# MISRA-C:2004

# Guidelines for the use of the C language in critical systems

October 2004

First published October 2004

by MIRA Limited
Watling Street
Nuneaton
Warwickshire CV10 0TU
UK

www.misra-c.com

Edition 2 reprinted July 2008 incorporating Technical Corrigendum 1

ISBN 978-0-9524156-2-6 paperback
ISBN 978-0-9524156-4-0 PDF

Printed by Hobbs the Printers Ltd

**British Library Cataloguing in Publication Data.**

A catalogue record for this book is available from the British Library

# MISRA-C:2004

# Guidelines for the use of the C language in critical systems

October 2004

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 or guarantee portability and re-use.*

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

# Foreword

In preparing the original MISRA-C:1998 [1] document, it was hoped to make some impact in the use of software within the UK automotive industry. Since 1998, the successes and global use of MISRA-C1 across automotive, aerospace, medical and other industries has been staggering.

Since the publication of MISRA-C:1998, we have received considerable comment of the good, bad, and in some cases impractical rules included. We therefore set about the task of producing an update, MISRA-C:2004 (this document), which improves on, and corrects the issues faced by software engineers implementing MISRA-C:1998.

While producing MISRA-C:2004, the question of addressing the 1999 C standard [8] arose. At this time, only issues with MISRA-C:1998 are addressed due to the limited support for C99 on embedded microprocessors.

For the last few years, a dedicated group have met, representing a broad range of interests to refine and produce MISRA-C:2004. I would like to thank this team for their effort and support.

I would also like to recognise our global partners who have aided our global preparation of MISRA-C:2004. In the USA, this has been with the SAE J2632 committee led by Bruce Emaus. In Japan, we have worked with representatives of JSAE, JAMA, and the MISRA Study Group, and I would particularly like to thank Takao Futagami for his role in co-ordinating access to these groups.

I would also like to thank all those in a wider group who have provided comments and support to the MISRA-C effort. This includes all those who participated in the review during 2003, which led to some rules being re-designed to address the comments received.

In presenting MISRA-C:2004, we have attempted to refine the document in a number of ways.

- We have replaced general blanket rules with specific targeted rules.
- We have replaced “as appropriate” rules with definitive do / do not rules.
- We have introduced rules for arithmetic operations which provide a sound base for simple and robust statements.
- We have 122 mandatory and 20 advisory rules.
- We have removed 15 rules which did not make sense.
- We have split complex rules into component parts.
- We have attempted to remain compatible with MISRA-C:1998, to prevent MISRA-C:1998 code needing to be modified to conform to MISRA-C:2004.

The MISRA-C project remains on-going, and this document has now been supplemented with an Exemplar Test Case Suite available at at www.misra-c.com/forum to provide examples of compliant and non-compliant code.

This document specifies a subset of the C programming language which is intended to be suitable for embedded systems. It contains a list of rules concerning the use of the C programming language together with justifications and examples.

Gavin McCall BSc (Hons), MSc, C.Eng, MIEE

MISRA-C Team Leader

# 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 |

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

| | | | |
|---|---|---|---|
| Ganapti Avadhani | James F. Gimpel | Patrick Markl | Kotaro Seike |
| Walter Banks | Martin Gray | Martin Meyer | Yoko Sekimoto |
| David Blyth | Robert Gruszczynski | Claude Mignen | Raul Selgado |
| Paul Boultwood | Rob Hague | Andrew Mijat | Darren Sexton |
| Richard Burke | Trenton Haines | Svante Möller | Norman S. Shelvik |
| Ian Chalinder | Takahiro Hashimoto | Olwen Morgan | Joao Silva |
| Kwok Chan | Les Hatton | Stephen D. Morton | Jochem Spohr |
| Paul Clark | Manabu Hirozawa | Tadanori Nakagawa | Geert Starre |
| Valery Creux | Hartmut Hörner | Heather Neil | Richard Swanton |
| David Crocker | Motozo Ikeda | Jacob Nevins | Benjamin Sweet |
| William Derouchie | Yoshikazu Imura | Michael Niemetz | Musubi Uno |
| Alain Deutsch | Dern Jérôme | Yuji Ninagawa | Yan Wang |
| Todd Dowty | Bernd Jesse | Kenji Ohgoshi | David Ward |
| Manoj Dwivedi | Stuart Jobbins | Satoru Ohtsuka | Michael Warmuth |
| Mike Ellims | Derek Jones | Greg Palarski | Thomas Wengler |
| Bruce Emaus | Jeff Kanoza | Stephen Parker | Paul Whiston |
| Andy Fiore | Shigeyuki Kawana | Richard Parkins | Karl Woelfer |
| William Forbes | Roland Kilgore | Patrik Persson | David Wright |
| S.W.Freeman | Helmar Kuder | Bernd Reh | Shoji Yamashita |
| Takao Futagami | Koyo Kuroda | Chris Sampson | |
| Robert Galles | Tom Lake | Dana Sawyer | |
| Jon Garnsworthy | Lars Magnusson | Walter Schilling | |

The contributions of Society of Automotive Engineering (SAE) Embedded Software Taskforce, Japanese Society of Automotive Engineers (JSAE), Japanese Automotive Manufacturers Association (JAMA) and Herstellerinitiative Software (HIS) Working Group (Arbeitskreis) “Software Test” are acknowledged.

# Contents

# Contents (continued)

# 1. Background

# 1. Background – The use of C and issues with it

## 1.1 The use of C in the automotive industry

MISRA-C:1998 [1] was published in 1998. This document is a revision to that document to address the issues identified with that first version.

The C programming language [2] 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 [2] 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 [3] 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".

Note 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 standard used for this document is the C programming language as defined by ISO/IEC 9899:1990 [2], amended and corrected by ISO/IEC 9899/COR1:1995 [4], ISO/IEC 9899/AMD1:1995 [5], and ISO/IEC 9899/COR2: 1996 [6]. The base 1990 document [7] is the ISO version of ANSI X3.159-1989 [2]. In content, the ISO/IEC standard and the ANSI standard are identical. 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 [7] contains a useful appendix giving the rationale behind some of the decisions made by the standardization committee; this does not appear in the ISO edition.

This working group has considered ISO/IEC 9899:1999 [8]. At the time of publication (October 2004), no commercial embedded C99 compilers were available.

## 1.5 Introduction to this edition

Since the original publication of MISRA-C:2004, 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 Bulletin Board, a number of issues have been identified. A Technical Corrigendum document has been released providing clarification on these issues. The clarifications described by the Technical Corrigendum are incorporated into the Exemplar Suite release 1.0 dated 17 July 2007. This edition of MISRA-C:2004 integrates the modifications contained in the Technical Corrigendum.

# 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” [9] in 1994, which 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 safety integrity level, are recognised to be of major importance. Section 3.2.4.3 (b) and Table 3 of the MISRA Guidelines [9] 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 [9] 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.

# 3. Scope

# 3. MISRA-C: Scope

## 3.1 Base languages issues

The MISRA Guidelines [9] (Table 3) requires that “a restricted subset of a standardized structured language” be used. For C, this means that the language must only be used as defined in the ISO standard. This therefore precludes the use of:

- K&R C (as defined in the First Edition of “The C Programming language” by Kernighan and Ritchie)
- C++
- Proprietary extensions to the C language

## 3.2 Issues not addressed

Issues of style and code metrics are somewhat subjective. It would be hard for any group of people to agree on what was appropriate, and it would be inappropriate for MISRA to give definitive advice. What is important is not the exact style guidelines adopted by a user, or the particular metrics used, but rather that the user defines style guidelines and appropriate metrics and limits (see sections 4.2.2 and 4.2.4).

The MISRA consortium is not in a position to recommend particular vendors or tools to enforce the restrictions adopted. The user of this document is free to choose tools, and vendors are encouraged to provide tools to enforce the rules. The onus is on the user of this document to demonstrate that their tool set enforces the rules adequately.

## 3.3 Applicability

This document is designed to be applied to production code in automotive embedded systems.

In terms of the execution environments defined by ISO/IEC 9899 [2] (section 5.1.2), this document is aimed at a “free-standing environment”, although it also addresses library issues since some standard libraries will often be supplied with an embedded compiler.

Most of the requirements of this document may be applicable to embedded systems in other sectors if such use is considered appropriate. The requirements of this document will not necessarily be applicable to hosted systems.

It is also not necessary to apply the rules in full when performing compiler and static tool benchmarking. Sometimes it will be necessary to deliberately break the rules when benchmarking tools, so as to measure the tools’ responses.

## 3.4 Prerequisite knowledge

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 appropriate training and are competent C language programmers.

## 3.5 C++ issues

C++ is a different language to C, and the scope of this document does not include the C++ language, nor does it attempt to comment on the suitability or otherwise of C++ for programming safety-related systems. However the following comments about the use of C++ compilers and code should be noted.

C++ is not simply a super-set of C (i.e. C plus extra features). There are a few specific constructs which have different interpretations in C and C++. In addition, valid C code may contain identifiers which in C++ would be interpreted as reserved words. For both of these reasons, code written in C and conforming to the ISO C standard will not necessarily compile under a C++ compiler with the same results as under a true C compiler. Thus the use of C++ compilers for compiling C code is deprecated by this document. If, for reasons of availability, a C++ compiler must be used to compile C code then the areas of difference between the two languages must first be fully understood.

However, the use of additional compilers as extra static checking tools is encouraged. For this purpose, where the executable code is of no interest, C++ compilers may be used, and indeed can offer benefits because of the greater type checking they have over C.

Where a compiler which is marketed as "C++" has a strictly conforming ISO C mode of operation then this is equivalent to a C compiler and may be used as such (in the C mode only). The same is true of any other tool which includes a conforming ISO C compiler as part of its functionality.

C++ comments should not be used in C code. Although many C compilers support this form of comment (denoted by `//`), they are not a part of ISO standard C (see Rule 2.2).

## 3.6 Auto-generated code issues

If a code-generating tool is to be used, then it will be necessary to select an appropriate tool and undertake validation. Apart from suggesting that adherence to the requirements of this document may provide one criterion for assessing a tool, no further guidance is given on this matter and the reader is referred to the HSE recommendations for COTS [10].

Auto-generated code must be treated in just the same manner as manually produced code for the purpose of validation (See MISRA Guidelines [9] 3.1.3, Planning for V&V).

# 4. Using MISRA-C

## 4.1 The software engineering context

Using a programming language to produce source code is just one activity in the software development process. Adhering to best practice in this one activity is of very limited value if the other commonly accepted development issues are not addressed. This is especially true for the production of safety-related systems. These issues are all addressed in the MISRA Guidelines [9] and, for example, include:

- Documented development process
- Quality system capable of meeting the requirements of ISO 9001/ISO 90003/TickIT [11, 12, 13]
- Project management
- Configuration management
- Hazard analysis
- Requirements
- Design
- Coding
- Verification
- Validation

It is necessary for the software developers to justify that the whole of their development process is appropriate for the type of system they are developing. This justification will be incomplete unless a hazard analysis activity has been performed to determine the safety integrity level of the system.

## 4.2 The programming language and coding context

Within the coding phase of the software development process, the language subset is just one aspect of many and again adhering to best practice in this aspect is of very limited value if the other issues are not addressed. Key issues, following choice of language, are:

- Training
- Style guide
- Compiler selection and validation
- Checking tool validation
- Metrics
- Test coverage

All decisions made on these issues need to be documented, along with the reasons for those decisions, and appropriate records should be kept for any activities performed. Such documentation may then be included in a safety justification if required.

### 4.2.1 Training

In order to ensure an appropriate level of skill and competence on the part of those who produce the C source code formal training should be provided for:

- The use of the C programming language for embedded applications
- The use of the C programming language for high-integrity and safety-related systems
- The use of static checking tools used to enforce adherence to the subset

### 4.2.2 Style guide

In addition to adopting the subset, an organisation should also have an in-house style guide. This will contain guidance on issues which do not directly affect the correctness of the code but rather define a "house style" for the appearance of the source code. These issues are likely to be subjective. Typical issues to be addressed by a style guide include:

- code layout and use of indenting
- layout of braces "`{ }`" and block structures
- statement complexity
- naming conventions
- use of comment statements
- inclusion of company name, copyright notice and other standard file header information

While some of the content of the style guide may only be advisory, some may be mandatory. However the enforcement of the style guide is outside the scope of this document.

For further information on style guides see [14].

### 4.2.3 Tool selection and validation

When choosing a compiler (which should be understood to include the linker), an ISO C compliant compiler should be used whenever possible. Where the use of the language is reliant on an "implementation-defined" feature (as identified in Annex G.3 of the ISO standard [2]) then the developer must benchmark the compiler to establish that the implementation is as documented by the compiler writer. See section 5.5.2 for more explanation of Annex G.

When choosing a static checker tool it is clearly desirable that the tool be able to enforce as many of the rules in this document as possible. To this end it is essential that the tool is capable of performing checks across the whole program, and not just within a single source file. In addition, where a checker tool has capabilities to perform checks beyond those required by this document it is recommended that the extra checks are used.

The compiler and the static checking tool are generally seen as "trusted" processes. This means that there is a certain level of reliance on the output of the tools, therefore the developer must ensure that this trust is not misplaced. Ideally this should be achieved by the tool supplier running appropriate validation tests. Note that, while it is possible to use a validation suite to test a compiler for an embedded target, no formal validation scheme exists at the time of publication of this document. In addition, the tools should have been developed to a quality system capable of meeting the requirements of ISO 9001/ISO 90003 [11, 12, 13]

It should be possible for the tool supplier to show records of verification and validation activities together with change records that show a controlled development of the software. The tool supplier should have a mechanism for:

- recording faults reported by the users
- notifying existing users of known faults
- correcting faults in future releases

The size of the existing user base together with an inspection of the faults reported over the previous 6 to 12 months will give an indication of the stability of the tool.

It is often not possible to obtain this level of assurance from tool suppliers, and in these cases the onus is on the developer to ensure that the tools are of adequate quality.

Some possible approaches the developer could adopt to gain confidence in the tools are:

- perform some form of documented validation testing
- assess the software development process of the tool supplier
- review the performance of the tool to date

The validation test could be performed by creating code examples to exercise the tools. For compilers this could consist of known good code from a previous application. For a static checking tool, a set of code files should be written, each breaking one rule in the subset and together covering as many as possible of the rules. For each test file the static checking tool should then find the non-conformant code. Although such tests would necessarily be limited they would establish a basic level of tool performance.

It should be noted that validation testing of the compiler must be performed for the same set of compiler options, linker options and source library versions used when compiling the product code.

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

### 4.2.4 Source complexity metrics

The use of source code complexity metrics is highly recommended. These can be used to prevent unwieldy and untestable code being written by looking for values outside of established norms. The use of tools to collect the data is also highly recommended. Many of the static checking tools that may be used to enforce the subset also have the capability for producing metrics data.

For details of possible source code metrics see “Software Metrics: A Rigorous and Practical Approach” by Fenton and Pfleeger [15] and the MISRA report on Software Metrics [16].

### 4.2.5 Test coverage

The expected statement coverage of the software should be defined before the software is designed and written. Code should be designed and written in a manner that supports high statement coverage during testing. The term “Design For Test” (DFT) has been applied to this concept in mechanical, electrical and electronic engineering. This issue needs to be considered during the activity of writing the code, since the ability to achieve high statement coverage is an emergent property of the source code.

Use of a subset, which reduces the number of implementation-dependent features, and increases the rigour of module interface compatibility can lead to software that can be integrated and tested with greater ease.

Balancing the following metrics can facilitate achieving high statement coverage:

- code size
- cyclomatic complexity
- static path count

With a planned approach, the extra effort expended on software design, language use and design for test is more than offset by the reduction in the time required to achieve high statement coverage during test. See [16, 17].

## 4.3 Adopting the subset

In order to develop code that adheres to the subset the following steps need to be taken:

- Produce a compliance matrix which states how each rule is enforced
- Produce a deviation procedure
- Formalise the working practices within the quality management system

### 4.3.1 Compliance matrix

In order to ensure that the source code written does conform to the subset it is necessary to have measures in place which check that none of the rules have been broken. The most effective means of achieving this is to use one or more of the static checking tools that are available commercially. Where a rule cannot be checked by a tool, then a manual review will be required.

In order to ensure that all the rules have been covered then a compliance matrix should be produced which lists each rule and indicates how it is to be checked. See Table 1 for an example, and see Appendix A for a summary list of the rules, which could be used to assist in generating a full compliance matrix.

| Rule No. | Compiler 1 | Compiler 2 | Checking Tool 1 | Checking Tool 2 | Manual Review |
|---|---|---|---|---|---|
| **1.1** | warning 347 | | | | |
| **1.2** | | error 25 | | | |
| **1.3** | | | message 38 | | |
| **1.4** | | | | warning 97 | |
| **1.5** | | | | | Proc x.y |
| | | | | | |

**Table 1: Example compliance matrix**

If the developer has additional local restrictions, these too can be added to the compliance matrix. Where specific restrictions are omitted, full justifications shall be given. These justifications must be fully supported by a C language expert together with manager level concurrence.

### 4.3.2 Deviation procedure

It is recognised that in some instances it may be necessary to deviate from the rules given in this document. For example, source code written to interface with the microprocessor hardware will inevitably require the use of proprietary extensions to the language.

In order for the rules to have authority it is necessary that a formal procedure be used to authorise these deviations rather than an individual programmer having discretion to deviate at will. It is expected that the procedure will be based around obtaining a sign-off for each deviation, or class of deviation. The use of a deviation must be justified on the basis of both necessity and safety. While this document does not give, nor intend to imply, any grading of importance of each of the rules, it is accepted that some provisions are more critical than others. This should be reflected in the deviation procedure, where for more serious deviations greater technical competence is required to assess the risk incurred and higher levels of management are required to accept this increased risk. Where a formal quality management system exists, the deviation procedure should be a part of this system.

Deviations may occur for a specific instance, i.e. a one-off occurrence in a single file, or for a class of circumstances, i.e. a systematic use of a particular construct in a particular circumstance, for example the use of a particular language extension to implement an input/output operation in files which handle serial communications.

Strict adherence to all rules is unlikely and, in practice, deviations associated with individual situations, are admissible. There are two categories of deviation.

- Project Deviation: A Project Deviation is defined as a permitted relaxation of rule requirements to be applied in specified circumstances. In practice, Project Deviations will usually be agreed at the start of a project.
- Specific Deviation: A Specific Deviation will be defined for a specific instance of a rule violation in a single file and will typically be raised in response to circumstances which arise during the development process.

Project Deviations should be reviewed regularly and this review should be a part of the formal deviation process.

Many, if not most, of the circumstances where rules need to be broken are concerned with input/output operations. It is recommended that the software be designed such that input/output concerns are separated from the other parts of the software. As far as possible Project Deviations should then be restricted to this input/output section of the code. Code subject to Project Deviations should be clearly marked as such.

The purpose of this document is to avoid problems by thinking carefully about the issues and taking all responsible measures to avoid the problems. The deviation procedure should not be used to undermine this intention. In following the rules in this document the developer is taking advantage of the effort expended by MISRA in understanding these issues. If the rules are to be deviated from, then the developer is obliged to understand the issues for themselves. All deviations, standing and specific, should be documented.

For example, if it is known beforehand that it will be difficult to adhere to a rule, the software developer should submit a written Project Deviation Request and agreement with the customer should be obtained prior to programming.

A Project Deviation Request should include the following:

- Details of the deviation, i.e. the rule that is being violated
- Circumstances in which the need for the deviation arises
- Potential consequences which may result from the deviation
- Justification for the deviation
- A demonstration of how safety is assured

When the need for a deviation arises during or at the end of the development process, the software developer should submit a written Specific Deviation Request.

A Specific Deviation Request should include the following:

- Details of the deviation, i.e. the rule that is being violated
- Potential consequences which may result from the deviation
- Justification for the deviation
- A demonstration of how safety is assured

Detailed implementation of these procedures is left to the discretion of the user.

### 4.3.3 Formalisation within quality system

The use of the subset, the static checking tools and deviation procedure should be described by formal documents within the quality management system. They will then be subject to the internal and external audits associated with the quality system and this will help ensure their consistent use.

### 4.3.4 Introducing the subset

Where an organisation has an established C coding environment it is recommended that the requirements of this document be introduced in a progressive manner (see chapter 5 of Hatton [3]). It may take 1 to 2 years to implement all aspects of this document.

Where a product contains legacy code written prior to the use of the subset, it may be impractical to rewrite it to bring it into conformance with the subset. In these circumstances the developer must decide upon a strategy for managing the introduction of the subset (for example: all new modules will be written to the subset and existing modules will be rewritten to the subset if they are subject to a change which involves more than 30% of the non-comment source lines).

