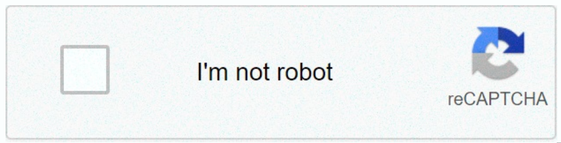
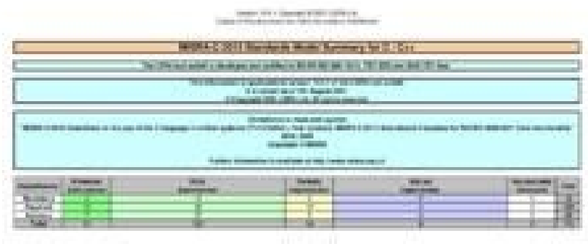


I'm not robot!

Misra-c 2012 pdf

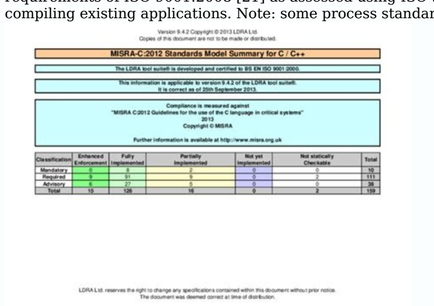
File loading please wait... MISRA C:2012 Guidelines for the use of the C language in critical systems March 2013 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 First published March 2013 by MIRA Limited Watling Street Nuneaton Warwickshire CV10 0TU UK www.misra.org.uk © MIRA Limited 2013. “MISRA”, “MISRA C” and the triangle logo are registered trademarks of MIRA Limited, held on behalf of the MISRA Consortium. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical or photocopying, recording or otherwise without the prior written permission of the Publisher. ISBN 978-1-906400-10-1 paperback ISBN 978-1-906400-11-8 PDF British Library Cataloguing in Publication Data A catalogue record for this book is available from the British Library This copy of MISRA C:2012 Guidelines for the use of the C language in critical systems is issued to LEE CHING MIN. The file must not be altered in any way. No permission is given for distribution of this file. This includes but is not exclusively limited to making the copy available to others by email, placing it on a server for access by intra- or inter-net, or by printing and distributing hardcopies. 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. The published paper document should be taken as authoritative. Information is available from the MISRA web site on how to purchase printed copies of the document. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 MISRA C:2012 Guidelines for the use of the C language in critical systems March 2013 i Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 MISRA Mission Statement We provide world-leading, best practice guidelines for the safe application of both embedded control systems and standalone software. MISRA, The Motor Industry Software Reliability Association, is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety-related embedded electronic systems and other software-intensive applications. To this end, MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners. www.misra.org.uk Disclaimer Adherence to the requirements of this document does not in itself ensure error-free robust software or guarantee portability and re-use. Compliance with the requirements of this document, or any other standard, does not of itself confer immunity from legal obligations. ii Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Foreword At first sight, this third revision of the MISRA C Guidelines may seem somewhat daunting.



Since it is roughly twice the size of the previous revision, one might think that it contains twice as many guidelines, and that compliance with those guidelines might take twice as much effort. In fact, the increase in the number of guidelines is relatively modest at around 10%. The remainder of the increase in size is due to improvements in the guidance given, such as: • Better rationales for guidelines; • More precise descriptions; • Code examples, showing compliance and non-compliance, for most of the guidelines; • More detailed guidance on compliance checking, and the deviation procedure; • Checklists that can be used to support a compliance statement. Finally, I would like to draw attention to the introductory sections of the document. These not only contain practical guidance on how to use MISRA C but, at the same time, have been made more concise than their predecessors. I encourage all users to familiarize themselves with this material. Steve Montgomery MA (Cantab), PhD Chairman, MISRA C Working Group iii Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Acknowledgements The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document: Dave Banham Rolls-Royce plc (previously of Als tom Grid) Andrew Banks Intuitive Consulting Mark Bradbury Aero Engine Controls Paul Burden Programming Research Ltd Mark Dawson-Butterworth Zytek Automotive Ltd Mike Hennell LDRA Ltd Chris Hills Phaedrus Systems Ltd Steve Montgomery Ricardo UK Ltd Chris Tapp LDRA Ltd (also Keylevel Consultants Ltd) Liz Whiting LDRA Ltd (previously of QinetiQ plc) The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process: Roberto Bagnara William Forbes Voilmy Laurent Koki Onoda John Bailey Takao Futagami Fred Long Paulo Pinheiro Johan Bezem Jim Gimpel Daniel Lundin Mohanraj Ragupathi Gunter Blache Gilles Goulas Gavin McCall Paul Rigas Michael Burke Wolfgang von Hansen Douglas Mearns Andrew Scholan Andrew Burnard Takahiro Hashimoto Svante Möller Marco Sorich Paul Butler Dave Higham Frederic Mondot Takuji Takuma Mirko Conrad Shinya Ito Jürgen Mottok Martin Thompson David Cozens David Jennings Yannick Moy Takafumi Wakita David Crocker Peter Jesty Alexander Much David Ward Greg Davis Grzegorz Konopko Robert Mummé Tetsuhiro Yamamoto Manoj Dwivedi Taneli Korhonen Tadanori Nakagawa Naoki Yoshikawa Carl Edmunds Joel Kuehner Greg Newman Achim Olaf Zacher Particular thanks are due to David Crocker for his significant contribution towards the development of Appendix H. The descriptions of implementation-defined behaviours in Appendix G have been reproduced from versions of the ISO Standards published by BSI Standards Limited; the text is identical to that in the ISO versions. Permission to reproduce extracts from British Standards is granted by the BSI Standards Limited (BSI) under Licence No. 2013ET0003. No other use of this material is permitted. British Standards can be obtained in PDF or hard copy formats from the BSI online shop: www.bsigroup. com/Shop or by contacting BSI Customer Services for hard copies only: Tel: +44 20 8996 9001, Email: . DokuWiki was used extensively during the drafting of this document. Our thanks go to all those involved in its development. iv This document was typeset using Open Sans. Open Sans is a trademark of Google and may be registered in certain jurisdictions. Digitized data copyright © 2010–2011, Google Corporation. Licensed under the Apache License, Version 2.0. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Contents 1 The vision 1 2 Background to MISRA C The popularity of C 2.1 2.2 Disadvantages of C 2 2 2 3 Tool selection 3.1 The C language and its compiler 3.2 Analysis tools 4 4 5 4 Prerequisite knowledge 4.1 Training 4.2 Understanding the compiler 4.3 Understanding the static analysis tools 6 6 6 6 5 Adopting and using MISRA C 5.1 Adoption 5.2 Software development process 5.3 Compliance 5.4 Deviation procedure 5.5 Claiming compliance 8 8 8 9 11 12 6 Introduction to the guidelines 6.1 Guideline classification 6.2 Guideline categories 6.3 Organization of guidelines 6.4 Redundancy in the guidelines 6.5 Decidability of rules 6.6 Scope of analysis 6.7 Multi-organization projects 6.8 Automatically generated code 6.9 Presentation of guidelines 6.10 Understanding the source references 13 13 13 14 14 14 15 15 16 17 18 7 Directives 7.1 The implementation 7.2 Compilation and build 7.3 Requirements traceability 7.4 Code design 21 21 23 23 24 v Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 8 9 Rules 8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9 8.10 8.11 8.12 8.13 8.14 8.15 8.16 8.17 8.18 8.19 8.20 8.21 8.22 A standard C environment Unused code Comments Character sets and lexical conventions Identifiers Types Literals and constants Declarations and definitions Initialization The essential type model Pointer type conversions Expressions Side effects Control statement expressions Control flow Switch statements Functions Pointers and arrays Overlapping storage Preprocessing directives Standard libraries Resources References 37 37 39 45 46 48 58 59 63 75 81 93 103 108 115 122 130 136 143 153 155 165 172 178 Appendix A Summary of guidelines 180 Appendix B Guideline attributes 189 Appendix C Type safety issues with C 193 Appendix D Essential types 196 Appendix E Applicability to automatically generated code 202 Appendix F Process and tools checklist 205 Appendix G Implementation-defined behaviour checklist 206 Appendix H Undefined and critical unspecified behaviour 210 Appendix I Example deviation record 220 Appendix J Glossary 223 vi Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 1 The vision The MISRA C Guidelines define a subset of the C language in which the opportunity to make mistakes is either removed or reduced. Many standards for the development of safety-related software require, or recommend, the use of a language subset, and this can also be used to develop any application with high integrity or high reliability requirements. As well as defining this subset, these MISRA C Guidelines provide: • Educational material for those developing C programs; • Reference material for tool developers. Previous editions of MISRA C have been based on the 1990 ISO definition of C. As the 1999 ISO definition has now been adopted, in varying degrees, by embedded implementations it was considered that the time was right to publish a new edition of MISRA C which recognized the 1999 ISO definition. Each aspect of the guidance presented in the previous edition has been comprehensively reviewed and improved where appropriate. This third edition also incorporates material created in response to the feedback that has been provided by users of earlier editions of the Guidelines. A major change in this third edition is the development of the second edition’s concept of underlying type into the essential type. Using the new essential type concept, it has been possible to develop a set of guidelines that bring stronger typing to the C language. The vision for the third edition of MISRA C is therefore to: • Adopt the 1999 ISO definition of the C language, while retaining support for the older 1990 definition; • Correct any known issues with the second edition; • Add new guidelines for which there is a strong rationale; • Improve the specification and the rationale for existing guidelines; • Remove any guidelines for which the rationale is insufficient; • Increase the number of guidelines that can be processed by static analysis tools; • Provide guidance on the applicability of the guidelines to automatically-generated code. 1 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 2 Background to MISRA C 2.1 The popularity of C The C programming language is popular because: • C compilers are readily available for many processors; • C programs can be compiled to efficient machine code; • It is defined by an international standard; • It provides mechanisms to access the input/output capabilities of the target processor, whether directly or by means of language extensions; • There is a considerable body of experience with using C in critical systems; • It is widely supported by static analysis and test tools. 2.2 Disadvantages of C While popular, the language has several drawbacks which are discussed in the following sub-sections. 2.2.1 Language definition The ISO Standard does not specify the language completely but places some aspects under the control of an implementation.

Contents

This is intentional, partly because of the desire to support many preexisting implementations for widely different target processors. As a result there are areas of the language in which; • The behaviour is undefined; • The behaviour is unspecified; • An implementation is free to choose its own behaviour provided that it is documented. A program that relies on undefined or unspecified behaviour is not necessarily guaranteed to behave in a predictable manner. A program that places excessive reliance on implementation-defined behaviour may be difficult to port to a different target. The presence of implementation-defined behaviour may also hinder static analysis if it is not possible to configure the analyser to handle it. 2.2.2 Language misuse While C programs can be laid out in a structured and comprehensible manner, C makes it easy for programmers to write obscure code that is difficult to understand. The specification of the operators makes it difficult for programming errors to be detected by a compiler. For example, the following two fragments of code are both perfectly legal so it is impossible for a compiler to know whether one has been mistakenly used in place of the other: if (a == b) if (a = b) /* tests whether a and b are equal */ /* assigns b to a and tests whether a is non-zero */ 2 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 There are areas of the language that are commonly misunderstood by programmers. For example, C has more operators than some other languages and consequently has a high number of different operator precedence levels, some of which are not intuitive. The type rules provided by C can also be confusing to programmers who are familiar with stronglytyped languages. For example, operands may be “promoted” to wider types, meaning that the type resulting from an operation is not necessarily the same as that of the operands. 2.2.4 Run-time error checking C programs can be compiled into small and efficient machine code, but the trade-off is that there is a very limited degree of run-time checking.
C programs generally do not provide run-time checking for common problems such as arithmetic exceptions (e.g. divide by zero), overflow, validity of pointers, or array bound errors. The C philosophy is that the programmer is responsible for making such checks explicitly. Section 2: Background to MISRA C 2.2.3 Language misunderstanding 3 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 3 Tool selection 3.1 The C language and its compiler When work started on this document, the ISO standard for the C language was ISO/IEC 9899:1999 [8] as corrected by [9], [10] and [11]. This version of the language is hereafter referred to as C99. However, many compilers are still available for its predecessor, defined by ISO/IEC 9899:1990 [2] as amended and corrected by [4], [5] and [6], and hereafter referred to as C90. Some compilers provide an option to select between the two versions of the language. The current ISO standard for the C language is ISO/IEC 9899:2011 [13] as corrected by [14]. When this standard was published, work on the MISRA C Guidelines had been nearly completed and it has therefore not been possible to incorporate guidance for this later version. Note: access to a copy of the relevant standard is not necessary for the use of MISRA C, but it may be helpful. The choice between C90 and C99 might be affected by factors such as the amount of legacy code being reused on the project and the availability of compilers for the target processor. The compiler selected for the project should meet the requirements of a conforming freestanding implementation for the chosen version of the C language. It may exceed these requirements, for example by providing all the features of the language (a freestanding implementation need only provide a well-defined subset), or it might provide extensions as permitted by the language standard. Ideally, confirmation that the compiler is indeed conforming should be supplied by the compiler developer, for example by providing details of the conformance tests that were run and the results that were obtained. Sometimes, there may be a limited choice of compilers whose quality may not be known. If it proves difficult to obtain information from the compiler developers, the following steps could be taken to assist in the selection process: • Checking that the compiler developer follows a suitable software development process, e.g. one that meets the requirements of ISO 9001:2008 [21] as assessed using ISO 90003:2004 [22]; • Reading reviews of the compiler and user experience reports; • Reviewing the size of the user base and types of applications developed using the compiler; • Performing and documenting independent validation testing, e.g. by using a conformance test suite or by compiling existing applications. Note: some process standards require qualification of compilers in some situations, e.g. IEC 61508:2010 [32], ISO 26262:2011 [23] and DO-178C [24].


