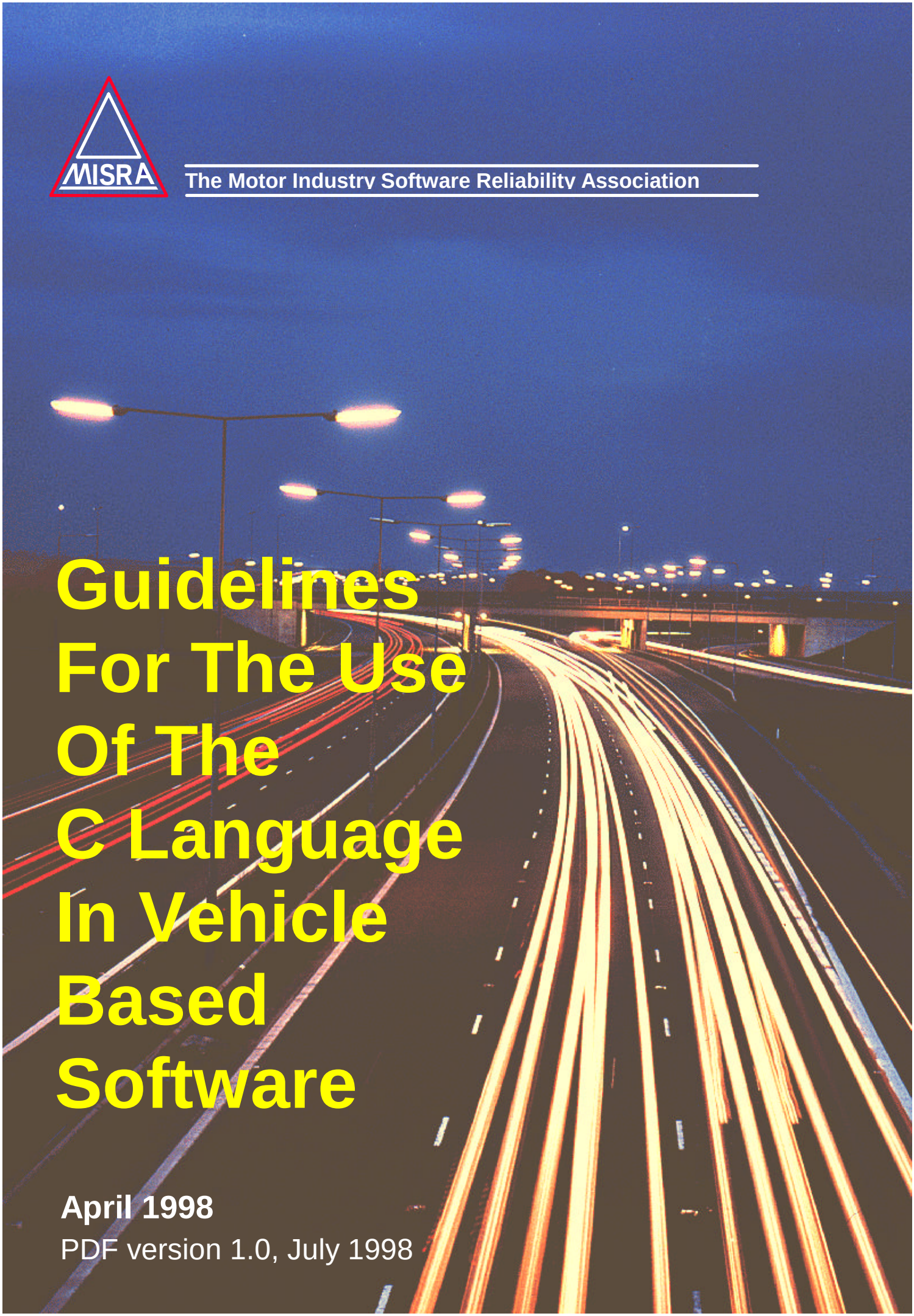
MISRA
The Motor Industry Software Reliability Association
Guidelines For The Use Of The C Language In Vehicle Based Software
April 1998
PDF version 1.0, July 1998

PDF version 1.0 © MIRA, 1998

This electronic version of the MISRA C Guidelines is issued solely for the use of member companies of the MISRA Consortium.

MIRA grants permission for member companies to distribute this PDF file within their own companies. The file should not be altered in any way, nor should hard copies be made. **No permission is given for use or distribution of this file by or to individuals or companies who are not members of the MISRA consortium, and any such use constitutes an infringement of copyright.**

MISRA gives no guarantees about the accuracy of the information contained in this PDF version of the Guidelines, and the published paper document should be taken as authoritative.

Information is available from the MISRA web site on how to obtain printed copies of the document.

First published April 1998
by The Motor Industry Research Association
Watling Street
Nuneaton
Warwickshire CV10 0TU

http://www.misra.org.uk/

Available in paperback (ISBN 0 9524156 9 0)

British Library Cataloguing in Publication Data.

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

The Motor Industry Software Reliability Association

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

April 1998

(PDF version 1.0, July 1998)

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.

Executive summary

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

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

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

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

Acknowledgements

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

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

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

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

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

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

Contents

Contents (continued)

1. Background

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

1.1 The use of C in the automotive industry

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

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

1.2 Language insecurities and the C language

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

1.2.1 The programmer makes mistakes

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

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

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

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

1.2.2 The programmer misunderstands the language

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

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

1.2.3 The compiler doesn't do what the programmer expects

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

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

1.2.4 The compiler contains errors

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

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

1.2.5 Run-time errors

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

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

1.3 The use of C for safety-related systems

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

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

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

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

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

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

1.4 C standardization

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

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

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

2. MISRA C: The vision

2.1 Rationale for the production of MISRA C

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

2.2 Objectives of MISRA C

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

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

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

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

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

3. Developing the subset

3. MISRA C: Developing the subset

The list of rules for this document was developed in three main phases. Firstly, a survey was conducted of available source material; secondly, this information was filtered and adapted into rules; and thirdly, the resulting material was subject to expert review and revision.

The initial information was collected from a variety of sources, both published and anecdotal. The aim at this stage was to collect as much information as possible on the concerns that people have with the C language and its 'danger areas'. Published sources of information included:

- The 'portability' annex (Annex G) of the ISO C standard [1]
- "Safer C" by Les Hatton [2]
- "C Traps and Pitfalls" by Andrew Koenig [5]

Standards created in-house by member companies of the MISRA consortium were also collated to provide additional information based on experience. Finally, additional ideas were collected from individuals, course notes and other such sources.