## 4.4 Claiming compliance

Compliance can only be claimed for a product and not for an organisation.

When claiming compliance to the MISRA-C document for a product a developer is stating that evidence exists to show:

- A compliance matrix has been completed which shows how compliance has been enforced
- All of the C code in the product is compliant with the rules of this document or subject to documented deviations
- A list of all instances of rules not being followed is being maintained, and for each instance there is an appropriately signed-off deviation
- The issues mentioned in section 4.2 have been addressed

## 4.5 Continuous improvement

Adherence to the requirements of this document should only be considered as a first step in a process of continuous improvement. Users should be aware of the other literature on the subject (see references) and actively seek to improve their development process by the use of metrics.

# 5. Introduction to the rules

This section explains the presentation of the rules in section 6 of this document. It serves as an introduction to the main content of the document as presented in that section.

## 5.1 Rule classification

Every rule in section 6 is classified as being either "required" or "advisory", as described below. Beyond this basic classification the document does not give, nor intend to imply, any grading of importance of each of the rules. All required rules should be considered to be of equal importance, as should all advisory rules. The omission of an item from this document does not imply that it is less important.

The meanings of "required" and "advisory" rules are as follows.

### 5.1.1 Required rules

These are mandatory requirements placed on the programmer. There are 122 "required" rules in this document. C code which is claimed to conform to this document shall comply with every required rule (with formal deviations required where this is not the case, as described in section 4.3.2).

### 5.1.2 Advisory rules

These are requirements placed on the programmer that should normally be followed. However they do not have the mandatory status of required rules. There are 20 "advisory" rules in this document. Note that the status of "advisory" does not mean that these items can be ignored, but that they should be followed as far as is reasonably practical. Formal deviations are not necessary for advisory rules, but may be raised if it is considered appropriate.

## 5.2 Organisation of rules

The rules are organised under different topics within the C language. However there is inevitably overlap, with one rule possibly being relevant to a number of topics. Where this is the case the rule has been placed under the most relevant topic.

## 5.3 Redundancy in the rules

There are a few cases within this document where a rule is given which refers to a language feature which is banned or advised against elsewhere in the document. This is intentional. It may be that the user chooses to use that feature, either by raising a deviation against a required rule, or by choosing not to follow an advisory rule. In this case the second rule, constraining the use of that feature, becomes relevant.

## 5.4 Presentation of rules

The individual requirements of this document are presented in the following format:

> **Rule <*number*> (<*category*>): <*requirement text*>**
>
> [<*source ref*>]
>
> Normative text

where the fields are as follows:

- <*number*> Every rule has a unique number. This number consists of a rule group prefix and a group member suffix.
- <*category*> is one of “required” or “advisory”, as explained in section 5.1.
- <*requirement text*> The rule itself.
- <*source ref*> This indicates the primary source(s) which led to this item or group of items, where applicable. See section 5.5 for an explanation of the significance of these references, and a key to the source materials.

Normative text is provided for each item or group of related items. In order to conform with MISRA-C:2004, it is necessary to meet the requirements of this normative text.

The normative text gives, where appropriate, some explanation of the underlying issues being addressed by the rule(s), and examples of how to apply the rule(s). If there is no explanatory text immediately following a rule then the relevant text will be found following the group of rules, and applies to all the rules which precede it. Similarly a source reference following a group of rules applies to the whole group.

The normative text is not intended as a tutorial in the relevant language feature, as the reader is assumed to have a working knowledge of the language. Further information on the language features can be obtained by consulting the relevant section of the language standard or other C language reference books. Where a source reference is given for one or more of the “Annex G” items in the ISO standard, then the original issue raised in the ISO standard may provide additional help in understanding the rule.

Within the rules and their normative text, the following font styles are used to represent C keywords and C code:

C *keywords* appear in italic text

C `code` appears in a monospaced font, either within other text or as

```
separate code fragments;
```

Note that where code is quoted, the fragments may be incomplete (for example an *if* statement without its body). This is for the sake of brevity.

In code fragments, the following *typedef*’d types have been assumed (to comply with Rule 6.3):

```
char_t      plain 8 bit character
uint8_t     unsigned 8 bit integer
uint16_t    unsigned 16 bit integer
uint32_t    unsigned 32 bit integer
int8_t      signed 8 bit integer
int16_t     signed 16 bit integer
int32_t     signed 32 bit integer
float32_t   32 bit floating-point
float64_t   64 bit floating-point
```

Non-specific variable names are constructed to give an indication of the type. For example:

```
uint8_t     u8a;
sint32_t    s32a;
```

## 5.5 Understanding the source references

Where a rule originates from one or more published sources these are indicated in square brackets after the rule. This serves two purposes. Firstly the specific sources may be consulted by a reader wishing to gain a fuller understanding of the rationale behind the rule (for example when considering a request for a deviation). Secondly, with regard to issues in "Annex G" of the ISO standard, the type of the source gives extra information about the nature of the problem (see section 5.5.2).

Rules which do not have a source reference may have originated from a contributing company's in-house standard, or have been suggested by a reviewer, or be widely accepted "good practice".

A key to the references, and advice on interpreting them, is given below.

### 5.5.1 Key to the source references

| Reference | Source |
|---|---|
| | **Annex G of ISO/IEC 9899 [2]** |
| Unspecified | Unspecified behaviour (G.1) |
| Undefined | Undefined behaviour (G.2) |
| Implementation | Implementation-defined behaviour (G.3) |
| Locale | Locale-specific behaviour (G.4) |
| | **Other** |
| MISRA Guidelines | The MISRA Guidelines [9] |
| K&R | Kernighan and Ritchie [18] |
| Koenig | "C Traps and Pitfalls", Koenig [19] |
| IEC 61508 | IEC 61508:1998–2000 [20] |

Where numbers follow the reference, they have the following meanings:

- Annex G of ISO/IEC 9899 references: The number of the item in the relevant section of the Annex, numbered from the beginning of that section. So for example [Locale 2] is the second item in section G.4 of the standard.
- In other references, the relevant page number is given (unless stated otherwise).

## 5.5.2 Understanding Annex G references

Where a rule is based on issues from Annex G of the ISO C standard, it is helpful for the reader to understand the distinction between "unspecified", "undefined", "implementation-defined" and "locale-specific" issues. These are explained briefly here, and further information can be found in Hatton [3].

### 5.5.2.1 Unspecified

These are language constructs that must compile successfully, but in which the compiler writer has some freedom as to what the construct does. An example of this is the "order of evaluation" described in Rule 12.2. There are 22 such issues.

It is unwise to place any reliance on the compiler behaving in a particular way. The compiler need not even behave consistently across all possible constructs.

### 5.5.2.2 Undefined

These are essentially programming errors, but for which the compiler writer is not obliged to provide error messages. Examples are invalid parameters to functions, or functions whose arguments do not match the defined parameters.

These are particularly important from a safety point of view, as they represent programming errors which may not necessarily by trapped by the compiler.

### 5.5.2.3 Implementation-defined

These are a bit like the "unspecified" issues, the main difference being that the compiler writer must take a consistent approach and document it. In other words the functionality can vary from one compiler to another, making code non-portable, but on any one compiler the behaviour should be well defined. An example of this is the behaviour of the integer division and remainder operators "`/`" and "`%`" when applied to one positive and one negative integer. There are 76 such issues.

These tend to be less critical from a safety point of view, provided the compiler writer has fully documented their approach and then stuck to what they have implemented. It is advisable to avoid these issues where possible.

### 5.5.2.4 Locale-specific

These are a small set of features which may vary with international requirements. An example of this is the facility to represent a decimal point by the "`,`" character instead of the "`.`" character. There are 6 such issues. No issues arising from this source are addressed in this document.

## 5.6 Scope of rules

In principle the rules are to be applied across the file boundaries of the complete set of program files for the application. While the majority of the rules can be applied within a single translation unit, rule numbers 1.1, 1.2, 1.3, 1.4, 2.1, 3.6, 5.1, 5.3, 5.4, 5.5, 5.6, 5.7, 8.4, 8.8, 8.9, 8.10, 12.2, 12.4, 14.1, 14.2, 16.2, 16.4, 17.2, 18.1, 18.2, 18.3 and 21.1 must be applied across the collective set of files.

# 6. Rules

## 6.1 Environment

**Rule 1.1 (required):** **All code shall conform to ISO/IEC 9899:1990 "Programming languages — C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.**

[MISRA Guidelines Table 3; IEC 61508 Part 7: Table C.1]

These guidelines are based on ISO/IEC 9899:1990 [2] amended and corrected by ISO/IEC 9899/COR1:1995 [4], ISO/IEC 9899/AMD1:1995 [5], and ISO/IEC 9899/COR2:1996 [6]. No claim is made as to their suitability with respect to the ISO/IEC 9899:1999 [8] version of the standard. Any reference in this document to "Standard C" refers to the older ISO/IEC 9899:1990 [2] standard.

It is recognised that it will be necessary to raise deviations (as described in section 4.3.2) to permit certain language extensions, for example to support hardware specific features.

Deviations are required if the environmental limits as specified in ISO/IEC 9899:1990 5.2.4 [2] are exceeded, other than as allowed by Rule 5.1.

**Rule 1.2 (required):** **No reliance shall be placed on undefined or unspecified behaviour.**

This rule requires that any reliance on undefined and unspecified behaviour, which is not specifically addressed by other rules, shall be avoided. Where a specific behaviour is explicitly covered in another rule, only that specific rule needs to be deviated if required. See ISO/IEC 9899:1990 Appendix G [2] for a complete list of these issues.

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

[Unspecified 11]

If a module is to be implemented in a language other than C, or compiled on a different C compiler, then it is essential to ensure that the module will integrate correctly with other modules. Some aspects of the behaviour of the C language are dependent on the compiler, and therefore these must be understood for the compiler being used. Examples of issues that need to be understood are: stack usage, parameter passing and the way in which data values are stored (lengths, alignments, aliasing, overlays, etc.)

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

[Undefined 7; Implementation 5, 6]

The ISO standard requires external identifiers to be distinct in the first 6 characters. However compliance with this severe and unhelpful restriction is considered an unnecessary limitation since most compilers/linkers allow at least 31 character significance (as for internal identifiers).

The compiler/linker must be checked to establish this behaviour. If the compiler/linker is not capable of meeting this limit, then use the limit of the compiler.

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

Floating-point arithmetic has a range of problems associated with it. Some (but not all) of the problems can be overcome by using an implementation that conforms to a recognised standard. An example of an appropriate standard is ANSI/IEEE Std 754 [21].

The definition of the floating-point types, in accordance with Rule 6.3, provides an opportunity for noting the floating-point standard in use, for example:

```
/* IEEE 754 single-precision floating-point */
typedef float float32_t;
```

## 6.2 Language extensions

**Rule 2.1 (required): Assembly language shall be encapsulated and isolated.**

[Unspecified 11]

Where assembly language instructions are required it is recommended that they be encapsulated and isolated in either (a) assembler functions, (b) C functions or (c) macros.

For reasons of efficiency it is sometimes necessary to embed simple assembly language instructions in-line, for example to enable and disable interrupts. If it is necessary to do this for any reason, then it is recommended that it be achieved by using macros.

Note that the use of in-line assembly language is an extension to standard C, and therefore also requires a deviation against Rule 1.1.

```
#define NOP asm("   NOP")
```

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

This excludes the use of `//` C99 style comments and C++ style comments, since these are not permitted in C90. Many compilers support the `//` style of comments as an extension to C90. The use of `//` in preprocessor directives (e.g. *#define*) can vary. Also the mixing of `/* … */` and `//` is not consistent. This is more than a style issue, since different (pre C99) compilers may behave differently.

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

C does not support the nesting of comments even though some compilers support this as a language extension. A comment begins with `/*` and continues until the first `*/` is encountered. Any `/*` occurring inside a comment is a violation of this rule. Consider the following code fragment:

```
    /* some comment, end comment marker accidentally omitted

    <<New Page>>
    Perform_Critical_Safety_Function(X);
    /* this comment is not compliant */
```

In reviewing the page containing the call to the function, the assumption is that it is executed code.

Because of the accidental omission of the end comment marker, the call to the safety critical function will not be executed.

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

Where it is required for sections of source code not to be compiled then this should be achieved by use of conditional compilation (e.g. *#if* or *#ifdef* constructs with a comment). Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.

## 6.3 Documentation

**Rule 3.1 (required): All usage of implementation-defined behaviour shall be documented.**

This rule requires that any reliance on implementation-defined behaviour, which is not specifically addressed by other rules, shall be documented, for example by reference to compiler documentation. Where a specific behaviour is explicitly covered in another rule, only that specific rule needs to be deviated if required. See ISO/IEC 9899:1990 Appendix G [2] for a complete list of these issues.

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

For example, ISO 10646 [22] defines an international standard for mapping character sets to numeric values. For portability, "character-constants" and "string-literals" should only contain characters that map to a documented subset. 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.

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

[Implementation 18]

Potentially an ISO compliant compiler can do one of two things when dividing two signed integers, one of which is positive and one negative. Firstly it may round up, with a negative remainder (e.g. -5/3 = -1 remainder -2), or secondly it may round down with a positive remainder (e.g. -5/3 = -2 remainder +1).

It is important to determine which of these is implemented by the compiler and to document it for programmers, especially if it is the second (perhaps less intuitive) implementation.

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

[Implementation 40]

This rule places a requirement on the user of this document to produce a list of any pragmas they choose to use in an application. The meaning of each pragma shall be documented. There shall be sufficient supporting description to demonstrate that the behaviour of the pragma, and its implications for the application, have been fully understood.

Any use of pragmas should be minimised, localised and encapsulated within dedicated functions wherever possible.

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

[Unspecified 10; Implementation 30, 31]

This is a particular problem where bit fields are used because of the poorly defined aspects of bit fields described under Rules 6.4 and 6.5. The "bit field" facility in C is one of the most poorly defined parts of the language. There are two main uses to which bit fields could be put:

- To access the individual bits, or groups of bits, in larger data types (in conjunction with unions). This use is not permitted (see Rule 18.4).
- To allow flags or other short-length data to be packed to save storage space.

The packing together of short-length data to economise on storage is the only acceptable use of bit fields envisaged in this document. Provided the elements of the structure are only ever accessed by their name, the programmer needs to make no assumptions about the way that the bit fields are stored within the structure.

It is recommended that structures be declared specifically to hold the sets of bit fields, and do not include any other data within the same structure. Note that Rule 6.3 need not be followed in defining bit-fields, since their lengths are specified in the structure.

If the compiler has a switch to force bit fields to follow a particular layout then this could assist in such a justification.

For example the following is acceptable:

```
struct message                /* Struct is for bit-fields only */
{
   signed   int little: 4;    /* Note: use of basic types is required */
   unsigned int x_set:  1;
   unsigned int y_set:  1;
} message_chunk;
```

If using bit fields, be aware of the potential pitfalls and areas of implementation-defined (i.e. non-portable) behaviour. In particular the programmer should be aware of the following:

- The alignment of the bit fields in the storage unit is implementation-defined, that is whether they are allocated from the high end or low end of the storage unit (usually a byte).
- Whether or not a bit field can overlap a storage unit boundary is also implementation-defined (e.g. if a 6-bit field and a 4-bit field are declared in that order, whether the 4 bit field will start a new byte or whether it will be 2 bits in one byte and 2 bits in the next).

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

[IEC 61508 Part 3]

This rule refers to any libraries used in the production code, which therefore may include standard libraries supplied with the compiler, other third-party libraries, or libraries designed in-house. This is recommended by IEC 61508 Part 3.

## 6.4 Character sets

**Rule 4.1 (required):** **Only those escape sequences that are defined in the ISO C standard shall be used.**

[Undefined 11; Implementation 11]

Only "simple-escape-sequences" in ISO/IEC 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.

**Rule 4.2 (required):** **Trigraphs shall not be used.**

Trigraphs are denoted by a sequence of 2 question marks followed by a specified third character (e.g. `??-` represents a "~" (tilde) character and `??)` represents a "`]`"). They can cause accidental confusion with other uses of two question marks. For example the string

```
"(Date should be in the form ??-??-??)"
```

would not behave as expected, actually being interpreted by the compiler as

```
"(Date should be in the form ~~]"
```

## 6.5 Identifiers

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

[Undefined 7; Implementation 5, 6]

The ISO standard requires internal identifiers to be distinct in the first 31 characters to guarantee code portability. This limitation shall not be exceeded, even if the compiler supports it. This rule shall apply across all name spaces. Macro names are also included and the 31 character limit applies before and after substitution.

The ISO standard requires external identifiers to be distinct in the first 6 characters, regardless of case, to guarantee optimal portability. However this limitation is particularly severe and is considered unnecessary. The intent of this rule is to sanction a relaxation of the ISO requirement to a degree commensurate with modern environments and it shall be confirmed that 31 character/case significance is supported by the implementation.

Note that there is a related issue with using identifier names that differ by only one or a few characters, especially if the identifier names are long. The problem is heightened if the differences are in easily mis-read characters like 1 (one) and l (lower case L), 0 and O, 2 and Z, 5 and S, or n and h. It is recommended to ensure that identifier names are always easily visually distinguishable. Specific guidelines on this issue could be placed in the style guidelines (see section 4.2.2).

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

The terms outer and inner scope are defined as follows. Identifiers that have file scope can be considered as having the outermost scope. Identifiers that have block scope have a more inner scope. Successive, nested blocks, introduce more inner scopes. The rule is only to disallow the case where a second inner definition hides an outer definition. If the second definition does not hide the first definition, then this rule is not violated.

Hiding identifiers with an identifier of the same name in a nested scope leads to code that is very confusing. For example:

```
int16_t i;
{
   int16_t i;    /* This is a different variable                   */
                 /* This is not compliant                          */
   i = 3;        /* It could be confusing as to which i this refers */
}
```

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

No *typedef* name shall be reused either as a *typedef* name or for any other purpose.

For example:

```
{
   typedef unsigned char uint8_t;
}
{
   typedef unsigned char uint8_t; /* Not compliant - redefinition */
}
{
   unsigned char uint8_t;     /* Not compliant - reuse of uint8_t */
}
```

*typedef* names shall not be reused anywhere within a program. The same *typedef* shall not be duplicated anywhere in the source code files even if the declarations are identical. 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.

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

No tag name shall be reused either to define a different tag or for any other purpose within the program. ISO/IEC 9899:1990 [2] does not define the behaviour when an aggregate declaration uses a tag in different forms of type specifier (*struct* or *union*). Either all uses of the tag should be in structure type specifiers, or all uses should be in union type specifiers, For example:

```
struct stag { uint16_t a; uint16_t b; };

struct stag a1 = { 0, 0 };      /* Compliant - compatible with above  */
union stag  a2 = { 0, 0 };      /* Not compliant - not compatible with
                                   previous declarations              */

void foo(void)
{
   struct stag { uint16_t a; }; /* Not compliant - tag stag redefined */
}
```

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

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

Regardless of scope, no identifier with static storage duration should be re-used across any source files in the system. This includes objects or functions with external linkage and any objects or functions with the *static* storage class specifier.

While the compiler can understand this and is in no way confused, the possibility exists for the user to incorrectly associate unrelated variables with the same name.

One example of this confusion is having an identifier name with internal linkage in one file and the same identifier name with external linkage in another file.

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

Name space and scope are different. This rule is not concerned with scope. For example, ISO C allows the same identifier (`vector`) for both a tag and a *typedef* at the same scope.

```
typedef struct vector { uint16_t x ; uint16_t y; uint16_t z; } vector;
/* Rule violation ^^                                             ^^ */
```

ISO C defines a number of different name spaces (see ISO/IEC 9899:1990 6.1.2.3 [2]). It is technically possible to use the same name in separate name spaces to represent completely different items. However this practice is deprecated because of the confusion it can cause, and so names should not be reused, even in separate name spaces.

The example below illustrates a violation of this rule in which `value` is inadvertently used instead of `record.value`:

```
struct { int16_t key; int16_t value; } record;
int16_t value;  /* Rule violation - second use of value */

record.key = 1;
value = 0;      /* should have been record.value */
```

By contrast, the example below does not violate the rule because two member names are less likely to be confused:

```
struct device_q { struct device_q *next;   /* ... */ }
devices[N_DEVICES];
struct task_q { struct task_q *next;      /* ... */ }
tasks[N_TASKS];

devices[0].next = &devices[1];
tasks[0].next   = &tasks[1];
```

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

Regardless of scope, no identifier should be re-used across any files in the system. This rule incorporates the provisions of Rules 5.2, 5.3, 5.4, 5.5 and 5.6.

```
struct air_speed
{
   uint16_t speed;   /* knots */
} * x;
struct gnd_speed
{
   uint16_t speed;   /* mph                                    */
                     /* Not Compliant - speed is in different units */
} * y;
x->speed = y->speed;
```

Where an identifier name is used in a header file, and that header file is included in multiple source files, this rule is not violated. The use of a rigorous naming convention can support the implementation of this rule.

## 6.6 Types

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

[Implementation 14]

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

There are three distinct *char* types, (plain) *char*, *signed char* and *unsigned char*. *signed char* and *unsigned char* shall be used for numeric data and plain *char* shall be used for character data. The signedness of the plain *char* type is implementation defined and should not be relied upon.

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".

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.

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

The basic numerical types of *signed* and *unsigned* variants of *char*, *int*, *short*, *long* and *float*, *double* should not be used, but specific-length *typedefs* should be used. Rule 6.3 helps to clarify the size of the storage, but does not guarantee portability because of the asymmetric behaviour of integral promotion. See discussion of integral promotion — section 6.10. It is still important to understand the integer size of the implementation.

Programmers should be aware of the actual implementation of the *typedefs* under these definitions.

For example, the ISO (POSIX) *typedefs* as shown below are recommended and are used for all basic numerical and character types in this document. For a 32-bit integer machine, these are as follows:

```
typedef          char   char_t;
typedef signed   char   int8_t;
typedef signed   short  int16_t;
typedef signed   int    int32_t;
typedef signed   long   int64_t;
typedef unsigned char   uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned int    uint32_t;
typedef unsigned long   uint64_t;
typedef          float  float32_t;
typedef          double float64_t;
typedef long     double float128_t;
```

*typedefs* are not considered necessary in the specification of bit-field types.

**Rule 6.4 (required):** **Bit fields shall only be defined to be of type *unsigned int* or *signed int*.**

[Undefined 38; Implementation 29]

Using *int* is implementation defined because bit fields of type *int* can be either *signed* or *unsigned*. The use of *enum*, *short* or *char* types for bit fields is not allowed because the behaviour is undefined.

**Rule 6.5 (required):** **Bit fields of *signed* type shall be at least 2 bits long.**

A signed bit field of 1 bit length is not useful.

## 6.7 Constants

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

[Koenig 9]

Any integer constant beginning with a "0" (zero) is treated as octal. So there is a danger, for example, with writing fixed length constants. For example, the following array initialisation for 3-digit bus messages would not do as expected (052 is octal, i.e. 42 decimal):

```
code[1] = 109;   /* equivalent to decimal 109 */
code[2] = 100;   /* equivalent to decimal 100 */
code[3] = 052;   /* equivalent to decimal 42  */
code[4] = 071;   /* equivalent to decimal 57  */
```

Octal escape sequences can be problematic because the inadvertent introduction of a decimal digit ends the octal escape and introduces another character. The value of the first expression in the following example is implementation-defined because the character constant consists of two characters, "`\10`" and "`9`". The second character constant expression below contains the single character "`\100`". Its value will be implementation-defined if character 64 is not represented in the basic execution character set.

```
code[5] = '\109';   /* implementation-defined, two character constant */
code[6] = '\100';   /* set to 64, or implementation-defined           */
```

It is better not to use octal constants or escape sequences at all, and to check statically for any occurrences. 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.

## 6.8 Declarations and definitions

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

[Undefined 22, 23]

The use of prototypes enables the compiler to check the integrity of function definitions and calls. Without prototypes the compiler is not obliged to pick up certain errors in function calls (e.g. different number of arguments from the function body, mismatch in types of arguments between call and definition). Function interfaces have been shown to be a cause of considerable problems, and therefore this rule is considered very important.

The recommended method of implementing function prototypes for external functions is to declare the function (i.e. give the function prototype) in a header file, and then include the header file in all those code files that need the prototype (see Rule 8.8).

The provision of a prototype for a function with internal linkage is a good programming practice.

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

```
extern          x;           /* Non-compliant - implicit int type */
extern int16_t  x;           /* Compliant - explicit type         */
const           y;           /* Non-compliant - implicit int type */
const  int16_t  y;           /* Compliant - explicit type         */
static          foo(void);   /* Non-compliant - implicit type     */
static int16_t  foo(void);   /* Compliant - explicit type         */
```

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

[Undefined 24; Koenig 59–62]

The types of the parameters and return values in the prototype and the definition must match. This requires identical types including *typedef* names and qualifiers, and not just identical base types.

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

[Undefined 10]

The definition of compatible types is lengthy and complex (ISO/IEC 9899:1990 [2], sections 6.1.2.6, 6.5.2, 6.5.3 and 6.5.4 give full details). Two identical types are compatible but two compatible types need not be identical. For example, the following pairs of types are compatible:

```
signed int            int
char [5]              char []
unsigned short int    unsigned short
```

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

Header files should be used to declare objects, functions, *typedefs*, and macros. Header files shall not contain or produce definitions of objects or functions (or fragment of functions or objects) that occupy storage. This makes it clear that only C files contain executable source code and that header files only contain declarations. A "header file" is defined as any file that is included via the *#include* directive, regardless of name or suffix.

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

[Undefined 36]

Declaring functions at block scope may be confusing, and can lead to undefined behaviour.

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

The scope of objects shall be restricted to functions where possible. File scope shall only be used where objects need to have either internal or external linkage. Where objects are declared at file scope Rule 8.10 applies. It is considered good practice to avoid making identifiers global except where necessary. Whether objects are declared at the outermost or innermost block is largely a matter of style. "Accessing" means using the identifier to read from, write to, or take the address of the object.

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

[Koenig 66]

Normally this will mean declaring an external identifier in a header file, that will be included in any file where the identifier is defined or used. For example:

```
extern int16_t a;
```

in `featureX.h`, then to define `a`:

```
#include <featureX.h>
int16_t a = 0;
```

There may be one or there may be many header files in a project, but each external object or function shall only be declared in one header file.

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

[Undefined 44; Koenig 55, 63–65]

Behaviour is undefined if an identifier is used for which multiple definitions exist (in different files) or no definition exists at all. Multiple definitions in different files are not permitted even if the definitions are the same, and it is obviously serious if they are different, or initialise the identifier to different values.

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

[Koenig 56, 57]

If a variable is only to be used by functions within the same file then use *static*. Similarly if a function is only called from elsewhere within the same file, use *static*. Use of the *static* storage-class specifier will ensure that the identifier is only visible in the file in which it is declared and avoids any possibility of confusion with an identical identifier in another file or a library.

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

The *static* and *extern* storage class specifiers can be a source of confusion. It is good practice to apply the *static* keyword consistently to all declarations of objects and functions with internal linkage.

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

```
int array1[ 10 ];                 /* Compliant     */
extern int array2[ ];             /* Not compliant */
int array2[ ] = { 0, 10, 15 };    /* Compliant     */
```

Although it is possible to declare an array of incomplete type and access its elements, it is safer to do so when the size of the array may be explicitly determined.

## 6.9 Initialisation

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

[Undefined 41]

The intent of this rule is that all variables shall have been written to before they are read. This does not necessarily require initialisation at declaration.

Note that according to the ISO C standard, variables with static storage duration are automatically initialised to zero by default, unless explicitly initialised. In practice, many embedded environments do not implement this behaviour. Static storage duration is a property of all variables declared with the *static* storage class specifier, or with external linkage. Variables with automatic storage duration are not usually automatically initialised.

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

[Undefined 42]

ISO C requires initialiser lists for arrays, structures and union types to be enclosed in a single pair of braces (though the behaviour if this is not done is undefined). The rule given here goes further in requiring the use of additional braces to indicate nested structures. This forces the programmer to explicitly consider and demonstrate the order in which elements of complex data types are initialised (e.g. multi-dimensional arrays).

For example, below are two valid (in ISO C) ways of initialising the elements of a two dimensional array, but the first does not adhere to the rule:

```
int16_t y[3][2] = { 1, 2, 3, 4, 5, 6 };               /* not compliant */
int16_t y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };   /* compliant     */
```

A similar principle applies to structures, nested combinations of structures, arrays and other types.

Note also that all the elements of arrays or structures can be initialised (to zero or *NULL*) by giving an explicit initialiser for the first element only. If this method of initialisation is chosen then the first element should be initialised to zero (or *NULL*), and nested braces need not be used.

The ISO standard [2] contains extensive examples of initialisation.

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.

**Rule 9.3 (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.**

If an enumerator list is given with no explicit initialisation of members, then C allocates a sequence of integers starting at 0 for the first element and increasing by 1 for each subsequent element.

An explicit initialisation of the first element, as permitted by the above rule, forces the allocation of integers to start at the given value. When adopting this approach, it is essential to ensure that the initialisation value used is small enough that no subsequent value in the list will exceed the *int* storage used by enumeration constants.

Explicit initialisation of all items in the list, which is also permissible, prevents the mixing of automatic and manual allocation, which is error prone. However, it is then the responsibility of the programmer to ensure that all values are in the required range, and that values are not unintentionally duplicated.

```
enum colour { red=3, blue, green, yellow=5 };         /* non compliant */
   /* green and yellow represent the same value - this is duplication */
enum colour { red=3, blue=4, green=5, yellow=5 };         /* compliant */
   /* green and yellow represent the same value - this is duplication */
```

## 6.10 Arithmetic type conversions

### 6.10.1 Implicit and explicit type conversions

The C language allows the programmer considerable freedom and will allow conversions between different arithmetic types to be performed automatically. An explicit cast may be introduced for functional reasons, for example:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

The insertion of a cast for purposes of clarification is often helpful, but when taken to excess, the practice can lead to unreadable code. As demonstrated below, there are some implicit conversions that can safely be ignored and others that cannot.

### 6.10.2 Types of implicit conversion

There are three particular categories of implicit type conversion that need to be distinguished.

#### Integral promotion conversions

Integral promotion describes a process whereby arithmetic operations are always conducted on integer operands of type *int* or *long* (*signed* or *unsigned*). Operands of any other integer type, (*char*, *short*, *bit-field* and *enum*) are always converted to type *int* or *unsigned int* before an arithmetic operation. These types are referred to as *small integer types*.

The rules of integral promotion decree that in most arithmetic operations, an operand of a *small integer type* be converted to an *int* if an *int* is able to represent all values of the original type; otherwise the value is converted to *unsigned int*.

Notice that integral promotion:

- is only applied to small integer types
- is applied to operands of unary, binary and ternary operators
- is not applied to the operands of the logical operators `&&`, `||`, `!`
- is applied to the control expression of a switch statement.

Integral promotion is frequently confused with "balancing" of operands (described below). In fact, integral promotion takes place in unary operations and it takes place in binary operations where both operands are of the same type.

Because of integral promotion, the result of adding two objects of type *unsigned short* is always a value of type *signed int* or *unsigned int*; in fact, the addition is performed in that type. It is therefore possible for such an operation to derive a result whose value exceeds the size that could be accommodated in the original type of the operands. For example, if the size of an *int* is 32 bits, it is possible to multiply two objects of type *short* (16 bits) and derive a 32-bit result with no danger of overflow. On the other hand, if the size of an *int* is only 16 bits, the product of two 16-bit objects will only yield a 16-bit result and appropriate restrictions must be placed on the size of the operands.

Integral promotion also applies to unary operators. For example, the result of applying a bitwise negation operator (~) to an *unsigned char* operand is typically a negative value of type *signed int*.

Integral promotion is a fundamental inconsistency in the C language whereby the *small integer types* behave differently from *long* and *int* types. The use of *typedefs* is a practice that is encouraged in MISRA-C. However, because the behaviour of the various integer types is not consistent, it can be unsafe to ignore the underlying base types (see description on following pages) unless some restrictions are placed on the way in which expressions are constructed. It is the intention of the following rules that the effects of integral promotion should be neutralised in order to avoid these anomalies.

## Assigning conversions

Assigning conversions occur when:

- The type of an assignment expression is converted to the type of the assignment object.
- The type of an initialiser expression is converted to the type of the initialised object.
- The type of a function call argument is converted to the type of the formal parameter as declared in the function prototype.
- The type of the expression used in a return statement is converted to the type of the function as declared in the function prototype.
- The type of the constant expression in a *switch* case label is converted to the promoted type of the controlling expression. This conversion is performed only for the purposes of comparison.

In each case, the value of an arithmetic expression is unconditionally converted, where necessary, to another type.

## Balancing conversions

Balancing conversions are described in the ISO C standard under the term "Usual Arithmetic Conversions". This is a set of rules which provides a mechanism to yield a common type when two operands of a binary operator are balanced to a common type or the second and third arguments of the conditional operator (… `?` … `:` …) are balanced to a common type. Balancing conversions always involve two operands of different type; one and sometimes both operands will be subject to an implicit conversion.

The balancing rules are complicated by the process of integral promotion (described above) under which any operand of a *small integer type* is first promoted to type *int* or *unsigned int*. Integral promotion happens as part of the usual arithmetic conversions even when two operands are of identical type.

The operators explicitly associated with balancing conversions are:

- Multiplicative `*`, `/`, `%`
- Additive `+`, `-`
- Bitwise `&`, `^`, `|`
- The conditional operator (… `?` … `:` …)
- Relational operators `>`, `>=`, `<`, `<=`
- Equality operators `==`, `!=`

Most of these operators yield a result that is the type resulting from the balancing process. Relational and equality operators are the exception in that they yield a Boolean value result of type *int*.

Notice that the operands of the bitwise shift operators (<< and >>) are not balanced. The type of the result is the promoted type of the first operand; the second operand may be of any signed or unsigned integer type.

### 6.10.3 Dangerous type conversions

There are a number of potential dangers associated with type conversions that it is necessary to avoid:

- **Loss of value**: Conversion to a type where the magnitude of the value cannot be represented.
- **Loss of sign**: Conversion from a signed type to an unsigned type resulting in loss of sign.
- **Loss of precision**: Conversion from a floating type to an integer type with consequent loss of precision.

The only type conversions that can be guaranteed safe for all data values and all possible conforming implementations are:

- Conversion of an integral value to a wider type of the same signedness.
- Conversion of a floating type to a wider floating type.

Of course, in practice, if assumptions are about typical type sizes, it is possible to classify other type conversions as safe. In general, MISRA-C:2004 adopts the principle that it is wise to identify potentially dangerous type conversions by making the conversion explicit.

There are some other dangers in the area of type conversion that also need to be recognised. These are issues that arise from areas of misunderstanding and difficulty in the C language rather than because data values are not preserved.

- **Type widening in integral promotion**: The type in which integral expressions are evaluated depends on the type of the operands after any integral promotion. It is always possible to multiply two 8-bit values and access a 16-bit result if the magnitude requires it. It is sometimes, but not always, possible to multiply two 16-bit values and retrieve a 32-bit result. This is a dangerous inconsistency in the C language and in order to avoid confusion it is safer never to rely on the widening type afforded by integral promotion. Consider the following example:

```
uint16_t u16a = 40000;    /* unsigned short / unsigned int ?   */
uint16_t u16b = 30000;    /* unsigned short / unsigned int ?   */
uint32_t u32x;            /* unsigned int / unsigned long  ?   */

u32x = u16a + u16b;       /* u32x = 70000 or 4464 ?            */
```

  The expected result is presumably 70000, but the value assigned to `u` will in practice depend on the implemented size of an *int*. If the implemented size of an *int* is 32 bits, the addition will occur in 32-bit signed arithmetic and the correct value will be stored. If the implemented size of an *int* is only 16 bits, the addition will take place in 16-bit unsigned arithmetic, wraparound will occur and will yield the value 4464 (70000 % 65536). Wraparound in unsigned arithmetic is well defined and may even be intended; but there is potential for confusion.

- **Evaluation type confusion**: A similar problem arises from a common misconception among programmers that the type in which a calculation is conducted is influenced in some way by the type to which the result is assigned or converted. For example, in the following code the two 16-bit objects are added together in 16-bit arithmetic (unless promoted to 32-bit *int* by integral promotion), and the result is converted to type *uint32_t* on assignment.

  ```
  u32x = u16a + u16b;
  ```

  It is not unusual for programmers to be deceived into thinking that the addition is performed in 32-bit arithmetic — because of the type of `u32x`.

  Confusion of this nature is not confined to integer arithmetic or to implicit conversions. The following examples demonstrate some statements in which the result is well defined but the calculation may not be performed in the type that the programmer assumes.

  ```
  u32a = (uint32_t)(u16a * u16b);
  f64a = u16a / u16b;
  f32a = (float32_t)(u16a / u16b);
  f64a = f32a + f32b;
  f64a = (float64_t)(f32a + f32b);
  ```

- **Change of signedness in arithmetic operations**: Integral promotion will often result in two unsigned operands yielding a result of type *(signed) int*. For example, the addition of two 16-bit unsigned operands will yield a signed 32-bit result if *int* is 32 bits but an unsigned 16-bit result if *int* is 16 bits.
- **Change of signedness in bitwise operations**: Integral promotion can have some particularly unfortunate repercussions when bitwise operators are applied to small unsigned types. For example a bitwise complement operation on an operand of type *unsigned char* will generally yield a result of type *(signed) int* with a negative value. The operand is promoted to type *int* before the operation and the extra high order bits are set by the complement process. The number of extra bits, if any, is dependent on the size of an *int* and it is hazardous if the complement operation is followed by a right shift.

In order to avoid the perils associated with the issues described above, it is important to establish some principles to constrain the way in which expressions are constructed. Firstly definitions of some concepts are given.

### 6.10.4 Underlying type

The *type* of an expression refers to the type of the value obtained when the expression is evaluated. When two items of type *long* are added, the expression has type *long*. Most arithmetic operators derive a result whose type is dependent on the type of the operands. On the other hand, there are some operators that yield a Boolean result of type *int* regardless of the type of the operands. So, for example, when two items of type *long* are compared with a relational operator the expression has type *int*.

The term "underlying type" is defined as describing the type that would be obtained from evaluating an expression if it were not for the effects of integral promotion.

When two operands of type *int* are added the result is of type *int* and the expression may be said to have type *int*.

When two operands of type *unsigned char* are added the result is also (usually) of type *int* (because of integral promotion), but the **underlying** type of the expression is defined to be *unsigned char*.

The term "underlying type" is not known in the C standard or in other texts on the C language but is useful in describing some of the following rules. It describes a hypothetical departure from the C language in which integral promotion does not exist and the usual arithmetic conversions are applied consistently to all integer types. The concept is introduced because the effects of integral promotion are subtle and sometimes dangerous. Integral promotion is an unavoidable feature of the C language, but **the intention of these rules is that the effect of integral promotion should be neutralised by taking no advantage of the widening that occurs with small integer operands**.

Of course, the C standard does not explicitly define how small integer types would be balanced to a common type in the absence of integral promotion although it does establish the value-preserving principles.

When adding operands of type *int*, the programmer is obliged to ensure that the result of the operation will not exceed a value that can be represented in type *int*. If he fails to do so, overflow will occur and the results are undefined. It is the intention of the approach described here that the same principle should apply when small integer operands are added; the programmer should ensure that the result of adding two *unsigned chars* is representable in an *unsigned char*, even though integral promotion could give rise to evaluation in a larger type. In other words the limitations of the underlying type of an expression should be observed rather than the actual type.

## Underlying type of an integer constant expression

One unfortunate aspect of the C language is that it is not possible to define an integer constant with a *char* or *short* type. For example the value "5" can be expressed as a literal constant of type *int*, *unsigned int*, *long* or *unsigned long* by the addition of a suitable suffix; but no suffix is available to create a representation of the value in the various *char* or *short* types. This presents a difficulty when attempting to maintain type consistency in expressions. If it is desired to assign a value to an object of type *unsigned char*, then either an implicit type conversion from an integer type must be tolerated or a cast must be introduced. Many would argue that to use a cast in such circumstances serves only to reduce readability.

The same problem exists when constants are required in initialisers, function arguments or arithmetic expressions. However the problem is largely a philosophical one associated with the aspiration to observe principles of strong typing.

One way of addressing this problem is to **imagine** that an integer constant, an enumeration constant, a character constant or an integer constant expression has a type appropriate to its magnitude. This objective can be achieved by extending the concept of underlying type to integer constants and imagining that, where possible, the literal constant has been derived by integral promotion from an imaginary constant with a smaller underlying type.

The underlying type of an integer constant expression is therefore defined as follows:

1. If the actual type of the expression is *(signed) int*, the underlying type is defined to be the smallest signed integer type which is capable of representing its value.
2. If the actual type of the expression is *unsigned int*, the underlying type is defined to be the smallest unsigned integer type that is capable of representing its value.

3. In all other circumstances, the underlying type of the expression is defined to be the same as its actual type.

In a conventional architecture, the underlying type of an integer constant expression will be determined according to its magnitude and signedness as follows:

### Unsigned values

| | | | |
|---|---|---|---|
| 0U | to | 255U | 8 bit unsigned |
| 256U | to | 65535U | 16 bit unsigned |
| 65536U | to | 4294967295U | 32 bit unsigned |

### Signed values

| | | | |
|---|---|---|---|
| -2147483648 | to | -32769 | 32 bit signed |
| -32768 | to | -129 | 16 bit signed |
| -128 | to | 127 | 8 bit signed |
| 128 | to | 32767 | 16 bit signed |
| 32768 | to | 2147483647 | 32 bit signed |

Notice that underlying type is an artificial concept. It does not in any way influence the type of evaluation that is actually performed. The concept has been developed simply as a way of defining a safe framework in which to construct arithmetic expressions.

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*.

## 6.10.5 Complex expressions

The type conversion rules described in the following paragraphs refer in some places to the notion of a "complex expression". The term "complex expression" is defined to mean any expression that is not:

- a constant expression
- an *lvalue* (i.e. an object)
- the return value of a function

The conversions that may be applied to complex expressions are restricted in order to avoid some of the dangers outlined above. Specifically it is required that a sequence of arithmetic operations in an expression should be conducted in the same type.

The following expressions are complex:

```
s8a + s8b
~u16a
u16a >> 2
foo(2) + u8a
*ppc + 1
++u8a
```

The following expressions are not complex, even though some contain complex sub-expressions:

```
pc[u8a]
foo(u8a + u8b)
**ppuc
*(ppc + 1)
pcbuf[s16a * 2]
```

### 6.10.6 Implicit type conversions

**Rule 10.1 (required):** **The value of an expression of integer type shall not be implicitly converted to a different underlying type if:**
**(a) it is not a conversion to a wider integer type of the same signedness, or**
**(b) the expression is complex, or**
**(c) the expression is not constant and is a function argument, or**
**(d) the expression is not constant and is a return expression**

**Rule 10.2 (required):** **The value of an expression of floating type shall not be implicitly converted to a different type if:**
**(a) it is not a conversion to a wider floating type, or**
**(b) the expression is complex, or**
**(c) the expression is a function argument, or**
**(d) the expression is a return expression**

Notice also that in describing integer conversions, the concern is always with underlying type rather than actual type.

These two rules broadly encapsulate the following principles:

- No implicit conversions between signed and unsigned types
- No implicit conversions between integer and floating types
- No implicit conversions from wider to narrower types
- No implicit conversions of function arguments
- No implicit conversions of function return expressions
- No implicit conversions of complex expressions

The intention when restricting implicit conversion of complex expressions is to require that in a sequence of arithmetic operations within an expression, all operations should be conducted in exactly the same arithmetic type. Notice that this does not imply that all operands in an expression are of the same type.

The expression `u32a + u16b + u16c` is compliant — both additions will **notionally** be performed in type U32.

The expression `u16a + u16b + u32c` is not compliant — the first addition is **notionally** performed in type U16 and the second in type U32.

The word “notionally” is used because, in practice, the type in which arithmetic will be conducted will depend on the implemented size of an *int*. By observing the principle whereby all operations are performed in a consistent (underlying) type, it is possible to avoid programmer confusion and some of the dangers associated with integral promotion.

```
extern void foo1(uint8_t x);
int16_t t1(void)
{
   ...
   foo1(u8a);                                 /* compliant     */
   foo1(u8a + u8b);                           /* compliant     */
   foo1(s8a);                                 /* not compliant */
   foo1(u16a);                                /* not compliant */
   foo1(2);                                   /* not compliant */
   foo1(2U);                                  /* compliant     */
   foo1((uint8_t)2);                          /* compliant     */
   ... s8a + u8a                              /* not compliant */
   ... s8a + (int8_t)u8a                      /* compliant     */
   s8b  = u8a;                                /* not compliant */
   ... u8a + 5                                /* not compliant */
   ... u8a + 5U                               /* compliant     */
   ... u8a + (uint8_t)5                       /* compliant     */
   u8a  = u16a;                               /* not compliant */
   u8a  = (uint8_t)u16a;                      /* compliant     */
   u8a  = 5UL;                                /* not compliant */
   ... u8a + 10UL                             /* compliant     */
   u8a  = 5U;                                 /* compliant     */
   ... u8a + 3                                /* not compliant */
   ... u8a >> 3                               /* compliant     */
   ... u8a >> 3U                              /* compliant     */
   pca  = "P";                                /* compliant     */
   ... s32a + 80000                           /* compliant     */
   ... s32a + 80000L                          /* compliant     */
   f32a = f64a;                               /* not compliant */
   f32a = 2.5;                                /* not compliant -
                                                 unsuffixed floating
                                                 constants are of type
                                                 double        */
   u8a  = u8b + u8c;                          /* compliant     */
   s16a = u8b + u8b;                          /* not compliant */
   s32a = u8b + u8c;                          /* not compliant */
   f32a = 2.5F;                               /* compliant     */
   u8a  = f32a;                               /* not compliant */
   s32a = 1.0;                                /* not compliant */
   s32a = u8b + u8c;                          /* not compliant */
   f32a = 2.5F;                               /* compliant     */
   u8a  = f32a;                               /* not compliant */
   s32a = 1.0;                                /* not compliant */
   f32a = 1;                                  /* not compliant */
   f32a = s16a;                               /* not compliant */
```

```
      ... f32a + 1                                /* not compliant */
      ... f64a * s32a                             /* not compliant */
      ...
      return (s32a);                              /* not compliant */
      ...
      return (s16a);                              /* compliant     */
      ...
      return (20000);                             /* compliant     */
      ...
      return (20000L);                            /* not compliant */
      ...
      return (s8a);                               /* not compliant */
      ...
      return (u16a);                              /* not compliant */
   }
   int16_t foo2(void)
   {
      ...
      ... (u16a + u16b) + u32a                    /* not compliant */
      ... s32a + s8a + s8b                        /* compliant     */
      ... s8a + s8b + s32a                        /* not compliant */
      f64a = f32a + f32b;                         /* not compliant */
      f64a = f64b + f32a;                         /* compliant     */
      f64a = s32a / s32b;                         /* not compliant */
      u32a = u16a + u16a;                         /* not compliant */
      s16a = s8a;                                 /* compliant     */
      s16a = s16b + 20000;                        /* compliant     */
      s32a = s16a + 20000;                        /* not compliant */
      s32a = s16a + (int32_t)20000;               /* compliant     */
      u16a = u16b + u8a;                          /* compliant     */
      foo1(u16a);                                 /* not compliant */
      foo1(u8a + u8b);                            /* compliant     */
      ...
      return s16a;                                /* compliant     */
      ...
      return s8a;                                 /* not compliant */
   }
```

### 6.10.7 Explicit conversions (casts)

**Rule 10.3 (required):** **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.**

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

If a cast is to be used on any complex expression, the type of cast that may be applied is severely restricted. As explained in section 6.10, conversions on complex expressions are often a source of confusion and it is therefore wise to be cautious. In order to comply with these rules, it may be necessary to use a temporary variable and introduce an extra statement.

```
... (float32_t)(f64a + f64b)                    /* compliant     */
... (float64_t)(f32a + f32b)                    /* not compliant */
... (float64_t)f32a                             /* compliant     */
... (float64_t)(s32a / s32b)                    /* not compliant */
... (float64_t)(s32a > s32b)                    /* not compliant */
... (float64_t)s32a / (float32_t)s32b           /* compliant     */
... (uint32_t)(u16a + u16b)                     /* not compliant */
... (uint32_t)u16a + u16b                       /* compliant     */
... (uint32_t)u16a + (uint32_t)u16b             /* compliant     */
... (int16_t)(s32a - 12345)                     /* compliant     */
... (uint8_t)(u16a * u16b)                      /* compliant     */
... (uint16_t)(u8a * u8b)                       /* not compliant */
... (int16_t)(s32a * s32b)                      /* compliant     */
... (int32_t)(s16a * s16b)                      /* not compliant */
... (uint16_t)(f64a + f64b)                     /* not compliant */
... (float32_t)(u16a + u16b)                    /* not compliant */
... (float64_t)foo1(u16a + u16b)                /* compliant     */
... (int32_t)buf16a[u16a + u16b]                /* compliant     */
```

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

When these operators (~ and <<) are applied to *small integer types* (*unsigned char* or *unsigned short*), the operations are preceded by integral promotion, and the result may contain high order bits which have not been anticipated. For example:

```
uint8_t port = 0x5aU;
uint8_t result_8;
uint16_t result_16;
uint16_t mode;

result_8 = (~port) >> 4;                        /* not compliant */
```

`~port` is 0xffa5 on a 16-bit machine but 0xffffffa5 on a 32-bit machine. In either case, the value of `result` is 0xfa, but 0x0a may have been expected. This danger is avoided by inclusion of the cast as shown below:

```
result_8 = ((uint8_t)(~port)) >> 4 ;            /* compliant */
result_16 = ((uint16_t)(~(uint16_t)port)) >> 4 ; /* compliant */
```

A similar problem exists when the << operator is used on *small integer types* and high order bits are retained. For example:

```
result_16 = ((port << 4) & mode) >> 6;          /* not compliant */
```

The value in `result_16` will depend on the implemented size of an *int*. Addition of a cast avoids any ambiguity.

```
result_16 = ((uint16_t)((uint16_t)port << 4) & mode) >> 6;
                                                /* compliant */
```

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.

### 6.10.8 Integer suffixes

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

The type of an integer constant is a potential source of confusion, because it is dependent on a complex combination of factors including:

- The magnitude of the constant
- The **implemented** sizes of the integer types
- The presence of any suffixes
- The number base in which the value is expressed (i.e. decimal, octal or hexadecimal).

For example, the integer constant "40000" is of type *int* in a 32-bit environment but of type *long* in a 16-bit environment. The value 0x8000 is of type *unsigned int* in a 16-bit environment, but of type *(signed) int* in a 32-bit environment.

Note the following:

- Any value with a "U" suffix is of *unsigned* type
- An unsuffixed decimal value less than $2^{31}$ is of *signed* type

But:

- An unsuffixed hexadecimal value greater than or equal to $2^{15}$ may be of *signed* or *unsigned* type
- An unsuffixed decimal value greater than or equal to $2^{31}$ may be of *signed* or *unsigned* type

Signedness of constants should be explicit. Consistent signedness is an important principle in constructing well formed expressions. If a constant is of an *unsigned* type, it is helpful to avoid ambiguity by applying a "U" suffix. When applied to larger values, the suffix may be redundant (in the sense that it does not influence the type of the constant); however its presence is a valuable contribution towards clarity.

## 6.11 Pointer type conversions

Pointer types can be classified as follows:

- Pointer to object
- Pointer to function
- Pointer to *void*
- The null pointer constant (the value 0 cast to type *void* *)

Conversions involving pointer types require an explicit cast except when:

- The conversion is between a pointer to object and a pointer to *void* and the destination type carries all the type qualifiers of the source type.
- A null pointer constant (*void* *) is converted automatically to a particular pointer type when it is assigned to or compared for equality with any type of pointer.

Only certain types of pointer conversion are defined in C and the behaviour of some conversions is implementation defined.

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

[Undefined 27, 28]

Conversion of a function pointer to a different type of pointer results in undefined behaviour. This means that a function pointer can be converted to or from an integral type. No other conversions involving function pointers are permitted.

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

[Undefined 29]

Such conversions are undefined. This rule 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.

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

[Implementation 24]

The size of integer that is required when a pointer is converted to an integer is implementation defined. Casting between a pointer and an integer type should be avoided where possible, but may be unavoidable when addressing memory mapped registers or other hardware specific features.

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

Conversions of this type may be invalid if the new pointer type requires a stricter alignment.

```
uint8_t  * p1;
uint32_t * p2;
p2 = (uint32_t *)p1;   /* Incompatible alignment ? */
```

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

[Undefined 39, 40]

Any attempt to remove the qualification associated with the addressed type by using casting is a violation of the principle of type qualification. Notice that the qualification referred to here is not the same as any qualification that may be applied to the pointer itself.

```
uint16_t                    x;
uint16_t * const            cpi = &x;  /* const pointer                */
uint16_t * const          * pcpi;      /* pointer to const pointer    */
const uint16_t *          * ppci;      /* pointer to pointer to const */
uint16_t *                * ppi;
const uint16_t            * pci;       /* pointer to const            */
volatile uint16_t         * pvi;       /* pointer to volatile         */
uint16_t                  * pi;
...
pi = cpi;                              /* Compliant - no conversion
                                                   no cast required */
pi = (uint16_t *)pci;                  /* Not compliant               */
pi = (uint16_t *)pvi;                  /* Not compliant               */
ppi = (uint16_t * *)pcpi;              /* Not compliant               */
ppi = (uint16_t * *)ppci;              /* Not compliant               */
```

## 6.12 Expressions

**Rule 12.1 (advisory): Limited dependence should be placed on C's operator precedence rules in expressions.**

In addition to the use of parentheses to override default operator precedence, parentheses should also be used to emphasise it. It is easy to make a mistake with the rather complicated precedence rules of C, and this approach helps to avoid such errors, and helps to make the code easier to read. However, do not add too many parentheses so as to clutter the code and make it unreadable.

The following guidelines are suggested in deciding when parentheses are required:

- no parentheses are required for the right-hand operand of an assignment operator unless the right-hand side itself contains an assignment expression:

```
x = a + b;                          /* acceptable                */
x = (a + b);                        /* () not required           */
```

- no parentheses are required for the operand of a unary operator:

```
x = a * -1;                         /* acceptable                */
x = a * (-1);                       /* () not required           */
```

- otherwise, the operands of binary and ternary operators shall be *cast-expressions* (see section 6.3.4 of ISO/IEC 9899:1990 [2]) unless all the operators in the expression are the same.

```
x = a + b + c;                      /* acceptable, but care needed */
x = f (a + b, c);                   /* no () required for a + b    */
x = (a == b) ? a : (a - b);
if (a && b && c)                    /* acceptable                  */
x = (a + b) - (c + d);
x = (a * 3) + c + d;
x = (uint16_t) a + b;               /* no need for ((uint16_t) a)  */
```

- even if all operators are the same, parentheses may be used to control the order of operation. Some operators (e.g. addition and multiplication) that are associative in algebra are not necessarily associative in C. Similarly, integer operations involving mixed types (prohibited by several rules) may produce different results because of the integral promotions. The following example written for a 16-bit implementation demonstrates that addition is not associative and that it is important to be clear about the structure of an expression:

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

d = (a + b) + c;    /* d is 9; a + b wraps modulo 65536 */
d = a + (b + c);    /* d is 65545                       */
/* this example also deviates from several other rules */
```

Note that Rule 12.5 is a special case of this rule applicable solely to the logical operators, `&&` and `||`.

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

[Unspecified 7–9; Undefined 18]

Apart from a few operators (notably the function call operator `()`, `&&`, `||`, `?:` and `,` (comma)) the order in which sub-expressions are evaluated is unspecified and can vary. This means that no reliance can be placed on the order of evaluation of sub-expressions, and in particular no reliance can be placed on the order in which side effects occur. Those points in the evaluation of an expression at which all previous side-effects can be guaranteed to have taken place are called “sequence points”. Sequence points and side effects are described in sections 5.1.2.3, 6.3 and 6.6 of ISO/IEC 9899:1990 [2].

Note that the order of evaluation problem is not solved by the use of parentheses as this is not a precedence issue.

The following notes give some guidance on how dependence on order of evaluation may occur, and therefore may assist in adopting the rule.

- ***increment or decrement operators***

  As an example of what can go wrong, consider

  ```
  x = b[i] + i++;
  ```

  This will give different results depending on whether `b[i]` is evaluated before `i++` or vice versa. The problem could be avoided by putting the increment operation in a separate statement. The example would then become:

  ```
  x = b[i] + i;
  i++;
  ```

- ***function arguments***

  The order of evaluation of function arguments is unspecified.

  ```
  x = func( i++, i );
  ```

  This will give different results depending on which of the function's two parameters is evaluated first.

- ***function pointers***

  If a function is called via a function pointer there shall be no dependence on the order in which function-designator and function arguments are evaluated.

  ```
  p->task_start_fn (p++);
  ```

- ***function calls***

  Functions may have additional effects when they are called (e.g. modifying some global data). Dependence on order of evaluation could be avoided by invoking the function prior to the expression that uses it, making use of a temporary variable for the value.

  For example

  ```
  x = f(a) + g(a);
  ```

  could be written as

  ```
  x = f(a);
  x += g(a);
  ```

  As an example of what can go wrong, consider an expression to get two values off a stack, subtract the second from the first, and push the result back on the stack:

  ```
  push( pop() - pop() );
  ```

  This will give different results depending on which of the `pop()` function calls is evaluated first (because `pop()` has side-effects).

- ***nested assignment statements***

  Assignments nested within expressions cause additional side effects. The best way to avoid any chance of this leading to a dependence on order of evaluation is to not embed assignments within expressions.

  For example, the following is not recommended:

  ```
  x = y = y = z / 3 ;
  x = y = y++;
  ```

- ***accessing a volatile***

  The *volatile* type qualifier is provided in C to denote objects whose value can change independently of the execution of the program (for example an input register). If an object of *volatile* qualified type is accessed this may change its value. C compilers will not optimise out reads of a volatile. In addition, as far as a C program is concerned, a read of a volatile has a side-effect (changing the value of the volatile).

It will usually be necessary to access volatile data as part of an expression, which then means there may be dependence on order of evaluation. Where possible though, it is recommended that volatiles only be accessed in simple assignment statements, such as the following:

```
volatile uint16_t v;
/* ... */
x = v;
```

The rule addresses the order of evaluation problem with side effects. Note that there may also be an issue with the number of times a sub-expression is evaluated, which is not covered by this rule. This can be a problem with function invocations where the function is implemented as a macro. For example, consider the following function-like macro and its invocation:

```
#define MAX(a, b)  ( ((a) > (b)) ? (a) : (b) )
/* ... */
z = MAX( i++, j );
```

The definition evaluates the first parameter twice if `a > b` but only once if `a` ≤ `b`. The macro invocation may thus increment `i` either once or twice, depending on the values of `i` and `j`.

It should be noted that magnitude-dependent effects, such as those due to floating-point rounding, are also not addressed by this rule. Although the order in which side-effects occur is undefined, the result of an operation is otherwise well-defined and is controlled by the structure of the expression. In the following example, `f1` and `f2` are floating-point variables; `F3`, `F4` and `F5` denote expressions with floating-point types.

```
f1 = F3 + (F4 + F5);
f2 = (F3 + F4) + F5;
```

The addition operations are, or at least appear to be, performed in the order determined by the position of the parentheses, i.e. firstly `F4` is added to `F5` then secondly `F3` is added to give the value of `f1`. Provided that `F3`, `F4` and `F5` contain no side-effects, their values are independent of the order in which they are evaluated. However, the values assigned to `f1` and `f2` are not guaranteed to be the same because floating-point rounding following the addition operations will depend on the values being added.

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

A possible programming error in C is to apply the *sizeof* operator to an expression and expect the expression to be evaluated. However the expression is not evaluated: *sizeof* only acts on the type of the expression. To avoid this error, *sizeof* shall not be used on expressions that contain side effects, as the side effects will not occur. *sizeof()* shall only be applied to an operand which is a type or an object. This may include volatile objects. For example:

```
int32_t i;
int32_t j;
j = sizeof(i = 1234);
              /* j is set to the sizeof the type of i which is an int */
              /* i is not set to 1234.                                */
```

**Rule 12.4 (required): The right-hand operand of a logical `&&` 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++ ) )        /* Not compliant                */
if ( ishigh && ( x == f(x) ) )       /* Only acceptable if f(x) is
                                        known to have no side effects */
```

The operations that cause side effects are described in section 5.1.2.3 of ISO/IEC 9899:1990 [2] as accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations, which cause changes in the state of the execution environment of the calling function.

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

"Primary expressions" are defined in ISO/IEC 9899:1990 [2], 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. Where an expression consists of either a sequence of only logical `&&` or a sequence of only logical `||`, extra parentheses are not required.

For example, write:

```
if ( ( x == 0 ) && ishigh )              /* make x == 0 primary      */
if ( x || y || z )                       /* exception allowed,
                                            if x, y and z are Boolean */
if ( x || ( y && z ) )                   /* make y && z primary      */
if ( x && ( !y ) )                       /* make !y primary          */
if ( ( is_odd (y) ) && x )               /* make call primary        */
```

Where an expression consists of either a sequence of only logical `&&` or a sequence of only logical `||`, extra parentheses are not required.

```
if ( ( x > c1) && (y > c2) && (z > c3) )   /* Compliant               */
if ( ( x > c1) && (y > c2) || (z > c3) )   /* not compliant           */
if ( ( x > c1) && ((y > c2) || (z > c3)) ) /* Compliant extra () used */
```

Note that this rule is a special case of Rule 12.1.