Version 9.4.2 Copyright © 2013 LDRA Ltd.
Copies of this document are not to be made or distributed.

MISRA-C:2012 Standards Model Summary for C / C++

The LDRA tool suite® is developed and certified to BS EN ISO 9001:2000.

This information is applicable to version 9.4.2 of the LDRA tool suite®.
It is correct as of 20th September 2013.

Compliance is measured against
"MISRA C:2012 Guidelines for the use of the C language in critical systems"
2013
Copyright © MISRA

Further information is available at http://www.misra.org.uk

Classification	Enhanced Enforcement	Fully Implemented	Partially Implemented	Not yet Implemented	Not statically Checkable	Total
Mandatory	0	8	2	0	0	10
Required	9	91	9	0	2	111
Advisory	6	27	5	0	0	38
Total	15	126	16	0	2	159

LDRA Ltd. reserves the right to change any specifications contained within this document without prior notice.
The document was deemed correct at time of distribution.

4 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Analysis tools It is possible to check that C source code complies with MISRA C by means of inspection alone. However, this is likely to be extremely time-consuming and error prone.

1 愿景(The vision)

MISRA C 指南定义了 C 语言的子集，以删除或减少犯错的机会。此 C 语言子集被许多用于开发与安全相关的软件的标准要求或建议使用，并且它也可以用于开发具有高完整性或高可靠性要求的任何应用程序。

除了定义此子集外，这些 MISRA C 指南还提供了：

◇ 为正在开发 C 计划的人准备的教育材料；

◇ 工具开发人员参考资料。

以前的 MISRA C 版本是基于 1990 年的 ISO C 定义的。由于现在 1999 年的 ISO 定义已在不同程度上被嵌入式实现所采用，人们认为发布 MISRA C 的新版本的时机已经成熟。 它认可了 1999 年的 ISO 定义。

上一版中提供的指南的各个方面均已进行了全面审查，并在适当时进行了改进。 此第三版还结合了根据早期版本的指南用户提供的反馈而创建的材料。

第三版的主要变化是将第二版的基础类型概念发展为基本类型。 使用新的基本类型概念，有可能开发出一套准则来为 C 语言带来更强的键入性。

第三版 MISRA C 的愿景是：

◇ 采用 1999 年的 C 语言 ISO 定义，同时保留对 1990 年的较早定义的支持；

◇ 更正第二版中的任何已知问题；

◇ 添加有充分理由的新准则；

◇ 改进现有准则的规范和理由；

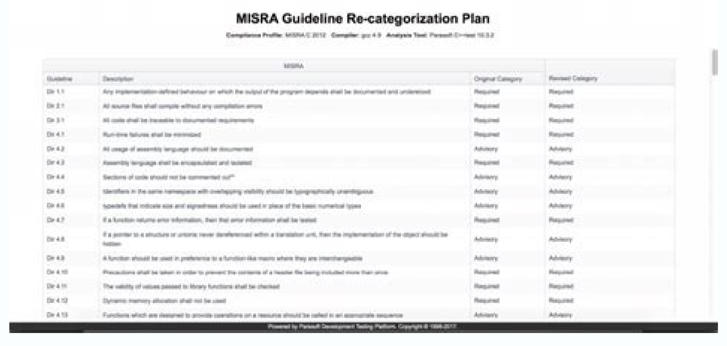
◇ 删除任何理由不足的准则；

◇ 增加静态分析工具可以处理的准则数量；

◇ 提供有关准则对自动生成代码的适用性的指南。

3

Any realistic process for checking code against MISRA C will therefore involve the use of at least one static analysis tool. All of the factors that apply to compiler selection apply also to the selection of static analysis tools, although the validation of analysis tools is a little different from that of compilers.
An ideal static analysis tool would: • Detect all violations of MISRA C guidelines; • Produce no “false positives”, i.e. would only report genuine violations and would not report non-violations or possible violations. Section 3: Tool selection 3.2 For the reasons explained in Section 6.5, it is not, and never will be, possible to produce a static analysis tool that meets this ideal behaviour. The ability to detect the maximum number of violations possible, while minimizing the number of false positive messages, is therefore an important factor in choosing a tool. There is a wide range of tools available with execution times ranging from seconds to days. Broadly speaking, tools that consume less time are more likely to produce false positives than those that consume large amounts of time. Consideration should also be given to the balance between analysis time and analysis precision during tool selection. Analysis tools vary in their emphasis. Some might be general purpose, whereas others might focus on performing a thorough analysis of a subset of potential issues. Thus it might be necessary to use more than one tool in order to maximize the coverage of issues. Note: some process standards, including IEC 61508:2010 [32], ISO 26262:2011 [23] and DO-178C [24], require the qualification of verification tools such as static analysers in some situations. 5 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 4 Prerequisite knowledge 4.1 Training In order to ensure an appropriate level of skill and competence on the part of those who produce 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. Since compilers and static analysis tools are complex pieces of software, consideration should also be given to providing training in their use.



