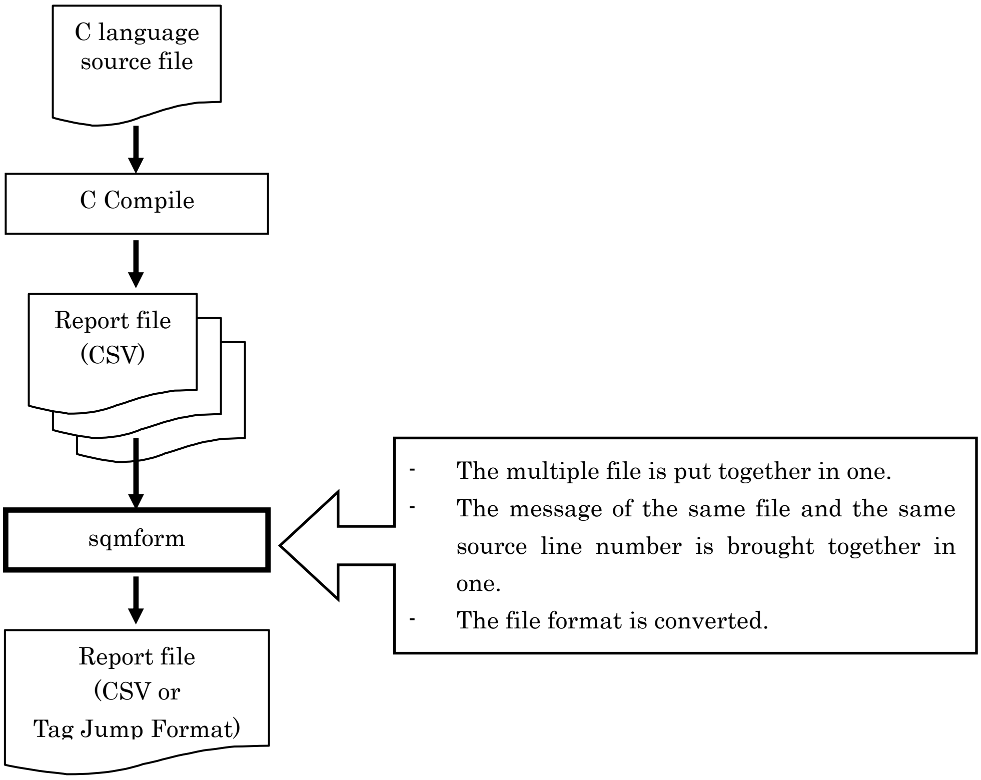
9. Appendix File format conversion Utility

SQMform is utility to convert the file that SQMlint generated into the file of the tag jump format. SQMform can bring two or more files together in one file. Moreover, because same information that exists in the multiple file is brought together in one, the output result of SQMlint can be compactly brought together.

9.1. Outline of Processing

9.1.1. Outline

SQMform connects CSV files that SQMlint generated. At this time, the overlapping message is brought together in one. Moreover, it is possible to convert it into the tag jump form for the text editor.



- The multiple file is put together in one.
- The message of the same file and the same source line number is brought together in one.
- The file format is converted.

Figure 9.1.1.1. Input/output files of sqmform

9.2. How to Use

9.2.1. Command Line

Command line usage:

```
sqmform [Options] CSV-file-name [CSV-file-name...]
```

Example: **sqmform -verbose -o summary.csv test1.csv test2.csv test3.csv**

See Section 9.2.2, “Options of sqmform,” for details about the options of sqmform.

9.2.2. Options

The options of sqmform and their features are described below.

Table 9.2.2.1 Options

Option	Feature
-o *output-file-name*	Specifies the output file name. - If this option is not specified, the output file name is a name that replaces the extension of the input file name with ".txt". - When the same file as the output file name already exists, an existing file is overwrited.
-form {tag1 | tag2 | tag3 | tag4 | csv}	The converted format is specified. 1. tag1 format file_name(line_number)message 2. tag2 format file_name line_number:message 3. tag3 format file_name line_number message 4. tag4 format file_name:line_number:message 5. csv format It is the same as output format of SQMlint. - If this option is not specified, "-form tag1" is made effective. - “tag1”, “tag2”, “tag3”, and “tag4” are the tag jump forms for the text editor. Please select the format suitable for the text editor that you use. Please use "csv" when you bring the output file of SQMlint together. - If you specify any of tag1, tag2, tag3, and tag4, path separator in the output file is always assumed to be "¥". The purpose of conversion is to correspond to the text editor that cannot use the tag jump function as path separator '/'.
-verbose	The following information is displayed in the standard output. - Beginning time - End time
-summary	The following information is displayed in the standard output.

	- MISRA C rule-number and output frequency. If the content of the message, the file name, and the line-number are all the same, it counts once.
-full_path	The absolute path is made by adding "Current directory" to "Relative path in the input file", and it is written in the output file. Please use this option when the tag jump function of the text editor needs the absolute path.
-V	Terminates execution of sqmform after displaying a startup message.

Add-in to Renesas' compilers
MISRA C Rule Checker SQMlint User's Manual

Publication Date: Aug. 1, 2006 Rev.1.00

Published by: Sales Strategic Planning Div.
Renesas Technology Corp.

Edited by: Microcomputer Tool Development Department
Renesas Solutions Corp.

MISRA C Rule Checker
SQMlint
User's Manual

RENESAS

Renesas Technology Corp.
2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan