The Motor Industry Software Reliability Association

Guidelines For The Use Of The C Language In Vehicle Based Software

April 1998

Executive summary

This document specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems up to and including safety integrity level 3 (as defined in the MISRA Guidelines). It contains a list of rules concerning the use of the C programming language together with justifications and examples.

In addition to these rules the document also briefly describes the reason why such a language subset is required and gives guidance on how to use it.

It is recognised that these language issues are only a small part of the overall task of developing software, and guidance is given on what else needs to be addressed by the developer if they are to have a credible claim of ‘best practice’ development.

This document is not intended to be an introduction or training aid to the subjects it embraces. It is assumed that readers of this document are familiar with the ISO C programming language standard and associated tools, and also have access to the primary reference documents. It also assumes that users have received the appropriate training and are competent C language programmers.

Acknowledgements

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

Paul Edwards	Rover Group Ltd
Simon Fisher	AB Automotive Electronics Ltd
Gavin McCall	Visteon Automotive Systems, an enterprise of Ford Motor Company Ltd
David Newman	Visteon Automotive Systems, an enterprise of Ford Motor Company Ltd
Frank O'Neill	Lucas Automotive Electronics
Richard Pearman	Lotus Engineering
Roger Rivett	Rover Group Ltd

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the review process:

David Blyth	D. C. Hurst	Steve Montgomery
Dave Bowen	Peter Jesty	Tim Mortimer
Mark Bradbury	Derek Jones	Edward Nelson
Kwok Chan	Ian Kendall	Frank Plagge
Shane Cook	Andreas Krüger	Agneta Sjögren
Frank Cooper	Lawrence Lane	Kevin Talbot
Dewi Daniels	Chris Leiby	Chris Tapp
Gavin Finnie	Thomas Maier	Lloyd Thomas
John Fox	Paul Martin	Ken Tindell
Kenshiro Hashimoto	Neil Martin	Ron Wilson
Richard Hathway	M. A. C. Mettes	Chris Winters
Steven Hughes	Charlie Monk	

The reviewers represented a broad cross-section of automotive developers and component suppliers originating from eight countries (Austria, Germany, Japan, Holland, Norway, Sweden, United Kingdom and United States of America) and the following types of organisations:

- Vehicle manufacturers (8)
- Component and system suppliers (7)
- Software specialists and consultancies (6)
- Compiler and static analyser suppliers (3)
- Research and development organisations (3)
- Universities (4)

Contents

Contents (continued)

1. Background – the use of C and issues with it

1.1 The use of C in the automotive industry

The C programming language [1] is growing in importance and use for real-time embedded applications within the automotive industry. This is due largely to the inherent language flexibility, the extent of support and its potential for portability across a wide range of hardware. Specific reasons for its use include:

- For many of the microprocessors in use, if there is any other language available besides assembly language then it is usually C. In many cases other languages are simply not available for the hardware.
- C gives good support for the high-speed, low-level, input/output operations, which are essential to many automotive embedded systems.
- Increased complexity of applications makes the use of a high-level language more appropriate than assembly language.
- C can generate smaller and less RAM-intensive code than many other high-level languages.
- A growth in portability requirements caused by competitive pressures to reduce hardware costs by porting software to new, and/or lower cost, processors at any stage in a project lifecycle.
- A growth in the use of auto-generated C code from modelling packages.
- Increasing interest in open systems and hosted environments.

1.2 Language insecurities and the C language

No programming language can guarantee that the final executable code will behave exactly as the programmer intended. There are a number of problems that can arise with any language, and these are broadly categorised below. Examples are given to illustrate insecurities in the C language.

1.2.1 The programmer makes mistakes

Programmers make errors, which can be as simple as mis-typing a variable name, or might involve something more complicated like misunderstanding an algorithm. The programming language has a bearing on this type of error. Firstly the style and expressiveness of the language can assist or hinder the programmer in thinking clearly about the algorithm. Secondly the language can make it easy or hard for typing mistakes to turn one valid construct into another valid (but unintended) construct. Thirdly the language may or may not detect errors when they are made.

Firstly, in terms of style and expressiveness C can be used to write well laid out, structured and expressive code. It can also be used to write perverse and extremely hard-to-understand code. Clearly the latter is not acceptable in a safety-related system.

Secondly the syntax of C is such that it is relatively easy to make typing mistakes that lead to perfectly valid code. For example, it is all too easy to type '=' (assignment) instead of '==' (logical comparison) and the result is nearly always valid (but wrong), while an extra semi-colon on the end of an *if* statement can completely change the logic of the code.

Thirdly the philosophy of C is to assume that the programmers know what they are doing, which can mean that if errors are made they are allowed to pass unnoticed by the language. An area in which C is particularly weak in this respect is that of 'type checking'. C will not object, for example, if the programmer tries to store a floating-point number in an integer that they are using to represent a true/false value. Most such mismatches are simply forced to become compatible. If C is presented with a square peg and a round hole it doesn't complain, but makes them fit!

1.2.2 The programmer misunderstands the language

Programmers can misunderstand the effect of constructs in a language. Some languages are more open to such misunderstandings than others.

There are quite a number of areas of the C language that are easily misunderstood by programmers. An example is the set of rules for operator precedence. These rules are well defined, but very complicated, and it is easy to make the wrong assumptions about the precedence that the operators will take in a particular expression.

1.2.3 The compiler doesn't do what the programmer expects

If a language has features that are not completely defined, or are ambiguous, then a programmer can assume one thing about the meaning of a construct, while the compiler can interpret it quite differently.

There are many areas of the C language which are not completely defined, and so behaviour may vary from one compiler to another. In some cases the behaviour can vary even within a single compiler, depending on the context. Altogether the C standard, in Annex G, lists 201 issues that may vary in this way. This can present a sizeable problem with the language, particularly when it comes to portability between compilers. However, in its favour, the C standard does at least list the issues, so they are known.

1.2.4 The compiler contains errors

A language compiler (and associated linker etc.) is itself a software tool. Compilers may not always compile code correctly. They may, for example, not comply with the language standard in certain situations, or they may simply contain 'bugs'.

Because there are aspects of the C language that are hard to understand, compiler writers have been known to misinterpret the standard and implement it incorrectly. Some areas of the language are more prone to this than others. In addition, compiler writers sometimes consciously choose to vary from the standard.

1.2.5 Run-time errors

A somewhat different language issue arises with code that has compiled correctly, but for reasons of the particular data supplied to it causes errors in the running of the code. Languages can build run-time checks into the executable code to detect many such errors and take appropriate action.

C is generally poor in providing run-time checking. This is one of the reasons why the code generated by C tends to be small and efficient, but there is a price to pay in terms of detecting errors during execution. C compilers generally do not provide run-time checking for such common problems as arithmetic exceptions (e.g. divide by zero), overflow, validity of addresses for pointers, or array bound errors.

1.3 The use of C for safety-related systems

It should be clear from section 1.2 that great care needs to be exercised when using C within safety-related systems. Because of the kinds of issues identified above, various concerns have been expressed about the use of C on safety-related systems. Certainly it is clear that the full C language should not be used for programming safety-related systems.

However in its favour as a language is the fact that C is very mature, and consequently well analysed and tried in practice. Therefore its deficiencies are known and understood. Also there is a large amount of tool support available commercially which can be used to statically check the C source code and warn the developer of the presence of many of the problematic aspects of the language.

If, for practical reasons, it is necessary to use C on a safety-related system then the use of the language must be constrained to avoid, as far as is practicable, those aspects of the language which do give rise to concerns. This document provides one such set of constraints (often referred to as a ‘language subset’).

Hatton [2] considers that, providing “... severe and automatically enforceable constraints ...” are imposed, C can be used to write “... software of *at least* as high intrinsic quality and consistency as with other commonly used languages”.

Nonetheless, it should be recognised that there are other languages available which are in general better suited to safety-related systems, having (for example) fewer insecurities and better type checking. Examples of languages generally recognised to be more suitable than C are Ada and Modula 2. If such languages could be available for a proposed system then their use should be seriously considered in preference to C.

Note also that assembly language is no more suitable for safety-related systems than C, and in some respects is worse. Use of assembly language in safety-related systems is not recommended, and generally if it is to be used then it needs to be subject to stringent constraints.

1.4 C standardization

The current full standard for the C programming language is ISO/IEC 9899:1990 [1] along with technical corrigendum 1 (1995), and this is the standard which has been adopted by this document. The standard is also published by BSI in the UK as BS EN 29899:1993 and Amendment 1.

The same standard was originally published by ANSI as X3.159-1989 [3]. In content the ISO/IEC standard and the ANSI standard are identical, and equally acceptable for use with this document. Note, however, that the section numbering is different in the two standards, and this document follows the section numbering of the ISO standard.

Also note that the ANSI standard [3] contains a useful appendix giving the rationale behind some of the decisions made by the standardization committee. This appendix does not appear in the ISO edition.

2. MISRA C: The vision

2.1 Rationale for the production of MISRA C

The MISRA consortium published its Development Guidelines for Vehicle Based Software [4] in 1994. This document describes the full set of measures that should be used in software development. In particular, the choices of language, compiler and language features to be used, in relationship with integrity level, are recognised to be of major importance. Section 3.2.4.3 (b) and Table 3 of the MISRA Guidelines [4] address this. One of the measures recommended is the use of a subset of a standardized language, which is already established practice in the aerospace, nuclear and defence industries. This document addresses the definition of a suitable subset of C.

2.2 Objectives of MISRA C

In publishing this document regarding the use of the C programming language, the MISRA consortium is not intending to promote the use of C in the automotive industry. Rather it recognises the already widespread use of C, and this document seeks only to promote the safest possible use of the language.

It is the hope of the MISRA consortium that this document will gain industry acceptance and that the adoption of a safer subset will become established as best practice both by vehicle manufacturers and the many component suppliers. It should also encourage training and enhance competence in general C programming, and in this specific subset, at both an individual level and a company level.

Great emphasis is placed on the use of static checking tools to enforce compliance with the subset and it is hoped that this too will become common practice by the developers of automotive embedded systems.

Although much has been written by academics concerning languages and their pros and cons this information is not well known among automotive developers. Another goal of this document is that engineers and managers within the automotive industry will become much more aware of the language-choice issues.

The availability of many tools to assist in the development of software, particularly tools to support the use of C, is a benefit. However there is always a concern over the robustness of their design and implementation, particularly when used for the development of safety-related software. It is hoped that the active approach of the automotive industry to establish software best practice (through the MISRA Guidelines [4] and this document) will encourage the commercial off-the-shelf (COTS) tool suppliers to be equally active in ensuring their products are suitable for application in the automotive industry.

7.9 Operators

See also:	Rules 47, 101, 103

Rule 33 (required): The right hand operand of a `&&` or `||` operator shall not contain side effects.

There are some situations in C code where certain parts of expressions may not be evaluated. If these sub-expressions contain side effects then those side effects may or may not occur, depending on the values of other sub expressions.

The operators which can lead to this problem are `&&`, `||` and `?:`. In the case of the first two (logical operators) the evaluation of the right-hand operand is conditional on the value of the left-hand operand. In the case of the `?:` operator, either the second or third operands are evaluated but not both. The conditional evaluation of the right hand operand of one of the logical operators can easily cause problems if the programmer relies on a side effect occurring. The `?:` operator is specifically provided to choose between two sub-expressions, and is therefore less likely to lead to mistakes.

For example:

```
if ( ishigh && ( x == i++ ) )   /* Incorrect */
if ( ishigh && ( x == f(x) ) )  /* Only acceptable if f(x) is
                                   known to have no side effects */
```

Rule 34 (required): The operands of a logical `&&` or `||` shall be primary expressions.

‘Primary expressions’ are defined in ISO 9899 [1], section 6.3.1. Essentially they are either a single identifier, or a constant, or a parenthesised expression. The effect of this rule is to require that if an operand is other than a single identifier or constant then it must be parenthesised. Parentheses are important in this situation both for readability of code and for ensuring that the behaviour is as the programmer intended.

For example write:

```
if ( ( x == 0 ) && ishigh )  /* x == 0 must have parentheses */
                             /* ishigh need not */
```

Rule 35 (required): Assignment operators shall not be used in expressions which return Boolean values.

[Koenig 6]

Strictly speaking, in C, there is no Boolean type, but there is a conceptual difference between expressions which return a numeric value and expressions which return a Boolean value.

If assignments are required then they must be performed separately outside of any expressions which are effectively of Boolean type. For example write:

Appendix B

Appendix B: Cross references to the ISO standard

This appendix gives cross references between the rules given in this document and the sections of ISO 9899 [1].

B1. Rule numbers to ISO 9899 references

Rule	ISO Ref	Rule	ISO Ref	Rule	ISO Ref
5	5.2.1	**30**	6.5.7	**53**	5.1.2.3
7	5.2.1.1	**31**	6.5.7	**54**	6.6.3
8	5.2.1.2, 6.1.4	**32**	6.5.2.2	**55**	6.6.1
9	6.1.9	**33**	6.3.13, 6.3.14	**56**	6.6.6.1
10	6.1.9	**34**	6.3.1, 6.3.13, 6.3.14	**57**	6.6.6.2
11	6.1.2	**35**	6.3.16	**58**	6.6.6.3
12	6.1.2.3	**36**	6.3.3.3, 6.3.10, 6.3.12, 6.3.13, 6.3.14	**59**	6.6.4.1, 6.6.5.1, 6.6.5.2, 6.6.5.3
13	6.1.2.5, 6.5.2, 6.5.6	**37**	6.3.3.3, 6.3.7, 6.3.10, 6.3.11, 6.3.12	**60**	6.6.4.1
14	6.1.2.5, 6.2.1.1, 6.5.2	**38**	6.3.7	**61**	6.6.4.2
15	6.1.2.5	**39**	6.3.3.3	**62**	6.6.4.2
16	6.1.2.5, 7.5	**40**	6.3.3.4	**63**	6.6.4.2
17	6.5.6	**41**	6.3.5	**64**	6.6.4.2
18	6.1.3	**42**	6.2, 6.3.17	**65**	6.6.5
19	6.1.3.2	**43**	6.3.4	**66**	6.6.5.3
20	6.1.2.1, 6.5	**44**	6.3.4	**67**	6.6.5.3
21	6.1.2.1	**45**	6.3.4	**68**	6.5.4.3
22	6.1.2.1, 6.5	**46**	5.1.2.3, 6.3	**69**	6.5.4.3, 7.8
23	6.1.2.1, 6.5.1	**47**	6.3	**70**	6.3.2.2
24	6.1.2.2	**48**	6.2, 6.3, 6.3.4	**71**	6.5.4.3
25	6.7	**50**	6.3.9	**72**	6.5.4.3
26	6.1.2.6, 6.5	**51**	6.4	**73**	6.5.4.3
27	6.5, 6.7			**74**	6.5.4.3
28	6.5.1			**75**	6.5.4.3
29	6.5.2.3			**76**	6.5.4.3
				77	6.3.2.2