In the case of static analysis tools, it might be possible to obtain training in their use specifically in relation to MISRA C. 4.2 Understanding the compiler In this document, the term “compiler”, referred to as “the implementation” by the ISO C standards [2], [8], means the compiler itself as well as any associated tools such as a linker, library manager and executable file format conversion tools.
The compiler may provide various options that control its behaviour. It is important to understand the effect of these options as their use, or non-use, might affect: • The availability of language extensions; • The conformance of the compiler with the ISO C standards; • The resources, especially processing time and memory space, required by the program; • The likelihood that a defect in the compiler will be exposed, such as might occur when complex highly-optimizing code transformations are performed. It is important to understand the way in which the compiler implements those features of the C language that are termed “implementation-defined” in the ISO C standards. It is also important to understand any language extensions that the compiler may provide.
The compiler developer may maintain a list of defects that are known to affect the compiler along with any workarounds that are available. Knowledge of the contents of this list would clearly be advantageous before starting a project using that compiler. If such a list is not available from the compiler developer then a local list should be maintained whenever a defect, or suspected defect, is discovered and reported to the compiler developer. 4.3 Understanding the static analysis tools The documentation for each static analysis tool being used on a project should be reviewed in order to understand: 6 • How to configure the analyser to match the compiler’s implementation-defined behaviour, e.g. sizes of the integer types; • Which MISRA C guidelines the analyser is capable of checking (Section 5.3); • Whether it is possible to configure the analyser to handle any language extensions that will be used; • Whether it is possible to adjust the analyser’s behaviour in order to achieve a different balance between analysis time and analysis precision. Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 Section 4: Prerequisite knowledge The tool developer may maintain a list of defects that are known to affect the tool along with any workarounds that are available. Knowledge of the contents of this list would clearly be advantageous before starting a project using that tool.
If such a list is not available from the tool developer then a local list should be maintained whenever a defect or suspected defect is discovered and reported to the tool developer. 7 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 5 Adopting and using MISRA C 5.1 Adoption MISRA C should be adopted from the outset of a project. If a project is building on existing code that has a proven track record then the benefits of compliance with MISRA C may be outweighed by the risks of introducing a defect when making the code compliant. In such cases a judgement on adopting MISRA C should be made based on the net benefit likely to be obtained. 5.2 Software development process MISRA C is intended to be used within the framework of a documented software development process. While MISRA C can be used in isolation, it is of greater benefit when used within a process to ensure that other activities have been performed correctly so that, for example: • The software requirements, including any safety requirements, are complete, unambiguous and correct; • The design specifications reaching the coding phase are correct, consistent with the requirements and do not contain any other functionality; • The object modules produced by the compiler behave as specified in the corresponding designs; • The object modules have been tested, individually and together, to identify and eliminate errors. MISRA C should be used by programmers before code is submitted for review or unit testing.
A project that checks for MISRA C compliance late in its lifecycle is likely to spend a considerable amount of time re-coding, re-reviewing and re-testing. It is therefore expected that the software development process will require the early application of MISRA C principles. A full discussion of the requirements for safety-related software development processes is outside the scope of this document. Examples of development processes may be found in standards and guidelines such as IEC 61508:2010 [32], ISO 26262:2011 [23], DO-178C [24], EN 50128:2011 [33] and IEC 62304:2006 [34]. The remainder of this section deals with the interaction between MISRA C and the software development process. 5.2.1 Process activities required by MISRA C In order to use MISRA C, it is necessary to develop and document: • A compliance matrix, showing how compliance with each MISRA C guideline will be checked; • A deviation process by which justifiable non-compliances can be authorized and recorded.
The software development process should also document the steps that will be taken to avoid run-time errors, and to demonstrate that they have been avoided.
For example, it should include descriptions of the processes by which it is demonstrated and recorded that: 8 • The execution environment provides sufficient resources, especially processing time and stack space, for the program; • Run-time errors, such as arithmetic overflow, are absent from areas of the program: for example by virtue of code that checks the ranges of inputs to a calculation. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 It is recognized that a consistent style assists programmers in understanding code written by others. However, since style is a matter for individual organizations, MISRA C does not make any recommendations related purely to programming style. It is expected that local style guides will be developed and used as part of the software development process.
The use of metrics is recommended by many software process standards as a means to identify code that may require additional review and testing effort. However, the nature of the metrics being collected, and their corresponding thresholds, will be determined by the industry, organization and/or the nature of the project. This document, therefore, does not offer any guidance on software metrics. 5.3 Compliance In order to ensure that code complies with all of the MISRA C guidelines, a compliance matrix should be produced. This matrix lists each guideline and indicates how it is to be checked.
For most guidelines, the easiest, most reliable and most cost-effective means of checking will be to use a static analysis tool or tools, the compiler, or a combination of these. Where a guideline cannot be completely checked by a tool, then a manual review will be required. Section 5: Adopting and using MISRA C 5.2.2 Process activities expected by MISRA C Some automatic code generator developers supply a compliance statement along with their tools. This statement identifies those MISRA C guidelines that will not be violated by the generated code provided the tool has been configured as specified by the tool developer. For example it might: • List the guidelines that are not violated; or • State that all mandatory and required guidelines are not violated. The use of an automatic code generator compliance statement can significantly reduce the number of MISRA C guidelines that need to be checked by other means. See Table 1 for an example of a compliance matrix, and see Appendix A for a summary of the guidelines, which could be used to assist in generating a full compliance matrix. Guideline Compilers ‘A’ Checking tools ‘B’ ‘A’ ‘B’ Dir 1.1 Dir 2.1 Manual review Procedure x no errors no errors … Rule 4.1 message 38 Rule 4.2 Rule 5.1 warning 97 warning 347 … Table 1: Example compliance matrix 9 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 5: Adopting and using MISRA C Once a compliance matrix is in place, the compilers and analysers can be configured and used to generate a list of messages that require investigation. The following information should be recorded for each tool used in the checking process: • Version number; • Options used when invoking the tool; • Any configuration data that the tool uses. 5.3.1 Compiler configuration If the compiler provides a choice between C90 and C99, it must be configured for the variant being used on the project. Similarly, if the compiler provides a choice of targets, it must be configured for the correct variant of the selected target. The compiler’s optimization options should be reviewed and selected carefully in order to ensure that an appropriate balance between execution speed and code size has been obtained.
Using more aggressive optimizations may increase the risk of exposing defects in the compiler. Even if the compiler is not being used to ensure compliance, it may produce messages during the translation process that indicate the presence of potential defects. If the compiler provides control over the number and nature of the messages it produces, these should be reviewed and an appropriate level selected. 5.3.2 Static analysis tool configuration While a compiler is usually specific to a particular processor, or family of processors, static analysis tools tend to be general purpose. It is therefore important to configure each static analysis tool to reflect the implementation decisions made by the compiler. For example, static analysers need to know the sizes of the integer types. If the static analyser supports both C90 and C99, it will need to be configured to match the variant in use on the project. If possible, the static analyser should be configured to support language extensions. If this is not possible then an alternative procedure will need to be put in place for checking that the code conforms to the extended language. It may also be necessary to configure a static analyser in order to allow it to check certain guidelines. For example, unless a project is using _Bool to represent Boolean data, an analyser needs to be made aware of the essentially Boolean type (Appendix D) in order to check some MISRA C guidelines. 5.3.3 Investigating messages The messages produced by the compliance checking process, or by the translation process, fall into one of the following categories: 1. Correct diagnosis of a MISRA C guideline violation; 2. Diagnosis of a possible MISRA C guideline violation; 3. False diagnosis of a MISRA C guideline violation; 4. Diagnosis of something that is not a MISRA C guideline violation. 10 The preferred remedy for any messages in category (1) is to correct the source code in order to make it compliant with the MISRA C guideline. If it is undesirable or impossible to render the code MISRA C compliant then a deviation may need to be raised (see Section 5.4). Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 • Explain why, despite diagnosis of a possible violation, the code complies with the MISRA C guideline for category (2) messages; • Explain and, if possible, obtain the tool developer’s agreement that the tool has incorrectly diagnosed a violation of the MISRA C guideline for category (3) messages; • Justify why the message can safely be disregarded for category (4) messages. All records of such investigations should be reviewed and approved by an appropriately qualified technical authority. 5.4 Deviation procedure In some instances it may be necessary to deviate from the guidelines given in this document. For example, a common method of accessing memory-mapped I/O ports at fixed addresses does not comply with MISRA C because it involves converting an integer to a pointer: Section 5: Adopting and using MISRA C Any messages in the other categories should be investigated. Sometimes, the easiest and quickest solution will be to modify the source code to eliminate the message. However, this may not always be possible or desirable, in which case a record of the investigation should be kept. The purpose of the record is to: #define PORT (*(volatile unsigned char *)0x0002) PORT = 0x10u; It is important that such deviations are properly recorded and authorized. Individual programmers can be prevented from deviating guidelines by means of a formal authorization procedure. Formal recordkeeping is good practice and supports any safety argument that might be made for the software. The deviation procedure should be formalized within the software development process. No process is imposed by MISRA C because methods used will vary between organizations, for example: • The physical methods used to raise, review and approve a deviation; • The degree of review and evidence required, which may vary according to the guideline in question, before a deviation can be approved. Each MISRA C guideline is given a category which affects the way in which deviations may be applied (see Section 6.2). Deviations are divided into two categories: Project Deviations and Specific Deviations. A Project Deviation is used when a MISRA C guideline is deviated in a particular class of circumstances.
For example, if memory-mapped I/O is used in a communications driver then it might be appropriate to raise a Project Deviation to cover use of memory-mapped I/O in all files making up that driver. Project Deviations are typically, though not exclusively, raised early in the project. A Specific Deviation is used when a MISRA C guideline is deviated for a single instance in a single file. 11 Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 Section 5: Adopting and using MISRA C A deviation record should include: • The guideline being deviated; • The circumstances in which the deviation is permitted: for Project Deviations this may be a generic description such as “all code in module X” but for Specific Deviations it will identify the location at which the non-compliant code occurs; • Justification for the deviation, including an assessment of the risks associating with deviating compared with other possibilities; • Demonstration of how safety is assured, for example a proof that an unsafe condition cannot occur, together with any additional reviews or tests that are required; • Potential consequences of the deviation and any mitigating actions that have been taken. An example deviation record is provided in Appendix I. 5.5 Claiming compliance Compliance cannot be claimed for an organization, only for a project. When claiming compliance with MISRA C for a project, a developer is stating that evidence exists to show: 1. A compliance matrix has been completed which shows how compliance has been enforced; 2. All the C code in the project is compliant with the guidelines of this document, or is subject to approved deviations; 3. There is a record of each approved deviation; 4.
The guidance given in this section, as well as that given in Section 3 and Section 4, has been adopted; 5. Staff are suitably skilled and have a sufficient range of experience. Note: when claiming that code complies with the guidelines, the assumption is being made that the tools or reviewers have identified all non-compliances. Since other tools or reviewers might identify different non-compliances, it is important to recognize that compliance claims are not absolute, but depend on the checking process used. A summary of the guidance referred to in point (4) is provided in the form of a checklist in Appendix F. 12 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 6 Introduction to the guidelines This section explains the presentation of the guidelines in Section 7 and Section 8, the main content of this document, and serves as an introduction to those sections. 6.1 Guideline classification Every MISRA C guideline is classified as either being a “rule” or a “directive”. A directive is a guideline for which it is not possible to provide the full description necessary to perform a check for compliance. Additional information, such as might be provided in design documents or requirements specifications, is required in order to be able to perform the check. Static analysis tools may be able to assist in checking compliance with directives but different tools may place widely different interpretations on what constitutes a non-compliance. A rule is a guideline for which a complete description of the requirement has been provided. It should be possible to check that source code complies with a rule without needing any other information. In particular, static analysis tools should be capable of checking compliance with rules subject to the limitations described in Section 6.5. Section 7 contains all the directives and Section 8 contains all the rules. 6.2 Guideline categories Every MISRA C guideline is given a single category of “mandatory”, “required” or “advisory”, whose meanings are described below. Beyond this basic classification the document does not give, nor intend to imply, any grading of importance of each of the guidelines. All required guidelines, whether rules or directives, should be considered to be of equal importance, as should all mandatory and advisory ones. 6.2.1 Mandatory guidelines C code which is claimed to conform to this document shall comply with every mandatory guideline — deviation from mandatory guidelines is not permitted. Note: if a checking tool produces a diagnostic message, this does not necessarily mean that a guideline has been violated for the reasons given in Section 6.5. 6.2.2 Required guidelines C code which is claimed to conform to this document shall comply with every required guideline, with a formal deviation required, as described in Section 5.4, where this is not the case. An organization or project may choose to treat any required guideline as if it were mandatory. 6.2.3 Advisory guidelines These are recommendations. However, the status of “advisory” does not mean that these items can be ignored, but rather that they should be followed as far as is reasonably practical. Formal deviation is not necessary for advisory guidelines but, if the formal deviation process is not followed, alternative arrangements should be made for documenting non-compliances. An organization or project may choose to treat any advisory guideline as if it were mandatory or required. 13 Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 Section 6: Introduction to the guidelines 6.3 Organization of guidelines The guidelines are organized under different topics within the C language. However there is inevitably overlap, with one guideline possibly being relevant to a number of topics. Where this is the case the guideline has been placed under the most relevant topic. 6.4 Redundancy in the guidelines There are a few cases within this document where a guideline is given that refers to a language feature that 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 guideline, or by choosing not to follow an advisory guideline. In this case the second guideline, constraining the use of that feature, becomes relevant. 6.5 Decidability of rules Each mandatory, required and advisory rule is classified as decidable or undecidable. This classification describes the theoretical ability of a static analyser to answer the question “Does this code comply with this rule?” The directives are not classified in this way because it is impossible, given only the source code, to devise an algorithm that could guarantee to check for compliance. A rule is decidable if it is possible for a program to answer the question with a “yes” or a “no” in every case and undecidable otherwise. A review of the theory of computation, on which this classification is based, is beyond the scope of this document but a rule is likely to be undecidable if detecting violations depends on run-time properties such as: • The value that an object holds; • Whether control reaches a particular point in the program. Decidable rules have useful properties with regard to static analysis. Provided that a defect-free and complete static analyser is configured correctly: • A reported violation of a decidable rule indicates a real violation; • No reported violation of a decidable rule indicates there are no violations in the code being analysed. Some examples of decidable rules are: • Rule 5.2: depends on the names and scopes of identifiers; • Rule 11.3: depends on the source pointer and destination pointer types; • Rule 20.7: depends on the syntactic form of the result of a macro expansion.
Static analysers vary in their ability to detect violations of undecidable rules: • A reported violation of an undecidable rule may not necessarily indicate a real violation; some analysers take the approach of reporting possible violations to remind users of the uncertainty; • No reported violation of an undecidable rule does not necessarily indicate that there are no violations in the code being analysed. Some examples of undecidable rules are: 14 • Rule 12.2: depends on the value of the right-hand operand of a shift operator; • Rule 2.1: depends on knowing whether execution never reaches a certain point. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 6.6 Scope of analysis Each rule is classified according to the amount of code that needs to be checked in order to detect violations. As for decidability, the concept of analysis scope is not applied to directives.
The analysis scopes that may be applied to rules are “Single Translation Unit” and “System”. If a rule is classified as capable of being checked on a “Single Translation Unit” basis then it is possible to detect all violations within a project by checking each translation unit independently. For example, the presence of switch statements that do not contain default labels (Rule 16.4) within one translation unit has no effect on whether other translation units contain such switch statements. If a rule is classified as needing to be checked on a “System” basis, then identifying violations of a rule within a translation unit requires checking more than the translation unit in question. Rules that are classified as “System” are best checked by analysing all the source code, although it will be possible to identify some violations when checking a subset of the whole source. For example, if a project has two translation units A and B, it is possible to check that all declarations and definitions of an object within each translation unit use the same type names and qualifiers (Rule 8.3). However, this does not guarantee that the declarations and definitions in A use the same type names and qualifiers as those in B. All of the source code that will be compiled and linked into the executable therefore needs to be checked to guarantee compliance with this rule. Section 6: Introduction to the guidelines As indicated in Section 5.3.3, a process should be developed for analysing the results of static analysis and recording the outcome. Particular attention should be paid to the process for analysing any output that relates to undecidable rules. All undecidable rules need to be checked on a “System” basis because, in the general case, information about the behaviour of other translation units will be needed. For example, whether or not the value of the automatic object x is set before x is used (Rule 9.1) in the function g, below, will depend on the behaviour of the function f which is defined in another translation unit: extern void f (uint16_t *p); uint16_t y; void g (void) { uint16_t x; /* x is not given a value f (&x); y = x; */ /* f might modify the object pointed to by its parameter */ /* x may or may not be unset */ } 6.7 Multi-organization projects Projects may involve code from various organizations, for example: • The Standard Library code from the compiler implementer; • Low-level driver code from a device vendor; • Operating system and high-level driver code from a specialist supplier; • Application code which may be shared between collaborating organizations according to their expertise. 15 Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 Section 6: Introduction to the guidelines The Standard Library code is likely to be concerned with efficiency of execution. It may rely on implementation details or unspecified behaviours such as: • Casting pointers to object types into pointers to other object types; • The layout of stack frames, and in particular the locations of function parameters within those frames; • Embedding assembly language statements in C. As it is part of the implementation, and its functionality and interface is defined in the ISO C standard, The Standard Library code is not required to comply with MISRA C. Unless otherwise specified in the individual guidelines, the contents of standard header files, and any files that are included during processing of a standard header file, are not required to comply with MISRA C. However, guidelines that rely on the interface provided by standard header declarations and macros are still applicable. For example, the essential type rules apply to the types of arguments passed to functions specified in The Standard Library and to their results. All other code should comply with the MISRA C guidelines to the greatest extent possible. If all the source code is available then the entire program can be checked for compliance. However, in many projects it is not possible for the developing organization to obtain all the source code. For example, collaborating organizations may share functional specifications, interface specifications and object code but are likely to protect their own intellectual property by not sharing source code. When only part of the source code is available for analysis, other techniques should be applied in order to confirm compliance. For example: • Organizations that supply commercial code may be willing to issue a statement of MISRA C compliance; • Organizations that are collaborating might: - Agree upon a common procedure and tools that each will apply to their own source code; - Supply each other with stub versions of source code to permit cross-organizational checks to be made. A project should define which of the following methods it will use to deal with code supplied by other organizations: 1. In the same manner as code being developed in-house — this relies on having the source code available; 2. In the same manner as The Standard Library — this relies on the organization supplying a suitable MISRA C compliance statement; 3. Compliance checks to be performed on the contents of header files and their interface — this assumes that no compliance statement is available and no source code, other than header files, is available. In case (3), appropriate additional verification and validation techniques should be employed prior to using the code. Note: some process standards may include wider requirements for managing multi-organization developments, for example ISO 26262 [23] Part 8 Clause 5 “Interfaces in distributed development”. 6.8 16 Automatically generated code The MISRA C guidelines are applicable to code that has been generated automatically. Responsibility for compliance lies both with the developer of the automatic code generation tool, and with the developer of the model from which code is being generated. Since there are several modelling packages, each of which may have several automatic code generators, it is not possible to allocate this Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The category assigned to a MISRA C guideline when applied to automatically generated code is not necessarily the same as that when applied to manually generated code. When making this distinction, it is important to note that some modelling packages permit the injection of manually generated code into a model. Such code is not treated as having been generated automatically when determining the applicable category of a guideline.
Appendix E identifies those MISRA C guidelines whose category as applied to automatically generated code is different from that for manually generated code. It also states the documentation requirements that are placed on automatic code generator developers. 6.9 Presentation of guidelines The individual requirements of this document are presented in the following format: Ident Requirement text Section 6: Introduction to the guidelines responsibility individually for each MISRA C guideline. It is expected that users of modelling packages and code generators will employ relevant guidelines such as MISRA AC GMG [17], MISRA AC SLSF [18] and MISRA AC TL [19]. [Source ref] Category Category Analysis Decidability, Scope Applies to Cxx where: • “Ident” is a unique identifier for the guideline; • “Requirement text” is the guideline itself; • “Source ref” indicates the primary source(s) which led to this item or group of items, where applicable. See Section 6.10 for an explanation of the significance of these references, and a key to the source materials; • “Category” is one of “Mandatory”, “Required” or “Advisory”, as explained in Section 6.2; • “Decidability” is one of “Decidable” or “Undecidable”, as explained in Section 6.5; • “Scope” is one of “System” or “Single Translation Unit”, as explained in Section 6.6; • “Cxx” is one or more of “C90” and “C99”, separated by a comma, indicating which versions of the C language the guideline applies to. Note: since directives do not have a decidability or a scope of analysis, the “Analysis” line is omitted for directives.
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 guideline(s), and examples of how to apply the guideline(s). Within the supporting text, there may be a heading titled “Amplification”, followed by text that provides a more precise description of the guideline. An amplification is normative; if it conflicts with the headline, the amplification takes precedence. This mechanism is convenient as it allows a complicated concept to be conveyed using a short headline. Within the supporting text, there may be a heading titled “Exception”, followed by text that describes situations in which the rule does not apply. The use of exceptions permits the description of some Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 17 Section 6: Introduction to the guidelines guidelines to be simplified. It is important to note that an exception is a situation in which the guideline does not apply. Code that complies with a guideline by virtue of an exception does not require a deviation. 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 “Portability Issues” listed in The Standard, then the original issue raised in it may provide additional help in understanding the guideline. Within the guidelines, and their supporting text, the following font styles are used to represent C keywords and C code: • C keywords appear in italic text; • Items defined in the Glossary appear in italic text; • C code appears in a monospaced font, either within other text; • Or as separate code fragments. Note: 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 Dir 4.6): uint8_t uint16_t uint32_t int8_t int16_t int32_t float32_t float64_t /* /* /* /* /* /* /* /* unsigned 8-bit integer unsigned 16-bit integer unsigned 32-bit integer signed 8-bit integer signed 16-bit integer signed 32-bit integer 32-bit floating-point 64-bit floating-point */ */ */ */ */ */ */ */ Non-specific object names are constructed to give an indication of the type. For example: uint8_t int16_t int32_t bool_t enum atag char float32_t float64_t u8a; s16b; s32c; bla; ena; chb; f32a; f64b; /* /* /* /* /* /* /* /* 8-bit unsigned integer 16-bit signed integer 32-bit signed integer essentially Boolean enumerated type character 32-bit floating-point 64-bit floating-point */ */ */ */ */ */ */ */ 6.10 Understanding the source references Where a guideline originates from one or more published sources these are indicated in square brackets after the guideline. 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 guideline (for example when considering a request for a deviation). Secondly, with regard to Portability Issues described in the ISO C standard, the form of the source gives extra information about the nature of the problem.
Rules which do not have a source reference may have originated from a contributing company’s inhouse standard, or have been suggested by a reviewer, or be widely accepted good practice. 18 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The ISO C Portability Issues are described in subsections of Annexes in the relevant standard. It is important to note that for both C90 and C99, the numbering of references is derived from the original standard and not from any later version that incorporates corrections and amendments. The reason for this decision is partly historical, in that MISRA C:2004 used this scheme, and partly practical, in that there is no officially-available revision of the C99 Standard that incorporates its Technical Corrigenda although there is a draft [12]. The subsections of the relevant standard corresponding to this kind of reference are: Annex G of ISO 9899:1990 Annex J of ISO 9899:1999 Unspecified G.1 J.1 Undefined G.2 J.2 Implementation G.3 J.3 Locale G.4 J.4 Reference Where text follows the reference, it has the following meanings: • Annex G of [2] references: the text identifies the number of an item or items in the relevant section of the Annex, numbered from the beginning of that section. So for example [Locale 2] refers to the second item in section G.4 of The Standard and [Undefined 3, 5] refers to the third and fifth items in Section G.2 of The Standard.
• Annex J of [8] references: for sections J.1, J.2 and J.4, the same interpretation as above applies. For section J.3, the text identifies the subsection of J.3, followed by a parenthesized item number or numbers. So for example [Unspecified 6] refers to the sixth item in Section J.1 of The Standard and [Implementation J3.4(2, 5)] refers to the second and fifth items in Section J.3.4 of The Standard. Section 6: Introduction to the guidelines 6.10.1 ISO C portability issue references Where a guideline is based on issues from Annex G (C90) or Annex J (C99) of The Standard, it is helpful for the reader to understand the distinction between the Unspecified, Undefined, Implementation and Locale references. These are explained briefly here, and further information can be found in Hatton [3]. Unspecified These are language constructs that must compile successfully, but in which the compiler writer has some freedom as to what the construct does. An example of this is the “order of evaluation” described in Rule 13.2. While the use of some unspecified behaviour is unavoidable, it is unwise to assume that the compiler produces object code that behaves in a particular way; the compiler need not even perform consistently across all possible constructs. MISRA C identifies specific instances of unspecified behaviour which are liable to result in unsafe behaviour. Undefined These are essentially programming errors, but for which the compiler writer is not obliged to provide error messages. Examples are invalid parameters to functions, or functions whose arguments do not match the defined parameters.
These are particularly important from a safety point of view, as they represent programming errors which may not necessarily be trapped by the compiler. Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 19 Section 6: Introduction to the guidelines Implementation These are similar to 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. The implementation-defined behaviours tend to be less critical from a safety point of view, provided the compiler writer has fully documented the intended approach and then implemented it consistently. It is advisable to avoid these issues where possible. Locale The locale-specific behaviours form 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. No issues arising from this source are addressed in this document. 6.10.2 Other references References other than the ISO C Portability Issues are taken from the following sources: Reference Source MISRA Guidelines The MISRA Guidelines [15] Koenig “C Traps and Pitfalls”, Koenig [31] IEC 61508 IEC 61508:2010 [32] ISO 26262 ISO 26262:2011 [23] DO-178C DO-178C [24] Unless stated otherwise, the text following a reference gives the relevant page number.
20 Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 7 Directives 7.1 The implementation Dir 1.1 Any implementation-defined behaviour on which the output of the program depends shall be documented and understood C90 [Annex G.3], C99 [Annex J.3] Category Required Applies to C90, C99 Amplification Appendix G of this document lists, for both C90 and C99, those implementation-defined behaviours that: • Are considered to have the potential to cause unexpected program operation, and • May be present in a program even if it complies with all the other MISRA C guidelines. All of these implementation-defined behaviours on which the program output depends must be: • Documented, and • Understood by developers. Note: a conforming implementation is required to document its treatment of all implementationdefined behaviour.
The developer of an implementation should be consulted if any documentation is missing. Rationale It is important to know that the output of a program was intentional and was not produced by chance.
Some of the more common implementation-defined behaviours on which safety-related embedded software is likely to depend are described below. Core behaviour The fundamental implementation-defined behaviours, which are likely to be required by most programs, include: • How to identify a diagnostic message produced during translation; • The type of the function main, commonly declared as void main (void) in freestanding implementations; • The number of significant characters in identifiers — needed to configure an analysis tool for Rule 5.2; • The source and execution character sets; • The sizes of the integer types; • How a #include’d name is mapped onto a file name and located in the host file system. 21 Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 Section 7: Directives Extensions Extensions are often used in embedded systems to provide access to peripherals and to place objects into regions of memory with special properties such as Flash EEPROM or fast-access RAM.
