The Motor Industry Software Reliability Association

MISRA-C:2004 Technical Corrigendum 1

Technical clarification of MISRA-C:2004

17 July 2007

First published July 2007
by MIRA Limited
Watling Street
Nuneaton
Warwickshire CV10 0TU
UK
www.misra-c.com

British Library Cataloguing in Publication Data.
A catalogue record for this book is available from the British Library

MISRA Mission Statement: To provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software.

MISRA, The Motor Industry Software Reliability Association, is a collaboration between vehicle manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety-related electronic systems in road vehicles and other embedded systems. To this end MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

www.misra.org.uk

Disclaimer

Adherence to the requirements of this document does not in itself ensure error-free robust software.

Compliance with the requirements of this document, or any other standard, does not of itself confer immunity from legal obligations.

Executive summary

Since the publication of MISRA-C:2004 [1], the MISRA C Working Group have developed the Exemplar Suite. During the development of the Exemplar Suite, and based on questions raised on the MISRA C Forum [2], a number of issues have been identified. This document provides clarification on these issues. The clarifications described by this document are incorporated into the Exemplar Suite release 1.0 dated 17 July 2007.

Acknowledgements

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

Paul Burden	Programming Research Ltd
Andrew Burnard	Land Rover Ltd
Mike Hennell	LDRA Ltd
Chris Hills	Phaedrus Systems Ltd
Gavin McCall	Visteon Engineering Services Ltd
Steve Montgomery	Ricardo UK Ltd
Chris Tapp	Keylevel Consultants Ltd
Liz Whiting	QinetiQ Ltd
David Ward	MIRA Limited

Contents

Contents (continued)

1. Overview

1.1 Introduction

This document is the MISRA-C:2004 Technical Corrigendum 1. This document has been developed based on feedback received since the release of MISRA-C:2004 [1] posted on the MISRA C Forum [2], and of issues identified during the preparation of the MISRA-C:2004 Exemplar Suite.

This document is set out to match the structure of the original document, and comments sequentially as required on each section of MISRA-C:2004. It is assumed that all readers of this document will have a copy of the original document, and this document describes changes from the original document.

2. MISRA-C:2004 Technical Corrigendum 1

The following sections describe changes to MISRA-C:2004. For each change a justification is provided.

2.1 Chapter 1

No changes.

2.2 Chapter 2

No changes.

2.3 Chapter 3

No changes.

2.4 Chapter 4

4.2.3 Tool selection and validation

Add text to end of this section:

> Where additional static analysis checks are available, the use of these additional checks is recommended.

2.5 Chapter 5

No changes

2.6 Chapter 6

In this section, the format of the rules is described as follows.

> Rule x.y **This is the headline rule text**
> [This is reference text]
>
> This is normative text.

Each change below will reference which part of the Rule text is changed.

All Rules

The “supporting text” in MISRA-C:2004 is now replaced with “normative text”, which is identical to the previous text other than as detailed below. In order to

conform with MISRA-C:2004, it is now considered necessary to meet the requirements of this normative text.

Rule 3.2

Add to end of normative text:

> The source code is written in one, or more character sets. Optionally, the program can execute in a second or multiple character sets. All the source and execution character sets shall be documented.

Justification

Documenting the character sets used in either the source code or the application increases project awareness and prevents any issues from using incompatible character sets.

Rule 4.1

Replace normative text with:

> Only "simple-escape-sequences" in ISO 9899:1990 [3–6] section 6.1.3.4 and `\0` are permitted escape sequences.
>
> All "hexadecimal-escape-sequences" are prohibited.
>
> The "octal-escape-sequences" other than `\0` are also prohibited under Rule 7.1.

Justification

Only `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\'` (single quote), `\"` (double quote), `\?`, `\\` and `\0` are well-defined.

All other "octal-escape-sequences" and "hexadecimal-escape-sequences" exhibit implementation-defined behaviour.

Readability of "octal-escape-sequences" and "hexadecimal-escape-sequences" can be confusing, (see [7] Harbison and Steele, *C: A Reference Manual*, 4th Edition, on Numeric Escape Codes.)

Rule 5.1

Add to normative text at end of first paragraph:

> This rule shall apply across all name spaces.
>
> Macro names are also included and the 31 character limit applies before and after substitution.

Justification

Clarification of scope.

Rule 5.3

Insert before the last sentence in normative text:

> The same *typedef* shall not be duplicated anywhere in the source code files even if the declarations are identical.

Correct typing error in normative text — replace:

> Where the type definition is made in a header file, and that header file is included in multiple source files, this rule is not a violated.

with:

> Where the type definition is made in a header file, and that header file is included in multiple source files, this rule is not violated.

Justification

Clarification of headline rule and typographical error introduced during printing.

Rule 5.4

Insert before the last sentence in normative text:

> The same tag definition shall not be duplicated anywhere in the source code files even if the definitions are identical.

Replace text:

> Where the type definition is made in a header file, and that header file is included in multiple source files, this rule is not violated.

with:

> Where the tag definition is made in a header file, and that header file is included in multiple source files, this rule is not violated.

Justification

Clarification of headline rule and typographical error introduced during printing.

Rule 5.6

Insert the word “member” after “structure” in the headline rule.

Justification

Clarification of headline rule and removal of ambiguity. The exception applies to both “structure members” and “union members”.

Rules 6.1 and 6.2 (a)

Add normative text after first paragraph:

> Character values/data are character constants or string literals such as `'A'`, `'5'`, `'\n'`, `"a"`.
>
> Numeric values/data are numbers such as `0`, `5`, `23`, `\x10`, `-3`.
>
> Character sets map text characters onto numeric values. Character values are the “text”.

Justification

Provide explicit examples of these value types.

Rules 6.1 and 6.2 (b)

Replace last sentence of normative text:

> The only permissible operators on plain char types are assignment and equality operators (=, ==, !=).

with:

> The permissible operators on plain char types are the simple assignment operator (=), equality operators (==, !=) and explicit casts to integral types. Additionally, the second and third operands of the ternary conditional operator may both be of plain char type.

Justification

The ternary operator had previously been omitted inadvertently. Explicit casting from integral type to plain char type has always been permitted. Explicit casting from plain char is now also permitted.

Rule 6.3

Insert “numerical” into headline rule as follows:

> *typedefs* that indicate size and signedness should be used in place of the basic numerical types.

Replace the first sentence of normative text:

> The basic numerical types of *char*, *int*, *short*, *long*, *float* and *double* should not be used, but …

with:

> The basic numerical types of *signed* and *unsigned* variants of *char*, *int*, *short*, *long*, and *float*, *double* should not be used, but …

Justification

This rule is inappropriate for (plain) *char*, "Boolean" and *enum* types.

Rule 6.5

Change headline rule:

> Bit fields of type *signed int* shall be at least 2 bits long

to:

> Bit fields of *signed* type shall be at least 2 bits long

Justification

This rule applies to any signed integral type when Rule 6.4 is deviated.

Rule 7.1

Replace the following normative text:

> The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule.

with:

> The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule. Additionally "`\0`" is the only permitted octal escape sequence.

Justification

Clarification that exception to allow zero also extends to "`\0`".

Rule 8.5

Add the following to the end of normative text.

> A "header file" is defined as any file that is included via the *#include* directive, regardless of name or suffix.

Justification

Clarification of term "header file".

Rule 8.7

Add to end of normative text:

> "Accessing" means using the identifier to read from, write to, or take the address of the object.

Justification

Clarification of the term "accessing".

Rule 8.12

Remove reference to "Undefined 46".

Justification

Incorrect reference.

Rule 9.2

Add normative text:

> The intent of Rule 9.2 is that the non-zero initialisation of arrays and structures shall require an explicit initialiser for each element, e.g.

```
int16_t arraya1[5] = { 1, 2, 3, 0, 0 };
            /* Compliant - non-zero initialisation */

int16_t arraya2[5] = { 0 };
            /* Compliant- zero initialisation*/

int16_t arraya3[5] = { 1, 2, 3 };
            /* Not Compliant - non-zero initialisation */

int16_t arraya4[2][2] = { 0 };
            /*  Compliant  –  zero  initialisation  at  top-
      level */

int16_t arraya5[2][2] = { { 0 }, { 1, 2 }};
```

```
/* Not Compliant – zero initialisation at sub-level */
```

Zero or *NULL* initialisation shall only be applied at the top level of the array or structure.

Justification

Previously the rule did not address incomplete non-zero initialisation of arrays and structures.

Section 6.10.4

Add after last paragraph in Section 6.10.4:

Note: while in the ISO C language definition character constants have a type of *int*, MISRA C considers the underlying type of a character constant to be plain *char*.

Justification

Clarification.

Rule 10.3

Replace headline rule:

The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

with:

The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression

Justification

(a) Rule has been replaced to allow same size casting to avoid contradiction with Rule 10.5, and
(b) “may” replaced with “shall” since this is a required rule.

Rule 10.4

Replace headline rule:

The value of a complex expression of floating type may only be cast to a narrower floating type.

with:

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

Justification

(a) Rule has been replaced to allow same size casting to avoid contradiction with Rule 10.5, and
(b) “may” replaced with “shall” since this is a required rule.

Rule 10.5

Add normative text:

No cast is required if the result of the bitwise operation is:

(a) immediately assigned to an object of the same underlying type as the operand;
(b) used as a function argument of the same underlying type as the operand;
(c) used as a return expression of a function whose return type is of the same underlying type as the operand.

Rule 11.1

Replace normative text:

This means, for example, that a pointer to a function cannot be converted to a pointer to a different type of function.

with:

This means that a function pointer can be converted to or from an integral type. No other conversions involving function pointers are permitted.

Justification

Clarification.

Rule 11.2

Add additional normative text:

This means that an object pointer can be converted to or from:

(a) An integral type;
(b) Another pointer to object type;
(c) A pointer to void.

No other conversions involving object pointers are permitted.

Justification

Clarification.

Rule 12.1

Replace the example text:

```
uint16_t a = 10;
uint16_t b = 65535;
uint32_t c = 0;
```

with:

```
uint16_t a = 10U;
uint16_t b = 65535U;
uint32_t c = 0U;
```

Justification

These are unsigned constants.

Rule 12.3

Add to end of normative text:

> `sizeof()` shall only be applied to an operand which is a type or an object. This may include volatile objects.

Justification

Minor clarification.

Rule 12.6

Replace headline rule text:

> Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).

with:

> Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||`, `!`, `=`, `==`, `!=` and `?:`).

Justification

Omission.

Rule 12.7

Replace normative text:

> Bitwise operations (~, <<, >>, &, ^ and |) are not normally meaningful on signed integers.

with:

> Bitwise operations (~, <<, <<=, >>, >>=, &, &=, ^, ^=, | and |=) are not normally meaningful on signed integers.

Justification

Include compound assignment operations in excluded list.

Rule 12.13

Add following to start of normative text:

> It is the intention of the rule that when the increment or decrement operator is used, it should be the only side-effect in the statement.

Justification

The existing formulation of the rule incorrectly excluded array indexing, for example `a[i]++;` and `(*p)++;` are allowed.

Rule 13.5

Revise normative text as follows:

> When present, the three expressions of a *for* statement shall be used only for these purposes:

with:

> The three expressions of a *for* statement shall be used only for these purposes:

Add following normative text at end:

> The following options are allowed:
>
> (a) all three expressions shall be present;
>
> (b) the second and third expressions shall be present with prior initialisation of the loop counter;
>
> (c) all three expressions shall be empty for a deliberate infinite loop.

Justification

Clarification of allowed forms.

Rule 14.1

Add following normative text at end:

> Code which is excluded by pre-processor directives is not present following pre-processing, and is therefore not subject to this rule.

Justification

Clarification of application of rule.

Rule 14.10

Replace normative text:

> The *else* statement should either take appropriate action or contain a suitable comment as to why no action is taken.

with:

> The *else* statement shall either take appropriate action or contain a suitable comment as to why no action is taken.

Justification

Replace “should” with “shall” in normative text.

Rule 15.0

Insert normative text before Rule 15.1:

> The preamble normative text in section 15 shall be treated as Rule 15.0.
>
> Any deviation from the normative text shall be considered a non-compliance if no other rule in section 15 is not complied with.

Rule 15.3

Replace normative text:

> This clause should either take appropriate action or contain a suitable comment as to why no action is taken.

with:

This clause shall either take appropriate action or contain a suitable comment as to why no action is taken.

Justification

Replace “should” with “shall” in normative text.

Rule 16.5

Replace headline rule:

Functions with no parameters shall be declared with parameter type *void*.

with:

Functions with no parameters shall be declared and defined with the parameter list *void*.

Justification

Rule applies to both declaration and definition.

Rule 17.4

Replace example text with the following:

```
void my_fn(uint8_t * p1, uint8_t p2[])
{
   uint8_t index = 0U;
   uint8_t * p3;
   uint8_t * p4;

   *p1 = 0U;
   p1 ++;       /* not compliant – pointer increment               */
   p1 = p1 + 5; /* not compliant – pointer increment               */
   p1[5] = 0U;  /* not compliant – p1 was not declared as an array */
   p3 = &p1[5]; /* not compliant – p1 was not declared as an array */
   p2[0] = 0U;
   index ++;
   index = index + 5U;
   p2[index] = 0U; /* compliant     */
   p4 = &p2[5];   /* compliant     */
}
uint8_t a1[16];
uint8_t a2[16];

my_fn(a1, a2);

my_fn(&a1[4], &a2[4]);

uint8_t a[10];
uint8_t * p;
p = a;
```

```
*(p+5) = 0U; /* not compliant */
p[5] = 0U;   /* not compliant     */
```

Justification

- In all cases, array indexing shall only be permitted on objects explicitly declared to be arrays. `p[5]` is not compliant as `p` is not explicitly declared as an array .
- Suffix `U` omitted in error.

Rule 18.1

Replace normative text:

> A complete declaration of the structure or union shall be included within any translation unit that refers to that structure.

with:

> A complete declaration of the structure or union shall be included within any translation unit that reads from or writes to that structure. A pointer to an incomplete type is itself complete and is permitted, and therefore the use of opaque pointers is permitted.

Justification

Clarification — permit the use of opaque pointers.

Rule 19.2

Add to end of normative text:

> Use of the `\` character is permitted in filename paths without the need for a deviation if required by the host operating system of the development environment.

Justification

This rule could not previously be adhered to in a widely used development environment.

Rule 19.3

Replace example text:

```
#define FILE "filename.h"
#include FILE
```

with:

```
#define FILE_A "filename.h"
#include FILE_A
```

Justification

FILE is a reserved *typedef* name in the standard. The original use of *FILE* violated Rule 20.1 and has been changed.

Rule 19.4

Replace headline rule text (added string literal):

> C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

with:

> C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

Add the following example to the "compliant" section:

```
#define CAT (PI)                /* parenthesised expression */
#define FILE_A "filename.h"     /* string literal */
```

Add the following example to the "not compliant" section:

```
#define CAT PI                  /* non-parenthesised expression */
```

Justification

Clarification and completeness.

This rule attempts to limit the potential range of macro definitions to a sensible subset.

Rule 19.12

Correct the headline text:

> There shall be at most one occurrence of the # or # preprocessor operators in a single macro definition.

to:

> There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

Justification

Correction of typographical error introduced during printing.