Rule	ISO Ref	Rule	ISO Ref	Rule	ISO Ref
78	6.3.2.2	**95**	6.8.3	**112**	6.5.2.1
79	6.2.2.2, 6.3.2.2	**96**	6.8.3	**113**	6.5.2.1
80	6.2.2.2, 6.3.2.2	**97**	6.8	**114**	7.1.3
81	6.5.4.3	**98**	6.8.3.2, 6.8.3.3	**115**	7.1.3
82	6.6.6.4	**99**	6.8.6	**117**	7.1.7
83	6.6.6.4	**100**	6.8.1	**118**	7.10.3
84	6.6.6.4	**101**	6.3.6, 6.3.16.2	**119**	7.1.4
85	6.3.2.2	**102**	6.3.3.2, 6.5.4.1	**120**	7.1.6
86	6.3.2.2	**103**	6.3.8	**121**	7.4
87	6.8.2	**104**	6.5.4.1, 6.5.4.3	**122**	7.6
88	6.1.7	**105**	6.5.4.1, 6.5.4.3	**123**	7.7
89	6.8.2	**106**	6.3.3.2	**124**	7.9
90	6.8.3	**107**	6.2.2.3	**125**	7.10.1
91	6.8.3	**108**	6.1.2.5, 6.5.2.1	**126**	7.10.4
92	6.8.3.5	**109**	6.1.2.5, 6.5.2.1	**127**	7.12
93	6.8.3	**110**	6.1.2.5, 6.5.2.1		
94	6.8.3	**111**	6.5.2.1		

B2. ISO 9899 references to rule numbers

ISO Ref	Rule
5.1.2.3	46, 53
5.2.1	5
5.2.1.1	7
5.2.1.2	8
6.1.2	11
6.1.2.1	20, 21, 22, 23
6.1.2.2	24
6.1.2.3	12
6.1.2.5	13, 14, 15, 16, 108, 109, 110
6.1.2.6	26
6.1.3	18
6.1.3.2	19
6.1.4	8
6.1.7	88
6.1.9	9, 10
6.2	43, 48
6.2.1.1	14
6.2.2.2	79, 80
6.2.2.3	107
6.3	46, 47, 48
6.3.1	34
6.3.2.2	70, 77, 78, 79, 80, 85, 86
6.3.3.2	102, 106
6.3.3.3	36, 37, 39
6.3.3.4	40
6.3.4	43, 44, 45, 48
6.3.5	41
6.3.6	101
6.3.7	37, 38
6.3.8	103
6.3.9	50
6.3.10	36, 37
6.3.11	37
6.3.12	36, 37
6.3.13	33, 34, 36
6.3.14	33, 34, 36
6.3.16	35
6.3.16.2	101
6.3.17	42
6.4	51
6.5	20, 22, 26, 27
6.5.1	23, 28
6.5.2	13, 14
6.5.2.1	108, 109, 110, 111, 112, 113
6.5.2.2	32
6.5.2.3	29
6.5.4.1	102, 104, 105
6.5.4.3	68, 69, 71, 72, 73, 74, 75, 76, 81, 104, 105
6.5.6	13, 17
6.5.7	30, 31
6.6.1	55
6.6.3	54
6.6.4.1	59, 60
6.6.4.2	61, 62, 63, 64
6.6.5	65
6.6.5.1	59
6.6.5.2	59
6.6.5.3	59, 66, 67
6.6.6.1	56
6.6.6.2	57
6.6.6.3	58
6.6.6.4	82, 83, 84
6.7	25, 27
6.8	97
6.8.1	100
6.8.2	87, 89
6.8.3	90, 91, 93, 94, 95, 96
6.8.3.2	98
6.8.3.3	98
6.8.3.5	92
6.8.6	99
7.1.3	114, 115
7.1.4	119
7.1.6	120
7.1.7	117
7.4	121
7.5	16
7.6	122
7.7	123
7.8	69
7.9	124
7.10.1	125
7.10.3	118
7.10.4	126
7.12	127