This pool of gathered information provided, in some cases, direct ideas for rules, and in other cases issues or problems with the language or its use which ought to be addressed in some way. Thus the second stage, of producing the list of rules, involved filtering and refining existing rules, and developing rules to address other issues which were identified.

Although there is no totally objective way of arriving at a given set of rules, the following principles were used to guide this stage of the process:

- Issues of style were omitted, i.e. issues which were not believed to have any significant impact on code integrity, but which were more matters of preference.
- As far as practicable, rules were to be enforceable by static tools. In some cases, however, rules which are not statically enforceable were retained because the advice they give needs to be heeded by the programmer.

4. Scope

4. MISRA C: Scope

4.1 Base language issues

The MISRA Guidelines [4] (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

4.2 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 5.2.2 and 5.2.4).

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.

4.3 Applicability

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

In terms of the execution environments defined by ISO 9899 [1] (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.

4.4 Safety Integrity Level issues

This document requires that all its provisions shall be adopted in systems classified as Integrity Levels 2 or 3 as defined in the MISRA Guidelines [4] (Debilitating or Difficult to control).

To be consistent with the MISRA Guidelines [4] (3.2.4.3b), no recommendations are made at this time for Integrity Level 4 (Uncontrollable).

4.5 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.

4.6 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 1).

4.7 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 [6].

No judgement on the suitability of auto-code generation for MISRA Integrity Levels 2 and 3 should be implied by the inclusion of the topic within this document. Auto-generated code must be treated in just the same manner as manually produced code for the purpose of validation (See MISRA Guidelines [4] 3.1.3, Planning for V&V).

5. Using MISRA C

5.1 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 [4] and, for example, include:

- Documented development process
- Quality system capable of meeting the requirements of ISO9001/ISO9000-3/TickIT [7,8,9]
- 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.

5.2 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.

5.2.1 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

5.2.2 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 [10].

5.2.3 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 [1]) then the developer must benchmark the compiler to establish that the implementation is as documented by the compiler writer. See section 6.5.2 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 ISO9001/ISO9000-3 [7, 8].

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.

5.2.4 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 [11] and the MISRA report on Software Metrics [12].

5.2.5 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 rigor 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 [12, 13].

5.3 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

5.3.1 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	warning 347				
2		error 25			
3			message 38		
4				warning 97	
5					✓

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.

5.3.2 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. Deviations for a class of circumstances are referred to as standing deviations and may apply at different levels within an organisation, e.g. company, department, project. They are expected to detail alternative means of ensuring that the potential problem is avoided. They must also specify how these alternative measures are to be enforced. Standing 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 standing deviations should then be restricted to this input/output section of the code. Code subject to standing 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.

5.3.3 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.

5.3.4 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 [2]). 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).

5.4 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 5.2 have been addressed.

5.5 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.

6. Introduction to the rules

This section explains the presentation of the rules in section 7 of this document. It serves as an introduction to the main content of the document as presented in that section.

6.1 Rule classification

Every rule in section 7 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.

6.1.1 Required rules

These are mandatory requirements placed on the programmer. There are 93 ‘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 5.3.2).

6.1.2 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 34 ‘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.

6.2 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, and then at the start of each topic there are cross-references to those other rules which are also relevant.

6.3 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.

6.4 Presentation of rules

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

Rule <number> (<category>): <requirement text>

[<source ref>]

where the fields are as follows:

- <*number*> Every rule has a unique number. These are allocated sequentially through the document, regardless of the category of the requirement or the section.
- <*category*> is one of 'required' or 'advisory', as explained in section 6.1.
- <*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 6.5 for an explanation of the significance of these references, and a key to the source materials.

In addition, supporting text is provided for each item or group of related items. The 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 supporting 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 supporting 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 (this is to comply with Rule 13):

UI_8	Unsigned 8 bit integer
SI_16	Signed 16 bit integer
UI_16	Unsigned 16 bit integer
UI_32	Unsigned 32 bit integer
F_32	32 bit floating point
F_64	64 bit floating point

6.5 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 6.5.2).

Rules which do not have a source reference will have originated from one of the other sources (as listed in section 3). For example, a rule 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.

6.5.1 Key to the source references

Reference	Source
	Annex G of ISO 9899 [1]
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 [4]
K&R	Kernighan and Ritchie [14]
Koenig	"C Traps and Pitfalls", Koenig [5]

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

- Annex G of ISO 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).

6.5.2 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 [2].

6.5.2.1 Unspecified

These are language constructs which 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 46. 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.

6.5.2.2 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. There are 97 such issues.

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

6.5.2.3 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.

6.5.2.4 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.

Rule 121 states that the locale shall not be changed, and therefore avoids these issues.

7. Rules

7.1 Environment

See also:	Rules 11, 15, 88

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

[MISRA Guidelines Table 3]

The MISRA Guidelines [4] require the use of a 'standardized structured language'. The rule applies this to the C programming language.

It is recognised that it may be necessary to raise deviations (as described in section 5.3.2) to permit certain language extensions, for example to support hardware specific features.

Rule 2 (advisory): **Code written in languages other than C should only be used if there is a defined interface standard for object code to which the compilers/assemblers for both languages conform.**

[Unspecified 11]

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

Rule 3 (advisory): **Assembly language functions that are called from C should be written as C functions containing only in-line assembly language, and in-line assembly language should not be embedded in normal C code.**

[Unspecified 11]

Assembly language instructions should not be embedded in-line in C code. However where assembly language instructions are required it is recommended that they be encapsulated in a C function which is solely for this purpose (i.e. does not include any C code). Implementing such a function then makes use of C's built in parameter passing mechanisms.

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, so that the assembly language is encapsulated in a macro which is then used.

Note that the use of in-line assembly language is an extension to standard C, and therefore requires a deviation against Rule 1.

Rule 4 (advisory): Provision should be made for appropriate run-time checking.

[Undefined 19,96]

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 which are necessary for robust software.

It is therefore an issue which C programmers need to consider carefully, adding dynamic checks to code wherever there is potential for run-time errors to occur. Clearly in practical terms this may have to be balanced against performance and code-size constraints.

In embedded automotive systems there may sometimes be a problem finding an appropriate action to take at run time on detecting an error. However at very least, logging a diagnostic code may assist in removing the problem in the future. Often a better strategy is to provide some sort of error recovery mechanism such as using a default value when the computed value is thought to be erroneous.

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. Also be aware of the possibility of data in memory becoming corrupted and taking on seemingly random values.

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 or divide by zero.

  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.
- ***left shifts***
  Left shifting of an integer may cause the most significant bit(s) to be lost (effectively a kind of overflow).
- ***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 117.

7.2 Character Sets

Rule 5 (required): **Only those characters and escape sequences which are defined in the ISO C standard shall be used**

[Undefined 11; Implementation 7]

Section 5.2.1 of the ISO standard defines 91 characters which are the minimum source character set for all compilers. These are the only characters that should be used, even if the compiler supports a larger character set, or supports alternative character sets.

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

ISO 10646-1 [15] defines an international standard for mapping character sets to numeric values. The chosen implementation of C shall conform to a subset of ISO 10646-1, and the chosen subset shall be documented.

Rule 7 (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 ~~]"
```

Therefore trigraphs should be avoided. If the compiler has a switch to ignore trigraphs then this option should be used, or alternatively ensure that two adjacent question marks are never used in the code (this is simple to statically check).

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

[Undefined 3,13,95; Implementation 8,12,13]

Multibyte characters provide a way to extend the source character set. There are various undefined and implementation-defined behaviours associated with them and they shall not be used. This in turn precludes the use of wide string literals and wide character constants.

Wide string literals are denoted by an `L` in front of the string, e.g.

```
x = L"Fred";
```

7.3 Comments

Rule 9 (required): **Comments shall not be nested.**

C does not support the nesting of comments. After a `/*` begins a comment, the comment continues until the first `*/` is encountered, with no regard for any nesting which has been attempted.

Rule 10 (advisory): **Sections of code should not be 'commented out'.**

Where it is required for sections of source code to not be compiled then this should be achieved by use of conditional compilation (e.g. *#if* or *#ifdef* constructs). 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.

7.4 Identifiers

See also:	Rules 21, 24

Rule 11 (required): **Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.**

[Undefined 7; Implementation 4–6]

The ISO standard only requires external identifiers to be significant in the first 6 characters, though this is a severe restriction and most compilers/linkers allow at least 31 character significance (as for internal identifiers). This rule allows for 31 character significance for external identifiers. The compiler/linker must be checked to establish this behaviour. If the compiler/linker is not capable of meeting this, then use the limit of the compiler.

The main purpose of this rule is to ensure that code can be ported between the majority of compilers/linkers without requiring modification (shortening) of parameter names.

Note that there is a related issue with using identifier names which 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 5.2.2).

Rule 12 (advisory): **No identifier in one name space shall have the same spelling as an identifier in another name space.**

ISO C defines a number of different name spaces (see ISO 9899 [1] section 6.1.2.3). 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.

An exception to this rule is the naming of members of structures, where member names may be reused within separate structures.

7.5 Types

See also:	Rules 18, 28

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

The storage length of types can vary from compiler to compiler. It is safer if programmers work with types which they know to be of a given length. For example `SI_16` might be the *typedef* chosen for a signed 16 bit integer. The known-length substitute types should be defined in a header file which can then be included in all code files.

So, for example, on a platform where integers are 32 bit and short integers are 16 bit, the header file would contain the *typedef*:

```
typedef signed short SI_16;
```

whereas on a platform where integers are 16 bit and short integers are 8 bit, the header file would contain the *typedef*:

```
typedef signed int SI_16;
```

Code files would then include the appropriate header file, and programmers should only use the type `SI_16`. The new type names should be used for declaring the type of an identifier, and also for explicit cast operators.

Note that one suggested convention has been used throughout this document as an example, but the choice of naming convention used is at the discretion of the user of these guidelines.

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

[Implementation 14]

The type *char* may be implemented as a signed or an unsigned type depending on the compiler. Rather than making any assumptions about the compiler, it is preferable (and more portable) to always specify whether the required use of *char* is signed or unsigned.

If Rule 13 is being followed then this rule should only be relevant in setting up the initial *typedef*s.

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

Floating point arithmetic has a range of problems associated with it. A fuller discussion of the issues associated with the use of floating point in safety-related systems is contained in the MISRA Guidelines [4].

Some (but not all) of the problems can be overcome by using an implementation which conforms to a recognised standard. An example of an appropriate standard is ANSI/IEEE Std 754 [16].

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

[Unspecified 6; Implementation 20]

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

Rule 17 (required): ***typedef* names shall not be reused.**

Once a name has been assigned as a *typedef* it should not be used for any other purpose in any of the code files.

7.6 Constants

See also:	Rules 8, 51, 81

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

Specific suffixes are available in C to denote numeric constants as being of a particular type. Suffixes are provided for indicating floating point constants to be float (`f` or `F`) or long double (`l` or `L`), as opposed to the default of double; and for indicating integer constants to be (if the constant will fit) unsigned int (`u` or `U`) and/or long int (`l` or `L`), as opposed to the default (int (signed) for integers which can be represented as such). A full description of the meanings of constant suffixes can be found in sections 6.1.3.1 and 6.1.3.2 of ISO 9899 [1].

It is recommended that, for long double and long int, only the capital '`L`' suffix is used, as lower case '`l`' is easily confused with the digit '`1`' (one).

Thus, for example, a constant which is being used to give a value to a quantity which is represented as an unsigned integer should be given the '`u`' suffix. In particular this technique can be used in many situations to avoid mixing signed and unsigned arithmetic in an expression (see Rule 43). If `pressure` is declared as an *unsigned int*, then the following statement mixes signed and unsigned arithmetic, because the constant '`3`' is an int (signed):

```
pressure = pressure + 3;
```

This should instead be written as:

```
pressure = pressure + 3u;
```

in which case all the elements in the expression are unsigned.

Rule 19 (required): Octal constants (other than zero) 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;      /* set to decimal 109 */
code[2] = 100;      /* set to decimal 100 */
code[3] = 052;      /* set to decimal 42 */
code[4] = 071;      /* set to decimal 57 */
```

It is better not to use octal constants at all, and to statically check for any occurrences. Zero is a special case, because strictly speaking '0' is octal zero.

7.7 Declarations and Definitions

See also:	Rules 11, 13, 17, 30–32, 68, 71–76, 108–110, 113

Rule 20 (required): All object and function identifiers shall be declared before use.

Identifiers which represent objects or functions shall always have been declared before they are used, either by a declaration in the code file, or in an included header file.

Note that a declaration provides information to the compiler of the type of an object or function. A definition creates the object or function. For an object, the definition causes the compiler to create storage. For a function, the definition details the body of the function.

If an undeclared function is found, the compiler will make an assumption for the function prototype based on its use. No mechanism exists to check this assumption against the actual definition, and some undefined behaviour may occur.

Rule 21 (required): Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

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

```
SI_16 i;
{
     SI_16 i;  /* This is a different variable */
               /* This is not permitted */
     i = 3;    /* It could be confusing as to which i this refers */
}
```

Rule 22 (advisory): Declarations of objects should be at function scope unless a wider scope is necessary.

The scope of objects should be restricted to functions where possible. File scope should only be used where objects need to have a wider scope than a single function. Where objects are declared at file scope see Rule 23.

As a general principle it is considered good practice to avoid unnecessarily making identifiers global.

7. Rules (continued)

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

[Koenig 56–57]

Declarations at file scope are by default external. Therefore if two files both declare an identifier with the same name at file scope the linker will either give an error, or they will be the same variable, which may not be what the programmer intended. This is also true if one of the variables is in a library somewhere. Use of the *static* storage-class specifier will ensure that identifiers are only visible to the file in which they are declared.

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*.

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

[Undefined 8]

The rules for linkage in C are complicated. However the following example demonstrates how an identifier can be given both internal and external linkage in the same file:

```
static UI_8 x; /* This declaration is at file scope,
                  and the C rules give x internal linkage */

void foo( UI_16 a )
{
     extern UI_16 x; /* This declaration is at block
                        scope, and the C rules give x external linkage */
     /* ... */
}
```

Rule 25 (required): **An identifier with external linkage shall have exactly one external definition.**

[Undefined 44; Koenig 55,63–65]

This precludes the situations where there is no definition of the identifier, and where there is more than one definition of the identifier (perhaps in different files). Multiple definitions in different files can be a problem even if the definitions are the same, and it is obviously serious if they are different, or initialise the identifier to different values. See Rule 20 for an explanation of ‘definition’.

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

[Undefined 10]

Put crudely, ‘compatible types’ means the same type. It would be wrong, for example, to declare an identifier as being an *int*, and to declare the same identifier as also being a *float*. See ISO 9899 [1], section 6.1.2.6, for the full definition of compatible types. This applies whether the object is declared twice in the same file, or whether an external object is declared differently in two different files.

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

[Koenig 66]

Normally this will mean declaring external objects in header files which are then included in all those files which use their objects (including the files which define them). For example:

```
extern SI_16 a;
```

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

```
#include <globs.h>
SI_16 a;
```

If a method exists to ensure that multiple declarations are consistent, then this rule may be relaxed.

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

[Implementation 26]

ISO compilers are not obliged to take any notice of the *register* storage class specifier, so at best it is only a hint to the compiler. A good optimiser should be able to decide for itself what to put in registers.

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

Where a tag has been given in the declaration of a structure, union or enumeration type, all subsequent uses of the tag shall be consistent with the declaration. For example, it would be incorrect to initialise the tag with an initialiser which did not match the structure declared for that tag.

7.8 Initialisation

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

[Undefined 41]

The intent of this rule is that all variables should have been written to before they are read. This does not necessarily require initialisation at declaration, although that is one way of achieving the intent.

Ideally a static check should be performed for any automatic variables which might be used without having first been assigned a value.

If such a check is not possible, then requiring explicit initialisation of automatic variables would be an alternative strategy, though this is not an ideal solution as it permits possibly meaningless initial values to be used in the code.

If relying on initialisation, note that static variables are automatically initialised to zero. Local variables, however, may or may not be automatically initialised, and therefore no reliance should be placed on this mechanism.

Rule 31 (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 think about and show 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:

```
SI_16 y[3][2] = { 1, 2, 3, 4, 5, 6 };              /* incorrect */
SI_16 y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };  /* correct */
```

A similar principle applies to structures, and 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 [1] contains extensive examples of initialisation.

Rule 32 (required): **In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.**

[Implementation 32]

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. For most purposes this will be adequate.

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.

7.9 Operators

See also:	Rules 47, 101, 103

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

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

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

For example:

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

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

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

For example write:

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

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

[Koenig 6]

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

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

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

and not:

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

or even worse:

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

This helps to avoid getting '=' and '==' confused, and assists the static detection of mistakes.

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

[Koenig 48]

The logical operators `&&`, `||` and `!` can be easily confused with the bitwise operators `&`, `|` and `~`.

Static tools may spot possible misuses, but this is not guaranteed, and mistakes should also be checked for at review.

Consideration should be given to implementing a Boolean type within the Style Guidelines, especially if this assists static tools in detecting errors. A Boolean type would be created by *typedef*, and would then be used for any objects which are conceptually Boolean. This could bring many benefits, especially if the static checking tool can support it, and in particular it could help avoid confusion between logical operations and integer operations.

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

[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.

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

[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.

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.

Rule 39 (required): The unary minus operator shall not be applied to an unsigned expression.

Applying the unary minus operator to an unsigned expression is not a particularly meaningful operation to perform, except in a mixed signed/unsigned expression (which itself should be avoided, see comments to Rule 43). In certain circumstances and on some compilers it can produce indeterminate behaviour (for example with *unsigned short int* on a compiler where *short int* and *int* are the same length).

Rule 40 (advisory): The *sizeof* operator should not be used on expressions that contain side effects.

A common 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* should not be used on expressions which contain side effects, as the side effects will not occur.

Rule 41 (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 42 (required): The comma operator shall not be used, except in the control expression of a *for* loop.

Use of the comma operator in situations other than the control expression of a *for* loop is generally detrimental to the readability of code, and the same effect can be achieved by other means.

7.10 Conversions

See also:	Rules 16, 18, 48

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

[Implementation 16; Undefined 15]

C performs many type conversions implicitly and silently, so as to harmonise types within an expression before evaluating it. Some of these conversions can result in loss of information (e.g. conversions to a narrower type). Such implicit conversions shall not be used, but explicit casts should be used instead.

As a general principle, avoid mixing arithmetic of different precisions in the same expression, and avoid mixing signed and unsigned integers in the same expression. Mixed arithmetic normally entails implicit promotions and balancing of types (i.e. conversions), some of which can lead to unexpected behaviour.

However explicit type casting should not be used unnecessarily, as this may prevent static checker tools from detecting some errors. Explicit casts should normally only be used in the case where a conversion which could result in a loss of information is specifically required by the programmer. If the static checking of implicit conversions is overridden by the use of explicit casts in this way, then the programmer should be aware of the issues of truncation and loss of precision associated with the operation, and should provide appropriate checking of values in the code.

Note that types will normally have been defined using specific length *typedef*s (see Rule 13). The lengths of the types will need to be known to determine those situations in which there may be loss of information.

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

Explicit casting between identical types is unnecessary and clutters code. Furthermore it can mask problems if changes are made to the code (e.g. one of the types changes and a conversion with possible loss of information occurs).

The use of casting should be sufficient to cause the calculations required to occur with the desired precision. Unnecessary casting adds the possibility of confusion, and may be such that its interaction with the rules of promotion leads to results other than those expected. Unnecessary casting may also lead to code which is harder to maintain, should the types of variables change.

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

[Implementation 24; Undefined 29,39,40]

Pointer types shall not be cast to other types (including pointer types), nor shall other types be cast to pointers. This can lead to undefined or implementation-defined behaviour, and also circumvents type integrity.

For example, problems can arise if a pointer is converted to an integer type which is not large enough to hold the value, or when an arbitrary integer is converted to a pointer, or if a pointer to an object is converted to a pointer to a different type of object. Pointer to pointer conversion can also provide an unauthorised means of writing to a *const* qualified object.

7.11 Expressions

See also:	Rules 18, 35, 43, 44, 80

Rule 46 (required): **The value of an expression shall be the same under any order of evaluation that the standard permits.**

[Unspecified 7,8; Undefined 18]

Apart from a few operators (notably the function call operator `()`, `&&`, `||`, `?:` and `,` (comma)) the order in which subexpressions are evaluated is unspecified and can vary. This means that no reliance can be placed on the order of evaluation of subexpressions, 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 section 5.1.2.3 of ISO 9899 [1].

A static checking tool may enforce the above rule automatically. If such checks are not available then other measures will be needed to identify expressions which violate the rule. 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++;
x = func( i++, i++ );
```

  The first will give very different results depending on whether `b[i]` is evaluated before `i++` or vice versa. Similarly, the second will give different results depending on which of the function's two parameters is evaluated first. The problem could be avoided by putting the increment operation in a separate statement. The first example could then become:

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

- ***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 which 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()` functions is evaluated first (as an additional side-effect of `pop()` is to modify the stack).

- ***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 = func( y = z / 3 );
```

- ***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. The C compiler 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 SI_16 v;
/* . . . */
x = v;
```

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

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.

Rule 47 (advisory): No 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.

For example, write:

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

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

In an expression, sub-expressions are evaluated at the precision appropriate to the types of the operands. This may be less than the precision of the final result. It is therefore possible to be misled into creating sub-expressions which are evaluated at the wrong precision, which may result in values which are not as the programmer intended.

For example

```
UI_16 i = 1u;
UI_16 j = 3u;

F_64  d0 =         i / j;          /* incorrect   = 0.0   */
F_64  d1 = (F_64) (i / j);         /* incorrect   = 0.0   */
F_64  d2 = (F_64)  i / j;          /* correct     = 0.333 */
F_64  d3 = (F_64)  i / (F_64) j;   /* correct     = 0.333 */

UI_16 i = 65535u;
UI_16 j = 10u;

UI_32 e0 =          i + j;            /* incorrect   = 9     */
UI_32 e1 = (UI_32) (i + j);           /* incorrect   = 9     */
UI_32 e2 = (UI_32)  i + j;            /* correct     = 65545 */
UI_32 e3 = (UI_32)  i + (UI_32) j;    /* correct     = 65545 */
```

Rule 49 (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 data which is representing 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 ( x )      /* Incorrect, unless x is effectively Boolean data
              (e.g. a flag) */
```

Rule 50 (required): **Floating point variables shall not be tested for exact 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:

```
F_32 x, y;
/* some calculations in here */
if ( x == y )
     { /* ... */ }
```

See also Rule 65 and Rule 117.

Rule 51 (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.

For example the following should not be used:

```
#if (1u-2u)
{
     /*...*/
```

7.12 Control Flow

See also:	Rules 42, 82–84

Rule 52 (required): **There shall be no unreachable code.**

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

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

The occurrence of statements (other than null statements) which have no side-effect will normally indicate a programming error, and therefore a static check for such statements shall be performed.

Note that 'null statement' and 'side-effect' are defined in ISO 9899 [1].

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

Null statements should not normally be deliberately included, but where they are used they shall appear on a line by themselves and not with any other text (including comments). 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.

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

[Undefined 5]

Normally the only use for labels, other than in *switch* statements, is as targets for the *goto* statement, which is prohibited by this document (see Rule 56).

Rule 56 (required): **The *goto* statement shall not be used.**

Rule 57 (required): **The *continue* statement shall not be used.**

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

These three rules are in the interests of good structured programming.

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

The statement(s) which are conditionally selected by an *if* or *else* (or *else if*) statement, and the statements within the body of a *while*, *do ... while* or *for* loop, shall always be in a block (enclosed within braces), even if they are a single statement. This avoids the danger of adding code which is intended to be part of the conditional block but is actually not.

For example:

```
if ( test )
{
     x = 1;    /* Even a 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
               gets executed unconditionally */
```

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

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

This rule applies whenever an *if* statement is followed by one or more *else if* statements. In this case the final *else if* should 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 should either take appropriate action or contain a suitable comment as to why no action is taken.

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 */
     errorflag = 1;
}
```

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

Rule 62 (required): **All *switch* statements should contain a final *default* clause.**

Rule 63 (advisory): **A *switch* expression should not represent a Boolean value.**

Rule 64 (required): **Every *switch* statement shall have at least one *case*.**

[Koenig 22–24]

The behaviour of a *switch* statement is potentially confusing, in that normally each case when executed will fall through to the code of the next case. The use of a *break* statement at the end of each *case* clause causes just the code of that clause to be executed. Under Rule 61 this is the only permitted format for a *switch* statement, except in the situation where an empty *case* clause is used to provide multiple cases to execute the same code.

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

For example:

```
switch (x)
{
     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 */
          break;          /* break is required here */
     default:             /* default clause is required */
          errorflag = 1;  /* should be non-empty if possible */
          break;          /* break is a good idea here, in case a
                             future modification turns this into a
                             case clause */
}
```

If the expression in the *switch* statement is effectively representing Boolean data, then in effect it can only take two values, and an *if*, *else* construct is a better way of representing the two-way choice. Thus expressions of this nature should not be used in *switch* statements.

Note also that no code should be placed before the first *case* clause in a *switch* statement, as such code would be unreachable (see Rule 52).

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

A 'loop counter' is one whose value is tested to determine termination of the loop. Floating point variables should 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 50.

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

Rule 67 (advisory): **Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.**

The only expressions within a *for* statement should be to initialise, increment (or decrement) and test loop counter(s), and to test other loop control variables.

In the body of the loop these loop control variables should not be modified. However 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;      /* Permitted - allows early termination of loop */
     i = i + 3;     /* Incorrect - altering the loop counter */
}
```

7.13 Functions

See also:	Rules 2, 3, 20, 26, 93, 104, 105

Rule 68 (required): **Functions shall always be declared at file scope.**

[Undefined 36]

Declaring functions at block scope may be confusing, and can lead to undefined behaviour. The rule represents normal practice.

Rule 69 (required): **Functions with variable numbers of arguments shall not be used.**

[Unspecified 15; Undefined 25,45,61,70–76; Koenig 62]

There are a lot of potential problems with this feature and it shall not be used. This precludes the use of *stdarg.h*, *va_arg*, *va_start* and *va_end*, and the ellipsis notation in function prototypes.

Rule 70 (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 71 (required): **Functions shall always have prototype declarations and the prototype shall be visible at both the function definition and call.**

Rule 72 (required): **For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.**

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

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

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

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

[Undefined 22–24; Koenig 59–62]

Hatton [2] has concluded from statistical evidence that approximately 26% of all statically detectable faults in C code could be prevented by proper use of the function prototype mechanism. 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).

The above rules thus require the use of prototypes (i.e. function declarations), and require the prototype to be visible when the function is defined and whenever it is called.

The types of the parameters and return values in the prototype and the definition must match.

If identifiers are given for the parameters in the prototype, then they shall be same names as those used for the parameters in the function definition. It is recommended that parameter names be given, but for reasons of maintainability some programmers may prefer not to name parameters.

Functions shall always be declared with a return type, that type being *void* if the function does not return any data. Similarly, if the function has no parameters, the parameter list should 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 );
```

The recommended way for implementing function prototypes 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 which need the prototype (see Rule 27).

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

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

[Undefined 22]

The use of function prototypes enforces a much more robust function calling mechanism than if they are not used. See text on Rules 71–76 for more information. The function call shall comply with the function prototype in terms of the number of parameters and their types. 'Compatible types' are defined in ISO 9899 [1], section 6.1.2.6. However Rule 77 permits the types of parameter and prototype to differ in type qualification. This is intended to permit the use of *const* qualification. In other words, for example, an *int* variable can be passed as a parameter defined in the function prototype to be *int* or *const int*, but not as a parameter declared to be *long*.

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

[Undefined 17]

This results in a void expression, the behaviour of which is undefined.

Rule 80 (required): **Void expressions shall not be passed as function parameters.**

[Undefined 21]

Do not attempt to pass any variable or expression of type *void* as a function parameter. This can leave the lvalue undefined and behaviour may be unpredictable.

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

This rule helps to prevent the modification of a function parameter which is intended as input only, and also serves to indicate to the reader that this is the intention.

This rule only applies to parameters which are passed by reference (i.e. a pointer is passed). The *const* qualification should be applied to the object pointed to, not to the pointer, since it is the object itself which is being protected. For example:

```
void myfunc( const SI_16 * myparam )
{
     *myparam = 2;  /* Attempt to modify value pointed to by myparam.
                       The const in the parameter type prevents this */
     return;
}
```

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

This means that a function will normally have a single exit at the end of the function.

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

i) there shall be one *return* statement for every exit branch (including the end of the program) ,

ii) each *return* shall have an expression,

iii) the *return* expression shall match the declared return type.

[Undefined 43; Koenig 65,66]

In functions which have a non-void return type, every exit point must use a *return* statement (i.e. this includes the end of the function code, as well as any other return points), and each *return* statement within the function must have an expression of the type of the function. This expression gives the value which the function returns. The absence of a *return* with an expression leads to undefined behaviour (in other words the compiler may not give an error).

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

[Koenig 65,66]

If any *return* statements are used in a function which returns a void type, expressions shall not be placed after the *return*.

Rule 85 (advisory): **Functions called with no parameters should have empty parentheses.**

[Koenig 24]

Otherwise they will have no effect. For a function `f` which returns a logical value, if the programmer accidentally writes:

```
if (f)
{
     /* ... */
```

instead of

```
if (f())
{
    /* ... */
```

then the statement will not do as intended. `f` will return a pointer to the function, which will be non-zero, and so the test will always be true.

Rule 86 (advisory): **If a function returns error information, then that error information should 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 should 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 117). Note also that the use of *errno* (to return error information from functions) is not robust and shall not be used (see Rule 119).

7.14 Pre-processing Directives

See also:	Rule 7

Rule 87 (required): ***#include* statements in a file shall only be preceded by other pre-processor directives or comments.**

[Undefined 56]

All the *#include* statements in a particular code file should be grouped together near the head of the file. It is important to prevent the situation of executable code coming before a *#include* directive, otherwise there is a danger that the code may try to use items which would be defined in the header. To this end, the rule states that the only items which may precede a *#include* in a file are other pre-processor directives or comments.

Rule 88 (required): **Non-standard characters shall 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 pre-processing token, then the behaviour is undefined.

Rule 89 (required): **The *#include* directive shall be followed by either a <filename> or "filename" sequence.**

[Undefined 48]

The directive shall take one of the following two forms:

```
#include "filename"
#include <filename>
```

Rule 90 (required): **C macros shall only be used for symbolic constants, function-like macros, type qualifiers and storage class specifiers.**

[Koenig 82]

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, nor to redefine the syntax of the language.

For example:

```
/* The following are allowed */
#define PI 3.14159f             /* Constant */
#define PLUS2(X) ((X) + 2)      /* Function-like macro */
#define STOR extern             /* storage class specifier */

/* the following are NOT allowed */
#define SI_32 long              /* use typedef instead */
#define STARTIF if(             /* very bad */
```

Rule 91 (required): **Macros shall not be *#define*'d and *#undef*'d within a block.**

While it is legal C to place *#define* and *#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 92).

Rule 92 (advisory): ***#undef* should 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 93 (advisory): **A function should be used in preference to a function-like macro.**

While function-like macros can provide a speed advantage over functions, functions provide a 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 94 (required): **A function-like macro shall not be 'called' without all of its arguments.**

Rule 95 (required): **Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.**

[Undefined 49,50]

If one or more arguments to the function-like macro are not supplied when it is called, then the behaviour is undefined. Similarly, if any of the arguments look like pre-processor directives, the behaviour when macro substitution is made can be unpredictable.

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

[Koenig 78]

Within a definition of a function-like macro, the arguments shall always be enclosed in parentheses, and the whole expression shall always 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 pre-processor 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.

Similarly, consider what happens if the second definition of *abs* is substituted into the expression:

```
z = abs(a) + 1;
```

giving:

```
z = a > 0 ? a : -a + 1;
```

This is clearly not as intended. Enclosing the entire expression in parentheses in the macro definition avoids this problem.

Rule 97 (advisory): **Identifiers in pre-processor directives should be defined before use.**

If an attempt is made to use an identifier in a pre-processor directive, and that identifier has not been defined, the pre-processor will sometimes not give any warning but will assume a value (usually zero). This is particularly true in tests such as *#if*. 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 pre-processing 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 98 (required): **There shall be at most one occurrence of the # or ## pre-processor operators in a single macro definition.**

[Unspecified 12]

There is an issue of unspecified order of evaluation associated with the # and ## pre-processor operators. To avoid this problem only one occurrence of one operator may be used in any single macro definition (i.e. one #, OR one ## OR neither).

Rule 99 (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 100 (required): **The *defined* pre-processor operator shall only be used in one of the two standard forms.**

[Undefined 47]

The only two permissible forms for the *defined* pre-processor operator are:

defined (identifier *)*
defined identifier

Any other form may produce undefined behaviour.

7.15 Pointers and Arrays

See also:	Rule 45

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

[Undefined 30,31]

This rule refers to explicitly calculating pointer values. Any such pointer value then has the potential to access unintended or invalid memory addresses. This is an area where there is a danger of serious errors occurring in the code at run time. Pointers may go out of bounds of arrays or structures, or may even point to effectively arbitrary locations. There is a particular danger with complex memory models.

If pointer arithmetic is necessary, then appropriate reviews and/or dynamic checking code will be required to ensure that the resulting pointer cannot point to unintended or invalid memory addresses (see also Rule 4). Where such use is necessary the preferred method is to only increment and decrement pointers using ++ or --, rather than performing other arithmetic operations on the pointer.

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

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.

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

[Undefined 33]

Attempting to make comparisons between pointers is fraught with difficulty, and in particular will produce undefined behaviour if the two pointers do not point to the same object. Note that 'relational operators' do not include == and !=.

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

[Unspecified 9; Undefined 27,28]

This includes explicit casts to or from pointers to functions. 'Non-constant' means pointers whose value is calculated at run-time. If the value of a pointer to a function is known at compile time, either because it is a constant or because it is copied from some constant value or constant look-up table, then such use is permitted.

There is a danger with pointers calculated at run time that an error will cause the pointer to point to an arbitrary location. Furthermore pointers to functions cause problems with dependence on the order in which function-designator and function arguments are evaluated when calling functions.

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

[Undefined 27,28]

This is because pointers to functions can easily violate type integrity, so all pointed-to functions should be of identical type.

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

[Undefined 9,26]

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

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

[Koenig 35]

Where a function returns a pointer and that pointer is subsequently de-referenced the program should first check that the pointer is not *NULL*.

7.16 Structures and Unions

See also:	Rule 29, 31

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

[Undefined 35]

If this rule is not followed it leads to an incomplete type, which should be avoided.

Rule 109 (required): **Overlapping variable storage shall not be used.**

[Undefined 34,55; Implementation 27]

This rule refers to the technique of using memory to store some data, then using the same memory to store some other data at some other time during the execution of the program. This might be achieved by the use of unions or by various other mechanisms. 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; 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.

Rule 110 (required): **Unions shall not be used to access the sub-parts of larger data types.**

[Undefined 34,55]

One mechanism in C for accessing the sub-parts of larger data structures (e.g. the individual bits of an I/O port) is to overlay one data structure on another using unions. For example a structure containing bit fields might be overlaid on an integer.

This is not a secure mechanism for achieving the desired aim, because it assumes too much about the way in which the different data types are stored. This is a particular problem where bit fields are used because of the poorly defined aspects of bit fields described under Rules 111–113.

The preferred method of achieving the same effect is via the use of unsigned integer types and masks.

If it is necessary to deviate from this rule, then the justification will need to demonstrate that the alignment and packing of the relevant data types is determinate, consistent and understood. If the compiler has a switch to force bit fields to follow a particular layout then this could assist in such a justification.

Rule 111 (required): **Bit fields shall only be defined to be of type *unsigned int* or *signed int*.**

Rule 112 (required): **Bit fields of type *signed int* shall be at least 2 bits long.**

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

[Unspecified 10; Undefined 37,38; Implementation 29–31]

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:

1. To access the individual bits, or groups of bits, in larger data types (in conjunction with unions). This use is not permitted (see Rule 110).
2. 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 the bit fields are stored within the structure.

It is recommended that structures are declared specifically to hold the sets of bit fields, and do not include any other data within the same structure. Name all the bit fields in the structure (even ones which are there for padding, though padding should not be necessary), and only declare the bit fields to be of type *unsigned int* or *signed int*. Note that Rule 13 need not be followed in defining bit-fields, since their lengths are specified in the structure.

For example the following is acceptable:

```
struct message       /* Struct is for bit-fields only */
{
    signed int    little: 4;
    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).
- If a type other than *signed int*, *unsigned int* or *int* is used, then the behaviour is also implementation-defined.

7.17 Standard Libraries

See also:	Rule 15

Rule 114 (required): Reserved words and standard library function names shall not be redefined or undefined.

[Undefined 54,57,58,62]

It is generally bad practice to *#undef* or *#define* names which are C reserved words or which are function names in any of the standard libraries. In addition 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 92 regarding the use of *#undef*.

ISO C reserved identifiers and function names are documented in the standard [1], and should also be documented by the compiler writer.

Rule 115 (required): Standard library function names shall not be reused.

Where new versions of standard functions are produced by the programmer (e.g. enhanced functionality or checks of input values) the modified function shall have a new name. This is to avoid any confusion as to whether a standard 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.

With add-on libraries, be aware that as well as the function names presented to the user there may also be functions internal to the library which are invisible to the user. Normally, under ISO C, these functions will have 'hidden' names which the programmer will not inadvertently conflict with, but care should be exercised in this area.

Rule 116 (required): All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

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.

For more information on validation of software see the MISRA Guidelines [4].

Rule 117 (required): The validity of values passed to library functions shall be checked.

[Undefined 60,63; Implementation 45]

Many functions in the standard C libraries are not required by the ISO standard [1] 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, or in-house libraries).

Examples of functions which 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 a 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 which 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 which has a singularity at zero, it may be appropriate to perform a test of equality to zero. This is an acceptable exception to Rule 50 without the need to raise a deviation. 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 118 (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.

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

[Implementation 46]

errno is a facility of C which in theory should be useful, but which in practice is poorly defined by the standard. 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 117).

Rule 120 (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 121 (required): **<locale.h> and the *setlocale* function shall not be used.**

[Locale 1–6]

This means that the locale shall not be changed from the standard C locale.

Rule 122 (required): **The *setjmp* macro and the *longjmp* function shall not be used.**

[Unspecified 14; Undefined 64–67]

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

Rule 123 (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 124 (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 125 (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. They are unlikely to be required in an embedded system.