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

[Koenig 48]

The logical operators `&&`, `||` and `!` can be easily confused with the bitwise operators `&`, `|` and `~`. See "Boolean Expressions" in the glossary.

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

[Implementation 17–19]

Bitwise operations (`~`, `<<`, `<<=`, `>>`, `>>=`, `&`, `&=`, `^`, `^=`, `|` and `|=`) are not normally meaningful on signed integers. Problems can arise if, for example, a right shift moves the sign bit into the number, or a left shift moves a numeric bit into the sign bit.

See section 6.10 for a description of underlying type.

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

[Undefined 32]

If, for example, the left-hand operand of a left-shift or right-shift is a 16-bit integer, then it is important to ensure that this is shifted only by a number between 0 and 15 inclusive.

See section 6.10 for a description of underlying type.

There are various ways of ensuring this rule is followed. The simplest is for the right-hand operand to be a constant (whose value can then be statically checked). Use of an unsigned integer type will ensure that the operand is non-negative, so then only the upper limit needs to be checked (dynamically at run time or by review). Otherwise both limits will need to be checked.

```
    u8a  = (uint8_t) (u8a << 7);              /* compliant     */
    u8a  = (uint8_t) (u8a << 9);              /* not compliant */
    u16a = (uint16_t)((uint16_t) u8a << 9);   /* compliant     */
```

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

Applying the unary minus operator to an expression of type *unsigned int* or *unsigned long* generates a result of type *unsigned int* or *unsigned long* respectively and is not a meaningful operation. Applying unary minus to an operand of smaller unsigned integer type may generate a meaningful signed result due to integral promotion, but this is not good practice.

See section 6.10 for a description of underlying type.

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

Use of the comma operator is generally detrimental to the readability of code, and the same effect can be achieved by other means.

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

Because unsigned integer expressions do not strictly overflow, but instead wrap around in a modular way, any constant unsigned integer expressions which in effect "overflow" will not be detected by the compiler. Although there may be good reasons at run-time to rely on the modular arithmetic provided by unsigned integer types, the reasons for using it at compile-time to evaluate a constant expression are less obvious. Any instance of an unsigned integer constant expression wrapping around is therefore likely to indicate a programming error.

This rule applies equally to all phases of the translation process. Constant expressions that the compiler chooses to evaluate at compile time are evaluated in such a way that the results are identical to those that would be obtained by evaluation on the target with the exception of those appearing in conditional preprocessing directives. For such directives, the usual rules of arithmetic (see section 6.4 of ISO/IEC 9899:1990 [2]) apply **but** the *int* and *unsigned int* types behave instead as if they were *long* and *unsigned long* respectively.

For example, on a machine with a 16-bit *int* type and a 32-bit *long* type:

```
#define START 0x8000
#define END   0xFFFF
#define LEN   0x8000

#if ((START + LEN) > END)
   #error Buffer Overrun
             /* OK because START and LEN are unsigned long            */
#endif

#if (((END - START) - LEN) < 0)
   #error Buffer Overrun
             /* Not OK: subtraction result wraps around to 0xFFFFFFFF */
#endif

/* contrast the above START + LEN with the following */

if ((START + LEN) > END)
{
   error ("Buffer overrun");
             /* Not OK: START + LEN wraps around to 0x0000
                due to unsigned int arithmetic                         */
}
```

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

[Unspecified 6; Implementation 20]

The storage layout used for floating-point values may vary from one compiler to another, and therefore no floating-point manipulations shall be made which rely directly on the way the values are stored. The in-built operators and functions, which hide the storage details from the programmer, should be used.

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

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. The use of increment and decrement operators in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code.
- It introduces additional side effects into a statement with the potential for undefined behaviour.

It is safer to use these operations in isolation from any other arithmetic operators.

For example a statement such as the following is not compliant:

```
u8a = ++u8b + u8c--;    /* Not compliant */
```

The following sequence is clearer and therefore safer:

```
++u8b;
u8a = u8b + u8c;
u8c--;
```

## 6.13 Control statement expressions

**Rule 13.1 (required): Assignment operators shall not be used in expressions that yield a Boolean value.**

[Koenig 6]

No assignments are permitted in any expression which is considered to have a Boolean value. This precludes the use of both simple and compound assignment operators in the operands of a Boolean-valued expression. However, it does not preclude assigning a Boolean value to a variable.

If assignments are required in the operands of a Boolean-valued expression then they must be performed separately outside of those operands. This helps to avoid getting "=" and "==" confused, and assists the static detection of mistakes.

See "Boolean Expressions" in the glossary.

For example write:

```
x = y;
if ( x != 0 )
{
   foo();
}
```

and not:

```
if ( ( x = y ) != 0 )    /* Boolean by context */
{
   foo();
}
```

or even worse:

```
if ( x = y )
{
   foo();
}
```

**Rule 13.2 (advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.**

Where a data value is to be tested against zero then the test should be made explicit. The exception to this rule is when data represents a Boolean value, even though in C this will in practice be an integer. This rule is in the interests of clarity, and makes clear the distinction between integers and logical values.

For example, if `x` is an integer, then:

```
if ( x != 0 )  /* Correct way of testing x is non-zero               */
if ( y )       /* Not compliant, unless y is effectively Boolean data
                  (e.g. a flag)                                       */
```

See "Boolean Expressions" in the glossary.

**Rule 13.3 (required): Floating-point expressions shall not be tested for equality or inequality.**

The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true even when they are expected to. In addition, the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another. For example the result of the test in the following code is unpredictable:

```
float32_t x, y;
/* some calculations in here       */
if ( x == y )      /* not compliant */
   { /* ... */ }
if ( x == 0.0f)    /* not compliant */
```

An indirect test is equally problematic and is also forbidden by this rule, for example:

```
if ( ( x <= y ) && ( x >= y ) )
   { /* ... */ }
```

The recommended method for achieving deterministic floating-point comparisons is to write a library that implements the comparison operations. The library should take into account the floating-point granularity (*FLT_EPSILON*) and the magnitude of the numbers being compared. See also Rule 13.4 and Rule 20.3.

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

The controlling expression may include a loop counter, whose value is tested to determine termination of the loop. Floating-point variables shall not be used for this purpose. Rounding and truncation errors can be propagated through the iterations of the loop, causing significant inaccuracies in the loop variable, and possibly giving unexpected results when the test is performed. For example the number of times the loop is performed may vary from one implementation to another, and may be unpredictable. See also Rule 13.3.

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

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

| | |
|---|---|
| **First expression** | Initialising the loop counter (`i` in the following example) |
| **Second expression** | Shall include testing the loop counter (`i`), and optionally other loop control variables (`flag`) |
| **Third expression** | Increment or decrement of the loop counter (`i`) |

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.

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

Loop counters shall not be modified in the body of the loop. However other loop control variables representing logical values may be modified in the loop, for example a flag to indicate that something has been completed, which is then tested in the *for* statement.

```
flag = 1;
for ( i = 0; (i < 5) && (flag == 1); i++ )
{
    /* ... */
    flag = 0;    /* Compliant - allows early termination of loop */
    i = i + 3;   /* Not compliant - altering the loop counter    */
}
```

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

If a Boolean operator yields a result that can be proven to be always “true” or always “false”, it is highly likely that there is a programming error.

```
enum ec {RED, BLUE, GREEN} col;
...
if (u16a < 0)                  /* Not compliant - u16a is always >= 0 */
...
if (u16a <= 0xffff)            /* Not compliant - always true         */
...
if (s8a < 130)                 /* Not compliant - always true         */
...
if ((s8a < 10) && (s8a > 20))  /* Not compliant - always false        */
...

if ((s8a < 10) || (s8a > 5))   /* Not compliant - always true         */
...
if (col <= GREEN)              /* Not compliant - always true         */
...
if (s8a > 10)
{
   if (s8a > 5)                /* Not compliant - s8a is not volatile */
   {
   }
}
```

## 6.14 Control flow

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

This rule refers to code which cannot under any circumstances be reached, and which can be identified as such at compile time. Code that can be reached but may never be executed is excluded from the rule (e.g. defensive programming code).

A portion of code is unreachable if there is no control flow path from the relevant entry point to that code. For example, unlabelled code following an unconditional control transfer is unreachable:

```
switch (event)
{
   case E_wakeup:
      do_wakeup();
      break;       /* unconditional control transfer   */
      do_more();   /* Not compliant - unreachable code */
      /* ... */
   default:
      /* ... */
      break;
}
```

A whole function will be unreachable if there is no means by which it can be called.

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

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

Any statement (other than a null statement) which has no side-effect and does not result in a change of control flow will normally indicate a programming error, and therefore a static check for such statements shall be performed. For example, the following statements do not necessarily have side effects when executed:

```
/* assume uint16_t x;
   and uint16_t i; */
...
x >= 3u;    /* not compliant: x is compared to 3,
               and the answer is discarded */
```

Note that "null statement" and "side effect" are defined in ISO/IEC 9899:1990 [2] sections 6.6.3 and 5.1.2.3 respectively.

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

Null statements should not normally be deliberately included, but where they are used they shall appear on a line by themselves. White-space characters may precede the null statement to preserve indentation. If a comment follows the null statement then at least one white-space character shall separate the null statement from the comment. The use of a white-space character to separate the null statement from any following comment is required because it provides an important visual cue to reviewers. Following this rule enables a static checking tool to warn of null statements appearing on a line with other text, which would normally indicate a programming error. For example:

```
while ( ( port & 0x80 ) == 0 )
{
   ; /* wait for pin - Compliant */
   /* wait for pin */ ; /* Not compliant, comment before ; */
   ;/* wait for pin - Not compliant, no white-space char after ; */
}
```

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

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

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

These rules are in the interests of good structured programming. One *break* statement is allowed in a loop since this allows, for example, for dual outcome loops or for optimal coding.

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

[IEC 61508 Part 3 Table B.9]

This is required by IEC 61508, under good programming style.

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

The statement that forms the body of a *switch* statement or a *while*, *do … while* or *for* loop, shall be a compound statement (enclosed within braces), even if that compound statement contains a single statement.

For example:

```
for (i = 0; i < N_ELEMENTS; ++i)
{
   buffer[i] = 0;        /* Even a single statement must be in braces */
}

while ( new_data_available )
   process_data ();     /* Incorrectly not enclosed in braces         */
   service_watchdog (); /* Added later but, despite the appearance
                          (from the indent) it is actually not part of
                          the body of the while statement, and is
                          executed only after the loop has terminated */
```

Note that the layout for compound statements and their enclosing braces should be determined from the style guidelines. The above is just an example.

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

[Koenig 24]

For example:

```
if ( test1 )
{
 x = 1;              /* Even a single statement must be in braces     */
}
else if ( test2 )    /* No need for braces in else if                 */
{
 x = 0;              /* Single statement must be in braces            */
}
else
    x = 3;           /* This was (incorrectly) not enclosed in braces */
    y = 2;           /* This line was added later but, despite the
                        appearance (from the indent), it is actually
                        not part of the else, and is executed
                        unconditionally                               */
```

Note that the layout for compound statements and their enclosing braces should be determined from the style guidelines. The above is just an example.

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

This rule applies whenever an *if* statement is followed by one or more *else if* statements; the final *else if* shall be followed by an *else* statement. In the case of a simple *if* statement then the *else* statement need not be included.

The requirement for a final *else* statement is defensive programming. The *else* statement shall either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final default clause in a *switch* statement (15.3).

For example this code is a simple *if* statement:

```
if ( x < 0 )
{
   log_error(3);
   x = 0;
}                 /* else not needed */
```

whereas the following code demonstrates an *if*, *else if* construct

```
if ( x < 0 )
{
    log_error(3);
    x = 0;
}
else if ( y < 0 )
{
    x = 3;
}

else                 /* this else clause is required, even if the     */
{                    /* programmer expects this will never be reached */
   /* no change in value of x */
}
```

## 6.15 Switch statements

**Rule 15.0 (required): The MISRA C *switch* syntax shall be used.**

The syntax for the *switch* statement in C is weak, allowing complex, unstructured behaviour. The following text describes the syntax for *switch* statements as defined by MISRA-C and is normative. This, and the associated rules, enforce a simple and consistent structure on to the *switch* statement.

The following syntax rules are additional to the C standard syntax rules. All syntax rules not defined below are as defined in the C standard.

switch-statement :
  **switch (** expression **) {** case-label-clause-list default-label-clause**}**

case-label-clause-list:
  case-label case-clause$_{opt}$
  case-label-clause-list case-label case-clause$_{opt}$

case-label:
  **case** constant-expression **:**

case-clause:
  statement-list$_{opt}$ **break ;**
  { declaration-list$_{opt}$ statement-list$_{opt}$ **break ; }**

default-label-clause **:**
  default-label default-clause

default-label:
  **default :**

default-clause:
  case-clause

and

statement :
  /* labelled_statement removed */
  compound_statement
  expression_statement
  selection_statement
  iteration_statement
  /* jump_statement removed or just restricted to return depending on other rules */

The following terms are also used within the text of the rules:

| | |
|---|---|
| ***switch label*** | Either a case label or default label. |
| ***case clause*** | The code between any two switch labels. |
| ***default clause*** | The code between the default label and the end of the switch statement. |
| ***switch clause*** | Either a case clause or a default clause. |

A switch statement shall only contain *switch labels* and *switch clauses*, and no other code.

Any deviation from this normative text shall be considered a non-compliance if there are no other non-compliances with any other rule in section 15.

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

The scope of a *case* or *default* label shall be the compound statement, which is the body of a switch statement. All case clauses and the default clause shall be at the same scope.

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

[Koenig 22–24]

The last statement in every switch clause shall be a *break* statement, or if the *switch* clause is a compound statement, then the last statement in the compound statement shall be a *break* statement.

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

The requirement for a final *default* clause is defensive programming. This clause shall either take appropriate action or contain a suitable comment as to why no action is taken.

**Rule 15.4 (required): A *switch* expression shall not represent a value that is effectively Boolean.**

See “Boolean expressions” in the glossary.

For example:

```
switch (x == 0)   /* not compliant - effectively Boolean */
{
   ...
```

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

For example:

```
switch (x)
{
   uint8_t var;              /* not compliant - decl before 1st case     */
case 0:
   a = b;
   break;                    /* break is required here                   */
case 1:                      /* empty clause, break not required         */
case 2:
   a = c;                    /* executed if x is 1 or 2                  */
   if ( a == b )
   {
      case 3:                /* Not compliant - case is not allowed here */
   }
   break;                    /* break is required here                   */
case 4:
   a = b;                    /* Not compliant - non empty drop through   */
case 5:
   a = c;
   break;
default:                     /* default clause is required               */
   errorflag = 1;            /* should be non-empty if possible          */
   break;                    /* break is required here, in case a
                                future modification turns this into a
                                case clause                              */
}
```

## 6.16 Functions

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

[Unspecified 15; Undefined 25, 45, 61, 70–76]

There are a lot of potential problems with this feature. Users shall not write additional functions that use a variable number of arguments. This precludes the use of use of *stdarg.h*, *va_arg*, *va_start* and *va_end*.

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

This means that recursive function calls cannot be used in safety-related systems. Recursion carries with it the danger of exceeding available stack space, which can be a serious error. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

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

Names shall be given for all the parameters in the function declaration for reasons of compatibility, clarity and maintainability.

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

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

[Koenig 59–62]

Functions shall be declared with a return type (see Rule 8.2), that type being *void* if the function does not return any data. Similarly, if the function has no parameters, the parameter list shall be declared as *void*. Thus for example, a function, `myfunc`, which neither takes parameters nor returns a value would be declared as:

```
    void myfunc ( void );
```

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

[Undefined 22]

This problem is completely avoided by the use of function prototypes — see Rule 8.1. This rule is retained since compilers may not flag this constraint error.

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

This rule leads to greater precision in the definition of the function interface. The *const* qualification should be applied to the object pointed to, not to the pointer, since it is the object itself that is being protected.

For example:

```
void myfunc( int16_t * param1, const int16_t * param2, int16_t * param3)
/* param1: Addresses an object which is modified - no const
   param2: Addresses an object which is not modified - const required
   param3: Addresses an object which is not modified - const missing */
{
   *param1 = *param2 + *param3;
   return;
}
/* data at address param3 has not been changed,
   but this is not const therefore not compliant */
```

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

[Undefined 43]

This expression gives the value that the function returns. The absence of a *return* with an expression leads to undefined behaviour (and the compiler may not give an error).

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

[Koenig 24]

If the programmer writes:

```
if (f)   /* not compliant - gives a constant non-zero value which is
            the address of f - use either f() or &f                    */
{
   /* ... */
}
```

then it is not clear if the intent is to test if the address of the function is not *NULL* or that a call to the function `f()` should be made.

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

A function (whether it is part of the standard library, a third party library or a user defined function) may provide some means of indicating the occurrence of an error. This may be via an error flag, some special return value or some other means. Whenever such a mechanism is provided by a function the calling program shall check for the indication of an error as soon as the function returns.

However, note that the checking of input values to functions is considered a more robust means of error prevention than trying to detect errors after the function has completed (see Rule 20.3). Note also that the use of *errno* (to return error information from functions) is clumsy and should be used with care (see Rule 20.5).

## 6.17 Pointers and arrays

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

[Undefined 30]

Addition and subtraction of integers (including increment and decrement) from pointers that do not point to an array or array element results in undefined behaviour.

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

[Undefined 31]

Subtraction of pointers only gives well-defined results if the two pointers point (or at least behave as if they point) into the same array object.

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

[Undefined 33]

Attempting to make comparisons between pointers will produce undefined behaviour if the two pointers do not point to the same object. Note: it is permissible to address the next element beyond the end of an array, but accessing this element is not allowed.

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

Array indexing is the only acceptable form of pointer arithmetic, because it is clearer and hence less error prone than pointer manipulation. This rule bans the explicit calculation of pointer values. Array indexing shall only be applied to objects defined as an array type. Any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Pointers may go out of bounds of arrays or structures, or may even point to effectively arbitrary locations. See also Rule 21.1.

```
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                                   */
```

**Rule 17.5 (advisory): The declaration of objects should contain no more than 2 levels of pointer indirection.**

Use of more than 2 levels of indirection can seriously impair the ability to understand the behaviour of the code, and should therefore be avoided.

```
typedef int8_t * INTPTR;

struct s {
   int8_t *   s1;                            /* compliant     */
   int8_t **  s2;                            /* compliant     */
   int8_t *** s3;                            /* not compliant */
};

struct s *   ps1;                            /* compliant     */
struct s **  ps2;                            /* compliant     */
struct s *** ps3;                            /* not compliant */

int8_t **  (  *pfunc1)();                    /* compliant     */
int8_t **  ( **pfunc2)();                    /* compliant     */
int8_t **  (***pfunc3)();                    /* not compliant */
int8_t *** ( **pfunc4)();                    /* not compliant */

void function( int8_t *   par1,
               int8_t **  par2,
               int8_t *** par3,              /* not compliant */
               INTPTR *   par4,
               INTPTR *   const * const par5, /* not compliant */
               int8_t *   par6[],
               int8_t **  par7[])            /* not compliant */
{
   int8_t *   ptr1;
   int8_t **  ptr2;
   int8_t *** ptr3;                          /* not compliant */
   INTPTR *   ptr4;
   INTPTR *   const * const ptr5;            /* not compliant */
   int8_t *   ptr6[10];
   int8_t **  ptr7[10];
}
```

Explanation of types:

- *par1* and *ptr1* are of type pointer to *int8_t*.
- *par2* and *ptr2* are of type pointer to pointer to *int8_t*.
- *par3* and *ptr3* are of type pointer to a pointer to a pointer to *int8_t*. This is three levels and is not compliant.
- *par4* and *ptr4* are expanded to a type of pointer to a pointer to *int8_t*.
- *par5* and *ptr5* are expanded to a type of *const* pointer to a *const* pointer to a pointer to *int8_t*. This is three levels and is not compliant.
- *par6* is of type pointer to pointer to *int8_t* because arrays are converted to a pointer to the initial element of the array.
- *ptr6* is of type pointer to array of *int8_t*.
- *par7* is of type pointer to pointer to pointer to *int8_t* because arrays are converted to a pointer to the initial element of the array. This is three levels and is not compliant.
- *ptr7* is of type array of pointer to pointer to *int8_t*. This is compliant.

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

[Undefined 9, 26]

If the address of an automatic object is assigned to another automatic object of larger scope, or to a static object, or returned from a function then the object containing the address may exist beyond the time when the original object ceases to exist (and its address becomes invalid).

For example

```
int8_t * foobar(void)
{
   int8_t local_auto;
   return (&local_auto);    /* not compliant */
}
```

## 6.18 Structures and unions

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

[Undefined 35]

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. See section 6.1.2.5 of ISO/IEC 9899:1990 [2] for a full description of incomplete types.

```
struct tnode * pt;          /* tnode is incomplete at this point */
struct tnode
{
   int count;
   struct tnode *left;
   struct tnode * right;
};                          /* type tnode is now complete        */
```

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

[Undefined 34, 55]

The behaviour is undefined when two objects are created which have some overlap in memory and one is copied to the other.

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

This rule refers to the technique of using memory to store some data, and then using the same memory to store unrelated data at some other time during the execution of the program. Clearly it relies on the two different pieces of data existing at disjoint periods of the program's execution, and never being required simultaneously.

This practice is not recommended for safety-related systems as it brings with it a number of dangers. For example: a program might try to access data of one type from the location when actually it is storing a value of the other type (e.g. due to an interrupt). The two types of data may align differently in the storage, and encroach upon other data. Therefore the data may not be correctly initialised every time the usage switches. The practice is particularly dangerous in concurrent systems.

However it is recognised that sometimes such storage sharing may be required for reasons of efficiency. Where this is the case it is essential that measures are taken to ensure that the wrong type of data can never be accessed, that data is always properly initialised and that it is not possible to access parts of other data (e.g. due to alignment differences). The measures taken shall be documented and justified in the deviation that is raised against this rule.

This might be achieved by the use of unions, or various other means.

Note that there is no intention in the MISRA-C guidelines to place restrictions on how a compiler actually translates source code as the user generally has no effective control over this. So for example, memory allocation within the compiler whether dynamic heap, dynamic stack or static, may vary significantly with only slight code changes even at the same level of optimisation. (Note also that some optimisation may legitimately take place even when the user makes no specific request for this.)

**Rule 18.4 (required): Unions shall not be used.**

[Implementation 27]

Rule 18.3 prohibits the reuse of memory areas for unrelated purposes. However, even when memory is being reused for related purposes, there is still a risk that the data may be misinterpreted. Therefore, this rule prohibits the use of unions for any purpose.

It is recognised nonetheless that there are situations in which the careful use of unions is desirable in constructing an efficient implementation. In such situations, deviations to this rule are considered acceptable provided that all relevant implementation-defined behaviour is documented. This might be achieved in practice by referencing the implementation section of the compiler manuals from the design documentation. The kinds of implementation behaviour that might be relevant are:

- padding — how much padding is inserted at the end of the union
- alignment — how are members of any structures within the union aligned
- endianness — is the most significant byte of a word stored at the lowest or highest memory address
- bit-order — how are bits numbered within bytes and how are bits allocated to bit fields

The use of deviations is acceptable for (a) packing and unpacking of data, for example when sending and receiving messages, and (b) implementing variant records provided that the variants are differentiated by a common field. Variant records without a differentiator are not considered suitable for use in any situation.

## Packing and unpacking data

In this example, a union is used to access the bytes of a 32-bit word in order to store bytes received over a network most-significant byte first. The assumptions that this particular implementation rely on are:

- the *uint32_t* type occupies 32 bits
- the *uint8_t* type occupies 8 bits
- the implementation stores words with the most significant byte at the lowest memory address

The code to implement the receipt and packing of the bytes could be:

```
typedef union {
   uint32_t word;
   uint8_t bytes[4];
} word_msg_t;

uint32_t read_word_big_endian (void)

{
   word_msg_t tmp;

   tmp.bytes[0] = read_byte();
   tmp.bytes[1] = read_byte();
   tmp.bytes[2] = read_byte();
   tmp.bytes[3] = read_byte();

   return (tmp.word);
}
```

It is worth noting that the body of the routine could be written in a portable manner as follows:

```
uint32_t read_word_big_endian (void)
{
   uint32_t word;

   word =         ((uint32_t)read_byte()) << 24;
   word = word | (((uint32_t)read_byte()) << 16);
   word = word | (((uint32_t)read_byte()) << 8);
   word = word |   ((uint32_t)read_byte());

   return (word);
}
```

Unfortunately, most compilers produce far less efficient code when faced with the portable implementation. When high execution speed or low program memory usage is more important than portability, the implementation using unions might be preferred.

## Variant records

Unions are often used to implement variant records. Each variant shares common fields and has additional fields that are specific to the variant. This example is based on the CAN Calibration Protocol (CCP), in which each CAN message sent to a CCP client shares two common fields, each of one byte. Up to 6 additional bytes may follow, the interpretation of which depends on the message type stored in the first byte.

The assumptions that this particular implementation rely on are:

- the *uint16_t* type occupies 16 bits
- the *uint8_t* type occupies 8 bits
- the alignment and packing rules are such that there is no gap between the *uint8_t* and *uint16_t* members of the structure

In the interests of brevity, only two message types are considered in this example. The code that is presented here is incomplete and should be viewed merely to illustrate the purpose of variant records and not as a model implementation of CCP.

```
/* The fields common to all CCP messages */
typedef struct {
   uint8_t msg_type;
   uint8_t sequence_no;
} ccp_common_t;

/* CCP connect message */
typedef struct {
   ccp_common_t common_part;
   uint16_t station_to_connect;
} ccp_connect_t;

/* CCP disconnect message */
typedef struct {
   ccp_common_t common_part;
   uint8_t   disconnect_command;
   uint8_t   pad;
   uint16_t  station_to_disconnect;
} ccp_disconnect_t;
```

```
/* The variant */
typedef union {
   ccp_common_t     common;
   ccp_connect_t    connect;
   ccp_disconnect_t disconnect;
} ccp_message_t;

void process_ccp_message (ccp_message_t *msg)
{
   switch (msg->common.msg_type)
   {
   case Ccp_connect:
      if (MY_STATION == msg->connect.station_to_connect)
      {
         ccp_connect ();
      }
      break;

   case Ccp_disconnect:
      if (MY_STATION == msg->disconnect.station_to_disconnect)
      {
         if (PERM_DISCONNECT == msg->disconnect.disconnect_command)
         {
            ccp_disconnect ();
         }
      }
      break;

   default:
      break;      /* ignore unknown commands */
   }
}
```

## 6.19 Preprocessing directives

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

All the *#include* statements in a particular code file should be grouped together near the head of the file. The rule states that the only items which may precede a *#include* in a file are other preprocessor directives or comments.

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

[Undefined 14]

If the `'`, `\`, `"`, or `/*` characters are used between `<` and `>` delimiters or the `'`, `\`, or `/*` characters are used between the `"` delimiters in a header name preprocessing token, then the behaviour is undefined. 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.

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

[Undefined 48]

For example, the following are allowed.

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

**Rule 19.4 (required):** **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.**

[Koenig 82–84]

These are the only permitted uses of macros. Storage class specifiers and type qualifiers include keywords such as *extern*, *static* and *const*. Any other use of *#define* could lead to unexpected behaviour when substitution is made, or to very hard-to-read code.

In particular macros shall not be used to define statements or parts of statements except the use of the do-while construct. Nor shall macros redefine the syntax of the language. All brackets of whatever type `( ) { } [ ]` in the macro replacement list shall be balanced.

The do-while-zero construct (see example below) is the only permitted mechanism for having complete statements in a macro body. The do-while-zero construct is used to wrap a series of one or more statements and ensure correct behaviour. Note: the semicolon **must** be omitted from the end of the macro body.

For example:

```
/* The following are compliant */
#define PI 3.14159F                  /* Constant                    */
#define XSTAL 10000000               /* Constant                    */
#define CLOCK (XSTAL/16)             /* Constant expression         */
#define PLUS2(X) ((X) + 2)           /* Macro expanding to expression */
#define STOR extern                  /* storage class specifier     */
#define INIT(value){ (value), 0, 0}  /* braced initialiser          */
#define CAT (PI)                     /* parenthesised expression    */
#define FILE_A "filename.h"          /* string literal              */
#define READ_TIME_32() \
   do { \
      DISABLE_INTERRUPTS (); \
      time_now = (uint32_t)TIMER_HI << 16; \
      time_now = time_now | (uint32_t)TIMER_LO; \
      ENABLE_INTERRUPTS (); \
   } while (0)   /* example of do-while-zero   */

/* the following are NOT compliant */
#define int32_t long       /* use typedef instead                   */
#define STARTIF if(        /* unbalanced () and language redefinition */
#define CAT PI             /* non-parenthesised expression          */
```

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

While it is legal C to place *#define* or *#undef* directives anywhere in a code file, placing them inside blocks is misleading as it implies a scope restricted to that block, which is not the case.

Normally, *#define* directives will be placed near the start of a file, before the first function definition. Normally, *#undef* directives will not be needed (see Rule 19.6).

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

*#undef* should not normally be needed. Its use can lead to confusion with respect to the existence or meaning of a macro when it is used in the code.

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

[Koenig 78–81]

While macros can provide a speed advantage over functions, functions provide a safer and more robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times.

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

[Undefined 49]

This is a constraint error, but preprocessors have been known to ignore this problem. Each argument in a function-like macro must consist of at least one preprocessing token otherwise the behaviour is undefined.

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

[Undefined 50]

If any of the arguments act like preprocessor directives, the behaviour when macro substitution is made can be unpredictable.

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

[Koenig 78–81]

Within a definition of a function-like macro, the arguments shall be enclosed in parentheses. For example define an `abs` function using:

```
    #define abs(x) (((x) >= 0) ? (x) : -(x))
```

and not:

```
    #define abs(x) ((x >= 0) ? x : -x)
```

If this rule is not adhered to then when the preprocessor substitutes the macro into the code the operator precedence may not give the desired results.

Consider what happens if the second, incorrect, definition is substituted into the expression:

```
z = abs( a - b );
```

giving:

```
z = ((a - b >= 0) ? a - b : -a - b);
```

The sub-expression `-a - b` is equivalent to `(-a)-b` rather than `-(a-b)` as intended. Putting all the parameters in parentheses in the macro definition avoids this problem.

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

If an attempt is made to use an identifier in a preprocessor directive, and that identifier has not been defined, the preprocessor will sometimes not give any warning but will assume the value zero. *#ifdef*, *#ifndef* and *defined()* are provided to test the existence of a macro, and are therefore excluded.

For example:

```
#if x < 0   /* x assumed to be zero if not defined */
```

Consideration should be given to the use of a *#ifdef* test before an identifier is used.

Note that preprocessing identifiers may be defined either by use of *#define* directives or by options specified at compiler invocation. However the use of the *#define* directive is preferred.

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

[Unspecified 12]

There is an issue of unspecified order of evaluation associated with the `#` and `##` preprocessor operators. To avoid this problem only one occurrence of either operator shall be used in any single macro definition (i.e. one `#`, **or** one `##` **or** neither).

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

[Unspecified 12]

There is an issue of unspecified order of evaluation associated with the `#` and `##` preprocessor operators. Compilers have been inconsistent in the implementation of these operators. To avoid these problems do not use them.

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

[Undefined 47]

The only two permissible forms for the *defined* preprocessor operator are:

*defined (* identifier *)*

*defined* identifier

Any other form leads to undefined behaviour, for example:

```
#if defined(X > Y)    /* not compliant - undefined behaviour */
```

Generation of the token defined during expansion of a *#if* or *#elif* preprocessing directive also leads to undefined behaviour and shall be avoided, for example:

```
#define DEFINED defined
#if DEFINED(X)        /* not compliant - undefined behaviour */
```

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

When a translation unit contains a complex hierarchy of nested header files it can happen that a particular header file is included more than once. This can be, at best, a source of confusion. If it leads to multiple or conflicting definitions, the result can be undefined or erroneous behaviour.

Multiple inclusions of a header file can sometimes be avoided by careful design. If this is not possible, a mechanism must be in place to prevent the file contents from being included more than once. A common approach is to associate a macro with each file; the macro is defined when the file is included for the first time and used subsequently when the file is included again to exclude the contents of the file.

For example a file called "`ahdr.h`" might be structured as follows:

```
#ifndef AHDR_H
#define AHDR_H

/* The following lines will be excluded by the
   preprocessor if the file is included more
   than once */

...

#endif
```

Alternatively, the following may be used:

```
#ifdef AHDR_H
#error Header file is already included
#else
#define AHDR_H

/* The following lines will be excluded by the
   preprocessor if the file is included more
   than once */

...

#endif
```

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

When a section of source code is excluded by preprocessor directives, the content of each excluded statement is ignored until a *#else*, *#elif* or *#endif* directive is encountered (depending on the context). If one of these excluded directives is badly formed, it may be ignored without warning by a compiler with unfortunate consequences.

The requirement of this rule is that all preprocessor directives shall be syntactically valid even when they occur within an excluded block of code.

In particular, ensure that *#else* and *#endif* directives are not followed by any characters other than white-space. Compilers are not always consistent in enforcing this ISO requirement.

```
#define AAA 2
...
int foo(void)
{
   int x = 0;
   ...
#ifndef AAA
   x = 1;
#else1         /* Not compliant */
   x = AAA;
#endif
   ...
   return x;
}
```

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

When the inclusion and exclusion of blocks of statements is controlled by a series of preprocessor directives, confusion can arise if all of the relevant directives do not occur within one file. This rule requires that all preprocessor directives in a sequence of the form *#if* / *#ifdef* ... *#elif* ... *#else* ... *#endif* shall reside in the same file. Observance of this rule preserves good code structure and avoids maintenance problems.

Notice that this does not preclude the possibility that such directives may exist within included files so long as all directives that relate to the same sequence are located in one file.

**file.c**

```
#define A
...
#ifdef A
...
#include "file1.h"
#
#endif
...
#if 1
#include "file2.h"
...
EOF
```

**file1.h**

```
#if 1
...
#endif              /* Compliant     */
EOF
```

**file2.h**

```
...
#endif                  /* Not compliant */
```

## 6.20 Standard libraries

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

[Undefined 54, 57, 58, 62]

It is generally bad practice to *#undef* a macro which is defined in the standard library. 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. For example, there are some specific reserved words and function names which are known to give rise to undefined behaviour if they are redefined or undefined, including *defined*, *_ _LINE_ _*, *_ _FILE_ _*, *_ _DATE_ _*, *_ _TIME_ _*, *_ _STDC_ _*, *errno* and *assert*.

See also Rule 19.6 regarding the use of *#undef*.

Reserved identifiers are defined by ISO/IEC 9899:1990 [2] 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.

**Rule 20.2 (required): The names of standard library macros, objects and functions shall not be reused.**

Where new versions of standard library macros, objects or functions are used by the programmer (e.g. enhanced functionality or checks of input values) the modified macro, object or function shall have a new name. This is to avoid any confusion as to whether a standard macro, object or function is being used or whether a modified version of that function is being used. So, for example, if a new version of the *sqrt* function is written to check that the input is not negative, the new function shall not be named "sqrt" but shall be given a new name.

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

[Undefined 60, 63; Implementation 45, 47]

Many functions in the standard C libraries are not required by the ISO standard [2] to check the validity of parameters passed to them. Even where checking is required by the standard, or where compiler writers claim to check parameters, there is no guarantee that adequate checking will take place. Therefore the programmer shall provide appropriate checks of input values for all library functions which have a restricted input domain (standard libraries, other bought in libraries, and in-house libraries).

Examples of functions that have a restricted domain and need checking are:

- many of the maths functions in *math.h*, for example:
  - negative numbers must not be passed to the *sqrt* or *log* functions;
  - the second parameter of *fmod* should not be zero
- *toupper* and *tolower*: some implementations can produce unexpected results when the function *toupper* is passed a parameter which is not a lower case letter (and similarly for *tolower*)
- the character testing functions in *ctype.h* exhibit undefined behaviour if passed invalid values.
- the *abs* function applied to the most negative integer gives undefined behaviour.

Although most of the math library functions in *math.h* define allowed input domains, the values they return when a domain error occurs may vary from one compiler to another. Therefore pre-checking the validity of the input values is particularly important for these functions.

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.

There are a number of ways in which the requirements of this rule might be satisfied, including the following:

- Check the values before calling the function
- Design checks into the function. This is particularly applicable for in-house designed libraries, though it could apply to bought-in libraries if the supplier can demonstrate that they have built in the checks.
- Produce "wrapped" versions of functions, that perform the checks then call the original function.
- Demonstrate statically that the input parameters can never take invalid values.

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. However it will usually still be necessary to test to a tolerance around zero (or any other singularity) if the magnitude of the value of the function tends to infinity as the parameter tends to zero, so as to prevent overflow occurring.

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

[Unspecified 19; Undefined 91, 92; Implementation 69; Koenig 32]

This precludes the use of the functions *calloc*, *malloc*, *realloc* and *free*.

There is a whole range of unspecified, undefined and implementation-defined behaviour associated with dynamic memory allocation, as well as a number of other potential pitfalls. Dynamic heap memory allocation may lead to memory leaks, data inconsistency, memory exhaustion, non-deterministic behaviour.

Note that some implementations may use dynamic heap memory allocation to implement other functions (for example functions in the library *string.h*). If this is the case then these functions shall also be avoided.

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

[Implementation 46, Koenig 73]

*errno* is a facility of C, which in theory should be useful, but which in practice is poorly defined by the standard. A non zero value may or may not indicate that a problem has occurred; as a result it shall not be used. Even for those functions for which the behaviour of *errno* is well defined, it is preferable to check the values of inputs before calling the function rather than rely on using *errno* to trap errors (see Rule 16.10).

**Rule 20.6 (required): The macro *offsetof*, in library <*stddef.h*>, shall not be used.**

[Undefined 59]

Use of this macro can lead to undefined behaviour when the types of the operands are incompatible or when bit fields are used.

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

[Unspecified 14; Undefined 64–67, Koenig 74]

*setjmp* and *longjmp* allow the normal function call mechanisms to be bypassed, and shall not be used.

**Rule 20.8 (required): The signal handling facilities of <*signal.h*> shall not be used.**

[Undefined 68, 69; Implementation 48–52; Koenig 74]

Signal handling contains implementation-defined and undefined behaviour.

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

[Unspecified 2–5,16–18; Undefined 77–89; Implementation 53–68]

This includes file and I/O functions *fgetpos*, *fopen*, *ftell*, *gets*, *perror*, *remove*, *rename* and *ungetc*.

Streams and file I/O have a large number of unspecified, undefined and implementation-defined behaviours associated with them. It is assumed within this document that they will not normally be needed in production code in embedded systems.

If any of the features of *stdio.h* need to be used in production code, then the issues associated with the feature need to be understood.

**Rule 20.10 (required): The library functions *atof*, *atoi* and *atol* from library <*stdlib.h*> shall not be used.**

[Undefined 90]

These functions have undefined behaviour associated with them when the string cannot be converted.

**Rule 20.11 (required): The library functions *abort*, *exit*, *getenv* and *system* from library <*stdlib.h*> shall not be used.**

[Undefined 93; Implementation 70–73]

These functions will not normally be required in an embedded system, which does not normally need to communicate with an environment. If the functions are found necessary in an application, then it is essential to check on the implementation-defined behaviour of the function in the environment in question.

**Rule 20.12 (required): The time handling functions of library <*time.h*> shall not be used.**

[Unspecified 22; Undefined 97; Implementation 75, 76]

Includes *time, strftime*. This library is associated with clock times. Various aspects are implementation dependent or unspecified, such as the formats of times. If any of the facilities of *time.h* are used then the exact implementation for the compiler being used must be determined and a deviation raised.

## 6.21 Run-time failures

**Rule 21.1 (required): Minimisation of run-time failures shall be ensured by the use of at least one of**
**(a) static analysis tools/techniques;**
**(b) dynamic analysis tools/techniques;**
**(c) explicit coding of checks to handle run-time faults.**

[Undefined 19, 26, 94]

Run-time checking is an issue, which is not specific to C, but it is an issue which C programmers need to pay special attention to. This is because the C language is weak in its provision of any run-time checking. C implementations are not required to perform many of the dynamic checks that are necessary for robust software. It is therefore an issue that C programmers need to consider carefully, adding dynamic checks to code wherever there is potential for run-time errors to occur.

Where expressions consist only of values within a well-defined range, a run-time check may not be necessary, provided it can be demonstrated that for all values within the defined range the exception cannot occur. Such a demonstration, if used, should be documented along with the assumptions on which it depends. However if adopting this approach, be very careful about subsequent modifications of the code which may invalidate the assumptions, or of the assumptions changing for any other reason.

The following notes give some guidance on areas where consideration needs to be given to the provision of dynamic checks.

- ***arithmetic errors***

  This includes errors occurring in the evaluation of expressions, such as overflow, underflow, divide by zero or loss of significant bits through shifting.

  In considering integer overflow, note that unsigned integer calculations do not strictly overflow (producing undefined values), but the values wrap around (producing defined, but possibly wrong, values).