Rule 19.13

Correct the headline text:

> The # and # preprocessor operators should not be used.

to:

> The # and ## preprocessor operators should not be used.

Justification

Correction of typographical error introduced during printing.

Rule 20.1

Change second sentence in first paragraph of normative text:

> It is also bad practice to *#define* a macro name which is a C reserved identifier or C keyword which is the name of any macro, object or function in the standard library.

to:

> It is also bad practice to *#define* a macro name which is a C reserved identifier, a C keyword or the name of any macro, object or function in the standard library.

Replace last paragraph in normative text:

> ISO C reserved identifiers are documented in sections 7.1.3 and 7.13 of ISO 9899:1990 [2] and should also be documented by the compiler writer. Generally, all identifiers that begin with the underscore character are reserved.

with:

> Reserved identifiers are defined by ISO/IEC 9899:1990 Sections 7.1.3 “Reserved identifiers” and 6.8.8 “Predefined macro names”. Macros in the standard library are examples of reserved identifiers. Functions in the standard library are examples of reserved identifiers. Any identifier in the standard library is considered a reserved identifier in any context i.e. at any scope or regardless of header files.
>
> The defining, redefining or undefining of the reserved identifiers defined in 7.13 “Future library directions” is advisory.
>
> Rule 20.1 applies regardless of which, if any, header files are included.

Justification

Clarification of scope of rule. The relaxation of the future directions is due to the broad impact of enforcement, i.e. `total` or `is_safe` would have been prohibited.

Rule 20.3 (a)

Replace normative text in the final paragraph on page 82:

> The programmer should identify any domain constraints which should sensibly apply to a function being used (which may or may not be documented in the standard), and provide appropriate checks that the input value(s) lies within this domain. Of course the value may be restricted further, if required, by knowledge of what the parameter represents and what constitutes a sensible range of values for the

with:

> The programmer should identify any domain constraints which should sensibly apply to a function being used (which may or may not be documented in the standard), and provide appropriate checks that the input value(s) lies within this domain. Of course the value may be restricted further, if required, by knowledge of what the parameter represents and what constitutes a sensible range of values for the parameter.

Justification

Correction of typographical error introduced during printing.

Rule 20.3 (b)

Replace normative text in the final paragraph:

> Note that when checking a floating-point parameter to a function, that has a singularity at zero, it may be appropriate to perform a test of equality to zero. This is an acceptable exception to Rule 13.3 without the need to raise a deviation.

with:

> Note that when checking a floating-point parameter to a function, that has a singularity at zero, it may be appropriate to perform a test of equality to zero. This will require a deviation to Rule 13.3.

Justification

A deviation is required in all cases.

2.7 Appendix A

Replace:

14.8 (req) The statement forming the body of a *switch*, *while*, *do ... while* or *for* statement be a compound statement.

with:

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

Justification

Correction of typographical error introduced during printing.

2.8 Appendix D

Move cross-references for Rules 14.5–14.9 (column 1 of table), Rules 17.2–17.6 (column 2) and Rules 19.12–19.16 (column 3) from the bottom of the table on page 107 to the start of the respective columns in the continuation table on the following page.

Justification

Correction of typographical error introduced during printing.

3. References

[1] *MISRA-C:2004 Guidelines for the use of the C language in critical systems*, ISBN 0-9524156-4-X, MIRA, October 2004.

[2] MISRA Web Forum at http://www.misra-c.com/forum

[3] ISO/IEC 9899:1990, *Programming languages – C*, ISO, 1990.

[4] ISO/IEC 9899:COR1:1995, Technical Corrigendum 1, 1995.

[5] ISO/IEC 9899:AMD1:1995, Amendment 1, 1995.

[6] ISO/IEC 9899:COR2:1996, Technical Corrigendum 2, 1996.

[7] Harbison S.P. and Steele G.L, *C: A Reference Manual*, 4th Edition, ISBN 0-13-326232-4, Prentice Hall, 1994.