Rule 126 (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 127 (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.

8. References

[1] ISO/IEC 9899 : 1990, *Programming languages - C*, ISO, 1990 (with Technical corrigendum 1 - 1995)

[2] Hatton L. Safer C - Developing Software for High-integrity and Safety-critical Systems, McGraw-Hill, 1994.

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

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

[5] Koenig A. *C Traps and Pitfalls*, Addison-Wesley, 1988.

[6] *Use of COTS Software in Safety Related Applications*, CRR80, HSE Bookshops

[7] ISO 9001:1994, *Model for quality assurance in design, development, production, installation and servicing*, 1994

[8] ISO 9000-3:1997, *Guidelines for the application of ISO 9001 to the development, supply and maintenance of software*, 1997

[9] The TickIT Guide, *A Guide to Software Quality Management System Construction and Certification using ISO 9001*, TickIT, Issue 4.0, 1998

[10] Straker D. *C - Style Standards & Guidelines*, ISBN 0 1311689 8 3, Prentice Hall 1992

[11] Fenton N.E. and Pfleeger S.L. *Software Metrics A Rigorous and Practical Approach*, International Thomson Computer Press 1996 (also available in an earlier edition as Fenton N.E. *Software Metrics A Rigorous Approach*, Chapman & Hall 1992)

[12] MISRA *Software Metrics Report*, Motor Industry Research Association, Nuneaton, November 1994

[13] MISRA *Verification & Validation Report*, Motor Industry Research Association, Nuneaton, November 1994

[14] Kernighan B.W. and Ritchie D.M. *The C programming language*, second edition, Prentice Hall, 1988 (note that the first edition is not a suitable reference document as it does not describe ANSI/ISO C)

[15] ISO/IEC 10646-1, *Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Mutilingual Plane*, 1993

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

Appendix A

Appendix A: Summary of rules

This appendix gives a summary of all the rules in section 7 of this document.

Environment

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

2 (adv) Code written in languages other than C should only be used if there is a defined interface standard for object code to which the compilers/assemblers for both languages conform.

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

4 (adv) Provision should be made for appropriate run-time checking.

Character Sets

5 (req) Only those characters and escape sequences which are defined in the ISO C standard shall be used

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

7 (req) Trigraphs shall not be used.

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

Comments

9 (req) Comments shall not be nested.

10 (adv) Sections of code should not be ‘commented out’.

Identifiers

11 (req) Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

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

Types

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

14 (req) The type *char* shall always be declared as *unsigned char* or *signed char*.

15 (adv) Floating point implementations should comply with a defined floating point standard.

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

17 (req) *typedef* names shall not be reused.

Constants

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

19 (req) Octal constants (other than zero) shall not be used.

Declarations and Definitions

20 (req) All object and function identifiers shall be declared before use.

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

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

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

25 (req) An identifier with external linkage shall have exactly one external definition.

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

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

28 (adv) The *register* storage class specifier should not be used.

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

Initialisation

30 (req) All automatic variables shall have been assigned a value before being used.

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

32 (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.

Operators

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

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

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

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

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

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

39 (req) The unary minus operator shall not be applied to an unsigned expression.

40 (adv) The *sizeof* operator should not be used on expressions that contain side effects.

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

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

Conversions

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

44 (adv) Redundant explicit casts should not be used.

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

Expressions

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

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

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

49 (adv) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean

50 (req) Floating point variables shall not be tested for exact equality or inequality.

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

Control Flow

52 (req) There shall be no unreachable code.

53 (req) All non-null statements shall have a side-effect.

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

55 (adv) Labels should not be used, except in *switch* statements.

56 (req) The *goto* statement shall not be used.

57 (req) The *continue* statement shall not be used.

58 (req) The *break* statement shall not be used (except to terminate the cases of a *switch* statement).

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

60 (adv) All *if*, *else if* constructs should contain a final *else* clause.

61 (req) Every non-empty *case* clause in a *switch* statement shall be terminated with a *break* statement.

62 (req) All *switch* statements should contain a final *default* clause.

63 (adv) A *switch* expression should not represent a Boolean value.

64 (req) Every *switch* statement shall have at least one *case*.

65 (req) Floating point variables shall not be used as loop counters.

66 (adv) Only expressions concerned with loop control should appear within a *for* statement.

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

Functions

68 (req) Functions shall always be declared at file scope.

69 (req) Functions with variable numbers of arguments shall not be used.

70 (req) Functions shall not call themselves, either directly or indirectly.

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

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

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

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

76 (req) Functions with no parameters shall be declared with parameter type *void*.

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

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

79 (req) The values returned by *void* functions shall not be used.

80 (req) Void expressions shall not be passed as function parameters.

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

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

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

84 (req) For functions with void return type, *return* statements shall not have an expression.

85 (adv) Functions called with no parameters should have empty parentheses.

86 (adv) If a function returns error information, then that error information should be tested.

Pre-processing Directives

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

88 (req) Non-standard characters shall not occur in header file names in *#include* directives.

89 (req) The *#include* directive shall be followed by either a <filename> or "filename" sequence.

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

91 (req) Macros shall not be *#define*'d and *#undef*'d within a block.

92 (adv) *#undef* should not be used.

93 (adv) A function should be used in preference to a function-like macro.

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

95 (req) Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.

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

97 (adv) Identifiers in pre-processor directives should be defined before use.

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

99 (req) All uses of the *#pragma* directive shall be documented and explained.

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

Pointers and Arrays

101 (adv) Pointer arithmetic should not be used.

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

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

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

105 (req) 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.

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

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

Structures and Unions

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

109 (req) Overlapping variable storage shall not be used.

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

111 (req) Bit fields shall only be defined to be of type *unsigned int* or *signed int*.

112 (req) Bit fields of type *signed int* shall be at least 2 bits long.

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

Standard Libraries

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

115 (req) Standard library function names shall not be reused.

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

118 (req) Dynamic heap memory allocation shall not be used.

119 (req) The error indicator *errno* shall not be used.

120 (req) The macro *offsetof*, in library <stddef.h>, shall not be used.

121 (req) <locale.h> and the *setlocale* function shall not be used.

122 (req) The *setjmp* macro and the *longjmp* function shall not be used.

123 (req) The signal handling facilities of <signal.h> shall not be used.

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

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

126 (req) The library functions *abort*, *exit*, *getenv* and *system* from library <stdlib.h> shall not be used.

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

Appendix B

Appendix B: Cross references to the ISO standard

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

B1. Rule numbers to ISO 9899 references

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

Appendix B (continued)

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

Appendix B (continued)

B2. ISO 9899 references to rule numbers

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