- ***pointer arithmetic***

  Ensure that when an address is calculated dynamically the computed address is reasonable and points somewhere meaningful. In particular it should be ensured that if a pointer points within a structure or array, then when the pointer has been incremented or otherwise altered it still points to the same structure or array. See restrictions on pointer arithmetic – Rules 17.1, 17.2 and 17.4.

- ***array bound errors***

  Ensure that array indices are within the bounds of the array size before using them to index the array.

- ***function parameters***

  See Rule 20.3.

- ***pointer dereferencing***

  Where a function returns a pointer and that pointer is subsequently de-referenced the program should first check that the pointer is not *NULL*. Within a function, it is relatively straightforward to reason about which pointers may or may not hold *NULL* values. Across function boundaries, especially when calling functions defined in other source files or libraries, it is much more difficult.

```
/* Given a pointer to a message, check the message header and return
   a pointer to the body of the message or NULL if the message is
   invalid. */
const char_t *msg_body (const char_t *msg)
{
   const char_t *body = NULL;
 if (msg != NULL)
 {
    if (msg_header_valid (msg))
```

```
         {
            body = &msg[MSG_HEADER_SIZE];
         }
      }
      return (body);
   }
   ...
         char_t msg_buffer[MAX_MSG_SIZE];
   const char_t *payload;
   ...
   payload = msg_body (msg_buffer);
   if (payload != NULL)
   {
      /* process the message payload */
   }
```

The techniques that will be employed to minimise run-time failures should be planned and documented, e.g. in design standards, test plans, static analysis configuration files, code review checklists.

# 7. References

[1] MISRA *Guidelines for the Use of the C Language In Vehicle Based Software*, ISBN 0-9524159-9-0, Motor Industry Research Association, Nuneaton, April 1998

[2] ISO/IEC 9899:1990, *Programming languages — C*, International Organization for Standardization, 1990

[3] Hatton L., Safer C — *Developing Software for High-integrity and Safety-critical Systems*, ISBN 0-07-707640-0, McGraw-Hill, 1994

[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] ANSI X3.159-1989, *Programming languages — C*, American National Standards Institute, 1989

[8] ISO/IEC 9899:1999, *Programming languages — C*, International Organization for Standardization, 1999

[9] MISRA *Development Guidelines for Vehicle Based Software*, ISBN 0-9524156-0-7, Motor Industry Research Association, Nuneaton, November 1994

[10] CRR80, *The Use of Commercial Off-the-Shelf (COTS) Software in Safety Related Applications*, ISBN 0-7176-0984-7, HSE Books

[11] ISO 9001:2000, *Quality management systems — Requirements*, International Organization for Standardization, 2000

[12] ISO 90003:2004, *Software engineering — Guidelines for the application of ISO 9001:2000 to computer software*, ISO, 2004

[13] The TickIT Guide, *Using ISO 9001:2000 for Software Quality Management System Construction, Certification and Continual Improvement*, Issue 5, British Standards Institution, 2001

[14] Straker D., *C Style: Standards and Guidelines*, ISBN 0–13-116898-3, Prentice Hall 1991

[15] Fenton N.E. and Pfleeger S.L., *Software Metrics: A Rigorous and Practical Approach*, 2nd Edition, ISBN 0-534-95429-1, PWS, 1998

[16] MISRA *Report 5 Software Metrics*, Motor Industry Research Association, Nuneaton, February 1995

[17] MISRA *Report 6 Verification and Validation*, Motor Industry Research Association, Nuneaton, February 1995

[18] Kernighan B.W., Ritchie D.M., *The C programming language*, 2nd edition, ISBN 0-13-110362-8, Prentice Hall, 1988 (note: The 1st edition is not a suitable reference document as it does not describe ANSI/ISO C)

[19] Koenig A., *C Traps and Pitfalls*, ISBN 0-201-17928-8, Addison-Wesley, 1988

# 7. References (continued)

[20] IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, International Electromechanical Commission, in 7 parts published between 1998 and 2000

[21] ANSI/IEEE Std 754, *IEEE Standard for Binary Floating-Point Arithmetic*, 1985

[22] ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*, International Organization for Standardization, 2003

# Appendix A

## Appendix A: Summary of rules

This appendix gives a summary of all the rules in section 6 of this document.

### Environment

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

### Language extensions

| | |
|---|---|
| 2.1 (req) | Assembly language shall be encapsulated and isolated. |
| 2.2 (req) | Source code shall only use `/* … */` style comments. |
| 2.3 (req) | The character sequence `/*` shall not be used within a comment. |
| 2.4 (adv) | Sections of code should not be "commented out". |

### Documentation

| | |
|---|---|
| 3.1 (req) | All usage of implementation-defined behaviour shall be documented. |
| 3.2 (req) | The character set and the corresponding encoding shall be documented. |
| 3.3 (adv) | The implementation of integer division in the chosen compiler should be determined, documented and taken into account. |
| 3.4 (req) | All uses of the *#pragma* directive shall be documented and explained. |
| 3.5 (req) | The implementation defined behaviour and packing of bitfields shall be documented if being relied upon. |
| 3.6 (req) | 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. |

## Character sets

| | |
|---|---|
| 4.1 (req) | Only those escape sequences that are defined in the ISO C standard shall be used. |
| 4.2 (req) | Trigraphs shall not be used. |

## Identifiers

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

## Types

| | |
|---|---|
| 6.1 (req) | The plain *char* type shall be used only for storage and use of character values. |
| 6.2 (req) | *signed* and *unsigned char* type shall be used only for the storage and use of numeric values. |
| 6.3 (adv) | *typedefs* that indicate size and signedness should be used in place of the basic numerical types. |
| 6.4 (req) | Bit fields shall only be defined to be of type *unsigned int* or *signed int*. |
| 6.5 (req) | Bit fields of *signed* type shall be at least 2 bits long. |

## Constants

| | |
|---|---|
| 7.1 (req) | Octal constants (other than zero) and octal escape sequences shall not be used. |

## Declarations and definitions

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

## Initialisation

| | |
|---|---|
| 9.1 (req) | All automatic variables shall have been assigned a value before being used. |
| 9.2 (req) | Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures. |
| 9.3 (req) | In an enumerator list, the "=" construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised. |

## Arithmetic type conversions

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

- (a) it is not a conversion to a wider integer type of the same signedness, or
- (b) the expression is complex, or
- (c) the expression is not constant and is a function argument, or
- (d) the expression is not constant and is a return expression

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

- (a) it is not a conversion to a wider floating type, or
- (b) the expression is complex, or
- (c) the expression is a function argument, or
- (d) the expression is a return expression

10.3 (req) 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.

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

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

10.6 (req): A "U" suffix shall be applied to all constants of *unsigned* type.

## Pointer type conversions

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

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

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

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

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

## Expressions

| | |
|---|---|
| 12.1 (adv) | Limited dependence should be placed on C's operator precedence rules in expressions. |
| 12.2 (req) | The value of an expression shall be the same under any order of evaluation that the standard permits. |
| 12.3 (req) | The *sizeof* operator shall not be used on expressions that contain side effects. |
| 12.4 (req) | The right-hand operand of a logical `&&` or `\|\|` operator shall not contain side effects. |
| 12.5 (req) | The operands of a logical `&& or \|\|` shall be *primary-expressions*. |
| 12.6 (adv) | The operands of logical operators (`&&`, `\|\|` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `\|\|` , `!`, `=`, `==`, `!=` and `?:`). |
| 12.7 (req) | Bitwise operators shall not be applied to operands whose underlying type is signed. |
| 12.8 (req) | The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. |
| 12.9 (req) | The unary minus operator shall not be applied to an expression whose underlying type is unsigned. |
| 12.10 (req) | The comma operator shall not be used. |
| 12.11 (adv) | Evaluation of constant unsigned integer expressions should not lead to wrap-around. |
| 12.12 (req) | The underlying bit representations of floating-point values shall not be used. |
| 12.13 (adv) | The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression. |

## Control statement expressions

| | |
|---|---|
| 13.1 (req) | Assignment operators shall not be used in expressions that yield a Boolean value. |
| 13.2 (adv) | Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. |
| 13.3 (req) | Floating-point expressions shall not be tested for equality or inequality. |
| 13.4 (req) | The controlling expression of a *for* statement shall not contain any objects of floating type. |
| 13.5 (req) | The three expressions of a *for* statement shall be concerned only with loop control. |
| 13.6 (req) | Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop. |
| 13.7 (req) | Boolean operations whose results are invariant shall not be permitted. |

## Control flow

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

## Switch statements

| | |
|---|---|
| 15.0 (req) | The MISRA C *switch* syntax shall be used. |
| 15.1 (req) | A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement. |
| 15.2 (req) | An unconditional *break* statement shall terminate every non-empty switch clause. |
| 15.3 (req) | The final clause of a *switch* statement shall be the *default* clause. |
| 15.4 (req) | A *switch* expression shall not represent a value that is effectively Boolean. |
| 15.5 (req) | Every *switch* statement shall have at least one *case* clause. |

## Functions

| | |
|---|---|
| 16.1 (req) | Functions shall not be defined with variable numbers of arguments. |
| 16.2 (req) | Functions shall not call themselves, either directly or indirectly. |
| 16.3 (req) | Identifiers shall be given for all of the parameters in a function prototype declaration. |
| 16.4 (req) | The identifiers used in the declaration and definition of a function shall be identical. |
| 16.5 (req) | Functions with no parameters shall be declared and defined with the parameter list *void*. |
| 16.6 (req) | The number of arguments passed to a function shall match the number of parameters. |
| 16.7 (adv) | A pointer parameter in a function prototype should be declared as pointer to *const* if the pointer is not used to modify the addressed object. |
| 16.8 (req) | All exit paths from a function with non-void return type shall have an explicit *return* statement with an expression. |
| 16.9 (req) | A function identifier shall only be used with either a preceding `&`, or with a parenthesised parameter list, which may be empty. |
| 16.10 (req) | If a function returns error information, then that error information shall be tested. |

## Pointers and arrays

| | |
|---|---|
| 17.1 (req) | Pointer arithmetic shall only be applied to pointers that address an array or array element. |
| 17.2 (req) | Pointer subtraction shall only be applied to pointers that address elements of the same array. |
| 17.3 (req) | >, >=, <, <= shall not be applied to pointer types except where they point to the same array. |
| 17.4 (req) | Array indexing shall be the only allowed form of pointer arithmetic. |
| 17.5 (adv) | The declaration of objects should contain no more than 2 levels of pointer indirection. |
| 17.6 (req) | The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. |

## Structures and unions

| | |
|---|---|
| 18.1 (req) | All structure or union types shall be complete at the end of a translation unit. |
| 18.2 (req) | An object shall not be assigned to an overlapping object. |
| 18.3 (req) | An area of memory shall not be reused for unrelated purposes. |
| 18.4 (req) | Unions shall not be used. |

## Preprocessing directives

| | |
|---|---|
| 19.1 (adv) | *#include* statements in a file should only be preceded by other preprocessor directives or comments. |
| 19.2 (adv) | Non-standard characters should not occur in header file names in *#include* directives. |
| 19.3 (req) | The *#include* directive shall be followed by either a *<filename>* or *"filename"* sequence. |
| 19.4 (req) | 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. |
| 19.5 (req) | Macros shall not be *#define*'d or *#undef*'d within a block. |
| 19.6 (req) | *#undef* shall not be used. |
| 19.7 (adv) | A function should be used in preference to a function-like macro. |
| 19.8 (req) | A function-like macro shall not be invoked without all of its arguments. |
| 19.9 (req) | Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. |
| 19.10 (req) | In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`. |
| 19.11 (req) | All macro identifiers in preprocessor directives shall be defined before use, except in *#ifdef* and *#ifndef* preprocessor directives and the *defined()* operator. |
| 19.12 (req) | There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition. |
| 19.13 (adv) | The `#` and `##` preprocessor operators should not be used. |
| 19.14 (req) | The *defined* preprocessor operator shall only be used in one of the two standard forms. |
| 19.15 (req) | Precautions shall be taken in order to prevent the contents of a header file being included twice. |
| 19.16 (req) | Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor. |
| 19.17 (req) | All *#else*, *#elif* and *#endif* preprocessor directives shall reside in the same file as the *#if* or *#ifdef* directive to which they are related. |

## Standard libraries

| | |
|---|---|
| 20.1 (req) | Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined. |
| 20.2 (req) | The names of standard library macros, objects and functions shall not be reused. |
| 20.3 (req) | The validity of values passed to library functions shall be checked. |
| 20.4 (req) | Dynamic heap memory allocation shall not be used. |
| 20.5 (req) | The error indicator *errno* shall not be used. |
| 20.6 (req) | The macro *offsetof*, in library <*stddef.h*>, shall not be used. |
| 20.7 (req) | The *setjmp* macro and the *longjmp* function shall not be used. |
| 20.8 (req) | The signal handling facilities of <*signal.h*> shall not be used. |
| 20.9 (req) | The input/output library <*stdio.h*> shall not be used in production code. |
| 20.10 (req) | The library functions *atof*, *atoi* and *atol* from library <*stdlib.h*> shall not be used. |
| 20.11 (req) | The library functions *abort*, *exit*, *getenv* and *system* from library <*stdlib.h*> shall not be used. |
| 20.12 (req) | The time handling functions of library <*time.h*> shall not be used. |

## Run-time failures

21.1 (req) Minimisation of run-time failures shall be ensured by the use of at least one of

(a) static analysis tools/techniques;
(b) dynamic analysis tools/techniques;
(c) explicit coding of checks to handle run-time faults.

# Appendix B: MISRA-C:1998 to MISRA-C:2004 rule mapping

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 1 (req) | 1.1 (req) | All code shall conform to ISO/IEC 9899:1990 "Programming languages — C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996 |
| 1 (req) | 1.2 (req) | No reliance shall be placed on undefined or unspecified behaviour. |
| 1 (req) | 2.2 (req) | Source code shall only use `/* … */` style comments. |
| 1 (req) | 3.1 (req) | All usage of implementation-defined behaviour shall be documented. |
| 2 (adv) | 1.3 (req) | Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform. |
| 3 (adv) | 2.1 (req) | Assembly language shall be encapsulated and isolated. |
| 4 (adv) | 21.1 (req) | Minimisation of run-time failures shall be ensured by the use of at least one of<br>(a) static analysis tools/techniques;<br>(b) dynamic analysis tools/techniques;<br>(c) explicit coding of checks to handle run-time faults. |
| 5 (req) | 4.1 (req) | Only those escape sequences that are defined in the ISO C standard shall be used. |
| 6 (req) | 3.2 (req) | The character set and the corresponding encoding shall be documented. |
| 7 (req) | 4.2 (req) | Trigraphs shall not be used. |
| 8 (req) | | Rescinded |
| 9 (req) | 2.3 (req) | The character sequence `/*` shall not be used within a comment. |
| 10 (adv) | 2.4 (adv) | Sections of code should not be "commented out". |
| 11 (req) | 1.4 (req) | The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers. |
| 11 (req) | 5.1 (req) | Identifiers (internal and external) shall not rely on the significance of more than 31 characters. |
| 12 (adv) | 5.5 (adv) | No object or function identifier with static storage duration should be reused. |
| 12 (adv) | 5.6 (adv) | No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names. |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 12 (adv) | 5.7 (adv) | No identifier name should be reused. |
| 13(adv) | 6.3 (adv) | *typedefs* that indicate size and signedness should be used in place of the basic numerical types. |
| 14 (req) | 6.1 (req) | The plain *char* type shall be used only for storage and use of character values. |
| 14 (req) | 6.2 (req) | *signed* and *unsigned char* type shall be used only for the storage and use of numeric values. |
| 15 (adv) | 1.5 (adv) | Floating-point implementations should comply with a defined floating-point standard. |
| 16 (req) | 12.12 (req) | The underlying bit representations of floating-point values shall not be used. |
| 17 (req) | 5.3 (req) | A *typedef* name shall be a unique identifier. |
| 18 (adv) | | Rescinded |
| 19 (req) | 7.1 (req) | Octal constants (other than zero) and octal escape sequences shall not be used. |
| 20 (req) | | Rescinded |
| 21 (req) | 5.2 (req) | Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. |
| 22 (adv) | 8.7 (req) | Objects shall be defined at block scope if they are only accessed from within a single function. |
| 23 (adv) | 8.10 (req) | All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. |
| 24 (req) | 8.11 (req) | The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage. |
| 25 (req) | 8.9 (req) | An identifier with external linkage shall have exactly one external definition. |
| 26 (req) | 8.4 (req) | If objects or functions are declared more than once their types shall be compatible. |
| 27 (adv) | 8.8 (req) | An external object or function shall be declared in one and only one file. |
| 28 (adv) | | Rescinded |
| 29 (req) | 5.4 (req) | A tag name shall be a unique identifier. |
| 30 (req) | 9.1 (req) | All automatic variables shall have been assigned a value before being used. |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 31 (req) | 9.2 (req) | Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures. |
| 32 (req) | 9.3 (req) | In an enumerator list, the "=" construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised. |
| 33 (req) | 12.4 (req) | The right-hand operand of a logical `&&` or `\|\|` operator shall not contain side effects. |
| 34 (req) | 12.5 (req) | The operands of a logical `&&` or `\|\|` shall be *primary-expressions*. |
| 35 (req) | 13.1 (req) | Assignment operators shall not be used in expressions that yield a Boolean value. |
| 36 (adv) | 12.6 (adv) | The operands of logical operators (`&&`, `\|\|` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `\|\|`, `!`, `=`, `==`, `!=` and `?:`). |
| 37 (req) | 10.5 (req) | If the bitwise operators `~` and `<<` are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand. |
| 37 (req) | 12.7 (req) | Bitwise operators shall not be applied to operands whose underlying type is *signed*. |
| 38 (req) | 12.8 (req) | The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. |
| 39 (req) | 12.9 (req) | The unary minus operator shall not be applied to an expression whose underlying type is *unsigned*. |
| 40 (adv) | 12.3 (req) | The *sizeof* operator shall not be used on expressions that contain side effects. |
| 41 (adv) | 3.3 (adv) | The implementation of integer division in the chosen compiler should be determined, documented and taken into account. |
| 42 (req) | 12.10 (req) | The comma operator shall not be used. |
| 43 (req) | 10.1 (req) | The value of an expression of integer type shall not be implicitly converted to a different underlying type if:<br>(a) it is not a conversion to a wider integer type of the same signedness, or<br>(b) the expression is complex, or<br>(c) the expression is not constant and is a function argument, or<br>(d) the expression is not constant and is a return expression |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 44 (adv) | | Rescinded |
| 45 (req) | 11.1 (req) | Conversions shall not be performed between a pointer to a function and any type other than an integral type. |
| 45 (req) | 11.2 (req) | Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to *void*. |
| 45 (req) | 11.3 (adv) | A cast should not be performed between a pointer type and an integral type. |
| 45 (req) | 11.4 (adv) | A cast should not be performed between a pointer to object type and a different pointer to object type. |
| 45 (req) | 11.5 (req) | A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer. |
| 46 (req) | 12.2 (req) | The value of an expression shall be the same under any order of evaluation that the standard permits. |
| 47 (adv) | 12.1 (adv) | Limited dependence should be placed on C’s operator precedence rules in expressions. |
| 48 (adv) | 10.4 (req) | The value of a complex expression of floating type shall only be cast to a floating type that is narrower or of the same size. |
| 49 (adv) | 13.2 (adv) | Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. |
| 50 (req) | 13.3 (req) | Floating-point expressions shall not be tested for equality or inequality. |
| 51 (adv) | 12.11 (adv) | Evaluation of constant unsigned integer expressions should not lead to wrap-around. |
| 52 (req) | 14.1 (req) | There shall be no unreachable code. |
| 53 (req) | 14.2 (req) | All non-null statements shall either (a) have at least one side-effect however executed, or (b) cause control flow to change. |
| 54 (req) | 14.3 (req) | Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character. |
| 55 (adv) | | Rescinded |
| 56 (req) | 14.4 (req) | The *goto* statement shall not be used. |
| 57 (req) | 14.5 (req) | The *continue* statement shall not be used. |
| 58 (req) | | Rescinded |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 59 (req) | 14.8 (req) | The statement forming the body of a *switch*, *while*, *do … while* or *for* statement shall be a compound statement. |
| 59 (req) | 14.9 (req) | An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement. |
| 60 (adv) | 14.10 (req) | All *if … else if* constructs shall be terminated with an *else* clause. |
| 61 (req) | 15.1 (req) | A *switch* label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement. |
| 61 (req) | 15.2 (req) | An unconditional *break* statement shall terminate every non-empty *switch* clause. |
| 62 (req) | 15.3 (req) | The final clause of a *switch* statement shall be the *default* clause. |
| 63 (adv) | 15.4 (req) | A *switch* expression shall not represent a value that is effectively Boolean. |
| 64 (req) | 15.5 (req) | Every *switch* statement shall have at least one *case* clause. |
| 65 (req) | 13.4 (req) | The controlling expression of a *for* statement shall not contain any objects of floating type. |
| 66 (adv) | 13.5 (req) | The three expressions of a *for* statement shall be concerned only with loop control. |
| 67 (adv) | 13.6 (req) | Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop. |
| 68 (req) | 8.6 (req) | Functions shall be declared at file scope. |
| 69 (req) | 16.1 (req) | Functions shall not be defined with variable numbers of arguments. |
| 70 (req) | 16.2 (req) | Functions shall not call themselves, either directly or indirectly. |
| 71 (req) | 8.1 (req) | Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call. |
| 72 (req) | 8.3 (req) | For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical. |
| 73 (req) | 16.3 (req) | Identifiers shall be given for all of the parameters in a function prototype declaration. |
| 74 (req) | 16.4 (req) | The identifiers used in the declaration and definition of a function shall be identical. |
| 75 (req) | 8.2 (req) | Whenever an object or function is declared or defined, its type shall be explicitly stated. |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 76 (req) | 16.5 (req) | Functions with no parameters shall be declared and defined with the parameter list *void*. |
| 77 (req) | 10.2 (req) | The value of an expression of floating type shall not be implicitly converted to a different type if: (a) it is not a conversion to a wider floating type, or (b) the expression is complex, or (c) the expression is a function argument, or (d) the expression is a return expression |
| 78 (req) | 16.6 (req) | The number of arguments passed to a function shall match the number of parameters. |
| 79 (req) | | Rescinded |
| 80 (req) | | Rescinded |
| 81 (adv) | 16.7 (adv) | A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object. |
| 82 (adv) | 14.7 (req) | A function shall have a single point of exit at the end of the function. |
| 83 (req) | 16.8 (req) | All exit paths from a function with non-void return type shall have an explicit *return* statement with an expression. |
| 84 (req) | | Rescinded |
| 85 (adv) | 16.9 (req) | A function identifier shall only be used with either a preceding `&`, or with a parenthesised parameter list, which may be empty. |
| 86 (adv) | 16.10 (req) | If a function returns error information, then that error information shall be tested. |
| 87 (req) | 8.5 (req) | There shall be no definitions of objects or functions in a header file. |
| 87 (req) | 19.1 (adv) | *#include* statements in a file should only be preceded by other preprocessor directives or comments. |
| 88 (req) | 19.2 (adv) | Non-standard characters should not occur in header file names in *#include* directives. |
| 89 (req) | 19.3 (req) | The *#include* directive shall be followed by either a *<filename>* or *"filename"* sequence. |
| 90 (req) | 19.4 (req) | 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. |
| 91 (req) | 19.5 (req) | Macros shall not be *#define*'d or *#undef*'d within a block. |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 92 (adv) | 19.6 (req) | *#undef* shall not be used. |
| 93 (adv) | 19.7 (adv) | A function should be used in preference to a function-like macro. |
| 94 (req) | 19.8(req) | A function-like macro shall not be invoked without all of its arguments. |
| 95 (req) | 19.9 (req) | Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. |
| 96 (req) | 19.10 (req) | In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`. |
| 97 (adv) | 19.11 (req) | All macro identifiers in preprocessor directives shall be defined before use, except in *#ifdef* and *#ifndef* preprocessor directives and the *defined()* operator. |
| 98 (req) | 19.12 (req) | There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition. |
| 98 (req) | 19.13 (adv) | The `#` and `##` preprocessor operators should not be used. |
| 99 (req) | 3.4 (req) | All uses of the *#pragma* directive shall be documented and explained. |
| 100 (req) | 19.14 (req) | The *defined* preprocessor operator shall only be used in one of the two standard forms. |
| 101 (adv) | 17.1 (req) | Pointer arithmetic shall only be applied to pointers that address an array or array element. |
| 101 (adv) | 17.2 (req) | Pointer subtraction shall only be applied to pointers that address elements of the same array. |
| 101 (adv) | 17.4 (req) | Array indexing shall be the only allowed form of pointer arithmetic. |
| 102 (adv) | 17.5 (adv) | The declaration of objects should contain no more than 2 levels of pointer indirection. |
| 103 (req) | 17.3 (req) | >, >=, <, <= shall not be applied to pointer types except where they point to the same array. |
| 104 (req) | | Rescinded |
| 105 (req) | | Rescinded |
| 106 (req) | 17.6 (req) | The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. |
| 107 (req) | | Rescinded. |

| MISRA-C: 1998 | MISRA-C: 2004 | MISRA-C: 2004 rule |
|---|---|---|
| 108 (req) | 18.1 (req) | All structure or union types shall be complete at the end of a translation unit. |
| 109 (req) | 18.2 (req) | An object shall not be assigned to an overlapping object. |
| 109 (req) | 18.3 (req) | An area of memory shall not be reused for unrelated purposes. |
| 110 (req) | 18.4 (req) | Unions shall not be used. |
| 111 (req) | 6.4 (req) | Bit fields shall only be defined to be of type *unsigned int* or *signed int*. |
| 112 (req) | 6.5 (req) | Bit fields of *signed* type shall be at least 2 bits long. |
| 113 (req) | | Rescinded. |
| 114 (req) | 20.1 (req) | Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined. |
| 115 (req) | 20.2 (req) | The names of standard library macros, objects and functions shall not be reused. |
| 116 (req) | 3.6 (req) | 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. |
| 117 (req) | 20.3 (req) | The validity of values passed to library functions shall be checked. |
| 118 (req) | 20.4 (req) | Dynamic heap memory allocation shall not be used. |
| 119 (req) | 20.5 (req) | The error indicator *errno* shall not be used. |
| 120 (req) | 20.6 (req) | The macro *offsetof*, in library <*stddef.h*>, shall not be used. |
| 121 (req) | | Rescinded |
| 122 (req) | 20.7 (req) | The *setjmp* macro and the *longjmp* function shall not be used. |
| 123 (req) | 20.8 (req) | The signal handling facilities of <*signal.h*> shall not be used. |
| 124 (req) | 20.9 (req) | The input/output library <*stdio.h*> shall not be used in production code. |
| 125 (req) | 20.10 (req) | The library functions *atof*, *atoi* and *atol* from library <*stdlib.h*> shall not be used. |
| 126 (req) | 20.11 (req) | The library functions *abort*, *exit*, *getenv* and *system* from library <*stdlib.h*> shall not be used. |
| 127 (req) | 20.12 (req) | The time handling functions of library <*time.h*> shall not be used. |

## Appendix C: MISRA-C:1998 – Rescinded rules

| | |
|---|---|
| Rule 8 (rescinded): | Multibyte characters and wide string literals shall not be used. |
| Rule 18 (rescinded): | Integer constants should be suffixed to reflect their type where an appropriate suffix is available. |
| Rule 20 (rescinded): | All object and function identifiers shall be declared before use. |
| Rule 28 (rescinded): | The *register* storage class specifier should not be used. |
| Rule 44 (rescinded): | Redundant explicit casts should not be used. |
| Rule 55 (rescinded): | Labels should not be used, except in *switch* statements. |
| Rule 58 (rescinded): | The *break* statement shall not be used (except to terminate the cases of a *switch* statement). |
| Rule 79 (rescinded): | The values returned by *void* functions shall not be used. |
| Rule 80 (rescinded): | Void expressions shall not be passed as function parameters. |
| Rule 84 (rescinded): | For functions with *void* return type, *return* statements shall not have an expression. |
| Rule 104 (rescinded): | Non-constant pointers to functions shall not be used. |
| Rule 105 (rescinded): | All the functions pointed to by a single pointer to function shall be identical in the number and type of parameters and the return type. |
| Rule 107 (rescinded): | The null pointer shall not be de-referenced. |
| Rule 113 (rescinded): | All the members of a structure (or union) shall be named and shall only be accessed via their name. |
| Rule 121 (rescinded): | *<locale.h>* and the *setlocale* function shall not be used |

# Appendix D: Cross references to the ISO standard

This appendix gives cross references between the rules given in this document and the sections of ISO/IEC 9899:1990 [2].

## D.1 MISRA-C:2004 rule numbers to ISO/IEC 9899:1990 references

| Rule | ISO Ref | Rule | ISO Ref | Rule | ISO Ref |
|---|---|---|---|---|---|
| **1.4** | 6.1.2 | **8.7** | 6.1.2.1, 6.5 | **12.7** | 6.3.3.3, 6.3.7, |
| **1.5** | 6.1.2.5 | **8.8** | 6.7 | | 6.3.10, 6.3.11 |
| **2.2** | 6.1.9 | **8.9** | 6.7 | | 6.3.12 |
| **2.3** | 6.1.9 | **8.10** | 6.1.2.1, 6.1.2.2, | **12.8** | 6.3.7 |
| **2.4** | 6.1.9 | | 6.5.4 | **12.9** | 6.3.3.3 |
| **3.3** | 6.3.5 | **8.11** | 6.1.2.2, 6.5.1, | **12.10** | 6.3.17 |
| **3.4** | 6.8.6 | | 6.5.4 | **12.11** | 6.4, 6.8.1 |
| **3.5** | 6.5.2.1 | **9.1** | 6.5.7 | **12.12** | 5.2.4.2.2, 6.1.2.5 |
| **4.1** | 5.2.2 | **9.2** | 6.5.7 | **12.13** | 6.3.2.4, 6.3.3.1 |
| **4.2** | 5.2.1.1 | **9.3** | 6.5.2.2 | **13.1** | 6.3.16 |
| **5.1** | 6.1.2 | **10.1** | 6.2 | **13.2** | 6.6.4.1, 6.6.5 |
| **5.2** | 6.1.2.1 | **10.2** | 6.2 | **13.3** | 5.2.4.2.2, 6.3.9 |
| **5.3** | 6.5.6 | **10.3** | 6.2, 6.3.4 | **13.4** | 6.6.5.3 |
| **5.4** | 6.5.2.3 | **10.4** | 6.2, 6.3.4 | **13.5** | 6.6.5.3 |
| **5.5** | 6.1.2.2 | **10.5** | 6.3.3.3, 6.3.7 | **13.6** | 6.6.5.3 |
| **5.6** | 6.1.2.3 | **10.6** | 6.1.3.2 | **14.2** | 5.1.2.3, 6.6.3 |
| **5.7** | 6.1.2 | **11.1** | 6.3.4 | **14.3** | 6.6.3 |
| **6.1** | 6.1.2.5 | **11.2** | 6.3.4 | **14.4** | 6.6.6.1 |
| **6.2** | 6.2.1.1 | **11.3** | 6.3.4 | **14.5** | 6.6.6.2 |
| **6.3** | 6.1.2.5, 6.5.2, | **11.4** | 6.3.4 | **14.6** | 6.6.6.3 |
| | 6.5.6 | **11.5** | 6.5.3 | **14.7** | 6.6.6.4 |
| **6.4** | 6.5.2.1 | **12.1** | 6.3 | **14.8** | 6.6.5 |
| **6.5** | 6.5.2.1 | **12.2** | 5.1.2.3, 6.3, 6.6 | **14.9** | 6.6.4.1 |
| **7.1** | 6.1.3.2 | **12.3** | 5.1.2.3, 6.3.3.4 | **14.10** | 6.6.4.1 |
| **8.1** | 6.3.2.2, 6.5.4.3 | **12.4** | 5.1.2.3, 6.3.13, | **15.0** | 6.6.4.2 |
| **8.2** | 6.5.4 | | 6.3.14 | **15.1** | 6.6.4.2 |
| **8.3** | 6.5.4.3, 6.7.1 | **12.5** | 6.3.1, 6.3.13, | **15.2** | 6.6.4.2, 6.6.6.3 |
| **8.4** | 6.1.2.6, 6.5 | | 6.3.14 | **15.3** | 6.6.4.2 |
| **8.5** | 6.8.2 | **12.6** | 6.3.3.3, 6.3.13, | **15.4** | 6.6.4.2 |
| **8.6** | 6.1.2.1, 6.5.4.3 | | 6.3.14 | **15.5** | 6.6.4.2 |

| Rule | ISO Ref | Rule | ISO Ref | Rule | ISO Ref |
|---|---|---|---|---|---|
| **16.1** | 6.7.1, 7.8 | **18.2** | 6.3.16.1 | **19.15** | 6.8 |
| **16.2** | 6.3.2.2 | **18.4** | 6.1.2.5, 6.3.2.3, | **19.16** | 6.8 |
| **16.3** | 6.5.4.3 | | 6.5.2.1 | **20.1** | 6.8.8, 7.1.3, 7.13 |
| **16.4** | 6.5.4.3 | **19.1** | 6.8.2 | **20.2** | 7.1.3 |
| **16.5** | 6.5.4.3, 6.7.1 | **19.2** | 6.1.7 | **20.3** | 7.1.7, 7.3, 7.5.1, |
| **16.6** | 6.3.2.2 | **19.3** | 6.8.2 | | 7.5.6.4 |
| **16.7** | 6.5.4.1, 6.5.4.3 | **19.4** | 6.8.3 | **20.4** | 7.10.3 |
| **16.8** | 6.6.6.4 | **19.5** | 6.8.3, 6.8.3.5 | **20.5** | 7.1.4, 7.5.1 |
| **16.9** | 6.3.2.2, 6.3.3.2 | **19.6** | 6.8.3.5 | **20.6** | 7.1.6 |
| **17.1** | 6.3.6, 6.3.16.2 | **19.7** | 6.8.3 | **20.7** | 7.6 |
| **17.2** | 6.3.6, 6.3.16.2 | **19.8** | 6.8.3 | **20.8** | 7.7 |
| **17.3** | 6.3.8 | **19.9** | 6.8.3 | **20.9** | 7.9 |
| **17.4** | 6.3.2.1, 6.3.6 | **19.10** | 6.8.3 | **20.10** | 7.10.1 |
| **17.5** | 6.3.3.2, 6.5.4.1 | **19.11** | 6.8 | **20.11** | 7.10.4 |
| **17.6** | 6.1.2.4, 6.3.3.2 | **19.12** | 6.8.3.2, 6.8.3.3 | **20.12** | 7.12 |
| **18.1** | 6.1.2.5, 6.5, | **19.13** | 6.8.3.2, 6.8.3.3 | **21.1** | 6.3, 6.3.3.2, |
| | 6.5.2.1 | **19.14** | 6.8.1 | | 7.10.6 |

## D.2 ISO/IEC 9899:1990 references to MISRA-C:2004 rule numbers

| ISO Ref | Rule | ISO Ref | Rule | ISO Ref | Rule |
|---|---|---|---|---|---|
| **5.1.2.3** | 12.2, 12.3, 12.4, 14.2 | **6.3.5** | 3.3 | **6.6.5** | 13.2, 14.8 |
| | | **6.3.6** | 17.1, 17.2, 17.4 | **6.6.5.3** | 13.4, 13.5, 13.6 |
| **5.2.2** | 4.1 | **6.3.7** | 10.5, 12.7, 12.8 | **6.6.6.1** | 14.4 |
| **5.2.1.1** | 4.2 | **6.3.8** | 17.3 | **6.6.6.2** | 14.5 |
| **5.2.4.2.2** | 12.12, 13.3 | **6.3.9** | 13.3 | **6.6.6.3** | 14.6, 15.2 |
| **6.1.2** | 1.4, 5.1, 5.7 | **6.3.10** | 12.7 | **6.6.6.4** | 14.7, 16.8 |
| **6.1.2.1** | 5.2, 8.6, 8.7, 8.10 | **6.3.11** | 12.7 | **6.7** | 8.8, 8.9 |
| **6.1.2.2** | 5.5, 8.10, 8.11 | **6.3.12** | 12.7 | **6.7.1** | 8.3, 16.1, 16.4, 16.5 |
| **6.1.2.3** | 5.6 | **6.3.13** | 12.4, 12.5, 12.6 | | |
| **6.1.2.4** | 17.6 | **6.3.14** | 12.4, 12.5, 12.6 | **6.8** | 19.11, 19.15 |
| **6.1.2.5** | 1.5, 6.2, 6.3, 12.12, 18.1, 18.4 | **6.3.16** | 13.1 | **6.8.1** | 12.11, 19.14 |
| | | **6.3.16.1** | 18.2 | **6.8.2** | 8.5, 19.1, 19.3 |
| **6.1.2.6** | 8.4 | **6.3.16.2** | 17.1, 17.2 | **6.8.3** | 19.4, 19.5, 19.7 – 10 |
| **6.1.3.2** | 7.1, 10.6 | **6.3.17** | 12.10 | | |
| **6.1.7** | 19.2 | **6.4** | 12.11 | **6.8.3.2** | 19.12, 19.13 |
| **6.1.9** | 2.2, 2.3, 2.4 | **6.5** | 8.4, 8.7, 18.1 | **6.8.3.3** | 19.12, 19.13 |
| **6.2** | 10.1, 10.2, 10.3, 10.4 | **6.5.1** | 8.11 | **6.8.3.5** | 19.5, 19.6 |
| | | **6.5.2** | 6.3 | **6.8.6** | 3.4 |
| **6.2.1.1** | 6.1 | **6.5.2.1** | 3.5, 6.4, 6.5, 18.1, 18.4 | **6.8.8** | 20.1 |
| **6.3** | 12.1, 12.2, 21.1 | | | **7.1.3** | 20.1, 20.2 |
| **6.3.1** | 12.5 | **6.5.2.2** | 9.3 | **7.1.6** | 20.6 |
| **6.3.2.1** | 17.4 | **6.5.2.3** | 5.4 | **7.1.7** | 20.3 |
| **6.3.2.2** | 8.1, 16.2, 16.6, 16.9 | **6.5.3** | 11.5 | **7.3** | 20.3 |
| | | **6.5.4** | 8.2, 8.10, 8.11 | **7.5.1** | 20.3, 20.5 |
| **6.3.2.3** | 18.4 | **6.5.4.1** | 16.7, 17.5 | **7.5.6.4** | 20.3 |
| **6.3.2.4** | 12.13 | **6.5.4.3** | 8.1, 8.2, 8.3, 8.6, 16.3, 16.4, 16.5, 16.7 | **7.6** | 20.7 |
| **6.3.3.1** | 12.13 | | | **7.7** | 20.8 |
| **6.3.3.2** | 16.9, 17.5, 17.6, 21.1 | | | **7.8** | 16.1 |
| | | **6.5.6** | 5.3, 6.3 | **7.9** | 20.9 |
| **6.3.3.3** | 10.5, 12.6, 12.7, 12.9 | **6.5.7** | 9.1, 9.2 | **7.10.1** | 20.10 |
| | | **6.6** | 12.2 | **7.10.3** | 20.4 |
| **6.3.3.4** | 12.3 | **6.6.3** | 14.2, 14.3 | **7.10.6** | 21.1 |
| **6.3.4** | 10.3, 10.4, 11.1, 11.2, 11.3, 11.4 | **6.6.4.1** | 13.2, 14.9, 14.10 | **7.12** | 20.12 |
| | | **6.6.4.2** | 15.0 – 15.5 | **7.13** | 20.1 |

# Appendix E

## Appendix E: Glossary

### Underlying type

See section 6.10.4 “Underlying type”.

### Rule scope

The MISRA rules apply to **all** source files in a project. That is:

- Files being written by the project team
- All files supplied by other project teams and sub contractors
- Compiler library header files but not the library files only supplied as object code
- All third party library sources, particularly header files

It is recognised that there may be some problems getting third party source to comply but many tool and library vendors are adopting MISRA-C for their source code.

### Boolean expressions

Strictly speaking, there is no Boolean type in C, but there is a conceptual difference between expressions which return a numeric value and expressions which return a Boolean value. An expression is considered to represent a Boolean value either because it appears in a position where a Boolean value is expected or because it uses an operator that gives rise to a Boolean value.

Boolean values are expected in the following contexts:

- the controlling expression of an *if* statement
- the controlling expression of an iteration statement
- the first operand of the conditional operator `?`

Each of these contexts requires an “effectively Boolean” expression which is either Boolean-by-construct or Boolean-by-enforcement as defined below.

Boolean-by-construct values are produced by the following operators:

- equality operators (== and `!=`)
- logical operators (`!`, `&&` and `||`)
- relational operators (<, >, <= and >=)

Boolean-by-enforcement values can be introduced by implementing a specific type enforcement mechanism using a tool. A Boolean type could be associated with a specific *typedef*, and would then be used for any objects that are Boolean. This could bring many benefits, especially if the checking tool can support it, and in particular it could help avoid confusion between logical operations and integer operations.

### Small integer type

The term *small integer type* is used to describe those integer types that are subject to integral promotion. The affected types are *char*, *short*, *bit-field* and *enum*.