A conforming implementation is permitted to provide extensions provided that they do not alter the meaning of any strictly conforming program. Some C90 implementations may provide extensions which implement a subset of the C99 features. Some of the methods by which an implementation can provide extensions are: • The #pragma preprocessing directive or the _Pragma operator (C99 only); • New keywords. The Standard Library Some aspects of The Standard Library implementation that may be important are: • The values assigned to errno when certain Standard Library functions are used; • The implementation of clock and time functions; • The characteristics of the file system. The Application Binar y Interface It is sometimes necessary to interface C code with assembly language, for example to improve execution speed in those places where it is critical. It might also be necessary to interface code produced by different compilers, possibly for different languages. The Application Binary Interface (ABI) for a compiler provides the information necessary to perform this task, including some of the implementation-defined behaviours. It typically specifies: • How function parameters are passed in registers and on the stack; • How function values are returned; • Which registers must be preserved by a function; • How objects with automatic storage duration are allocated to stack frames; • Alignment requirements for each data type; • How structures are laid out and how bit-fields are allocated to storage units. Some processors have a standard ABI which will be used by all implementations. Where no standard ABI exists, an implementation will provide its own. Integer division In C90, signed integer division or remainder operations in which either operand has a negative value may either round downwards or towards zero. In C99, rounding is guaranteed to be towards zero. Floating-point implementation The implementation of floating-point types can have significant impact on program behaviour, for example: 22 • The range of values and the precision with which they are stored; • The direction of rounding following floating-point operations; Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The direction of rounding when converting to narrower floating-point types or to integer types; • The behaviour in the presence of underflow, overflow and NaNs; • The behaviour of library functions in the event of domain and range errors.
See also Rule 5.1, Rule 5.2 7.2 Section 7: Directives • Compilation and build Dir 2.1 All source files shall compile without any compilation errors Category Required Applies to C90, C99 Rationale A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program may produce unexpected behaviour. See Section 5.3.1 for further guidance on controlling and analysing compiler messages. See also Rule 1.1 7.3 Requirements traceability Dir 3.1 All code shall be traceable to documented requirements [DO-178C Section 6.4.4.3.d] Category Required Applies to C90, C99 Rationale Functionality that is not needed to meet the project requirements gives rise to unnecessary paths. It is possible that the developers of the software are not aware of the wider implications that might arise from this additional functionality. For example, developers might add code that toggles the state of a processor output pin every time a particular point in the program is reached. This might be very useful during development for measuring timing or for triggering an emulator or logic analyser. However, even though the pin in question might appear to be unused because it is not mentioned in the software requirements specification, it might be connected to an actuator in the target controller, resulting in unwanted external effects. The method by which code is traced back to documented requirements shall be determined by the project.
One method of achieving traceability is to review the code against the corresponding design documents which have in turn been reviewed against the requirements. Note: there should not be any conflict between this guideline and the provision of a protective coding strategy as the latter should be part of the requirements in a critical system.
Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 23 Section 7: Directives 7.4 Code design Dir 4.1 Run-time failures shall be minimized C90 [Undefined 15, 19, 26, 30, 31, 32, 94] C99 [Undefined 15, 16, 33, 40, 43–45, 48, 49, 113] Category Required Applies to C90, C99 Rationale The C language was designed to provide very limited built-in run-time checking. While this approach allows generation of compact and fast executable code, it places the burden of run-time checking on the programmer. In order to achieve the desired level of robustness, it is therefore important that programmers carefully consider adding dynamic checks wherever there is potential for run-time errors to occur. It is sometimes possible to demonstrate that the values of operands preclude the possibility of a runtime error during evaluation of an expression. In such cases, a dynamic check is not required provided that the argument supporting its omission is documented. Any such documentation should include the assumptions on which the argument depends. This information can be used during subsequent modifications of the code to ensure that the argument remains valid. The techniques that will be employed to minimize run-time failures should be planned and documented, for example in design standards, test plans, static analysis configuration files and code review checklists. The nature of these techniques may depend on the integrity requirements of the project.
See Section 5.2 for further details. Note: the presence of a run-time error indicates a violation of Rule 1.3. The following notes give some guidance on areas where consideration needs to be given to the provision of dynamic checks. • arithmetic errors This includes errors occurring in the evaluation of expressions, such as overflow, underflow, divide by zero or loss of significant bits through shifting.
In considering integer overflow, note that unsigned integer calculations do not strictly overflow but wrap around producing defined, but possibly unexpected, values. Careful consideration should be given to the ranges of values and order of operation in arithmetic expressions, for example: float32_t float32_t float32_t float32_t float32_t f1 f2 f3 f4 f5 = = = = = 1E38f; 10.0f; 0.1f; (f1 * f2) * f3; f1 * (f2 * f3); /* (f1 * f2) will overflow /* no overflow because (f2 * f3) * is (approximately) 1 */ */ if ((f3 >= 0.0f) && (f3 0) { mutex_unlock (m); } else { /* Mutex not unlocked on this path */ } } See also Rule 22.1, Rule 22.2, Rule 22.6 36 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 8 Rules 8.1 A standard C environment Rule 1.1 The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits [MISRA Guidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification The program shall use only those features of the C language and its library that are specified in the chosen version of The Standard (see Section 3.1). The Standard permits implementations to provide language extensions and the use of such extensions is permitted by this rule. Except when making use of a language extension, a program shall not: • Contain any violations of the language syntax described in The Standard; • Contain any violations of the constraints imposed by The Standard. A program shall not exceed the translation limits imposed by the implementation. The minimum translation limits are specified by The Standard but an implementation may provide higher limits.
Note: a conforming implementation generates a diagnostic for syntax and constraint violations but be aware that: • The diagnostic need not necessarily be an error but could, for example, be a warning; • The program may be translated and an executable generated despite the presence of a syntax or constraint violation; Note: a conforming implementation does not need to generate a diagnostic when a translation limit is exceeded; an executable may be generated but it is not guaranteed to execute correctly. Rationale Problems associated with language features that are outside the supported versions of ISO/IEC 9899 have not been considered during development of these guidelines. There is anecdotal evidence of some non-conforming implementations failing to diagnose constraint violations, for example in [38] p135, example 2 entitled “Error of writing into the const area”. Example Some C90 compilers provide support for inline functions using the __inline keyword. A C90 program that uses __inline will be compliant with this rule provided that it is intended to be translated using such a compiler. 37 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Many compilers for embedded targets provide additional keywords that qualify object types with attributes of the memory area in which the object is located, for example: • __zpage — the object can be accessed using a short instruction • __near — a pointer to the object can be held in 16 bits • __far — a pointer to the object can be held in 24 bits A program using these additional keywords will be compliant with this rule provided that the compiler supports those keywords as a language extension. See also Dir 2.1, Rule 1.2 Rule 1.2 Language extensions should not be used Category Advisory Analysis Undecidable, Single Translation Unit Applies to C90, C99 Rationale A program that relies on language extensions may be less portable than one that does not. Although The Standard requires that a conforming implementation document any extensions that it provides to the language, there is

a risk that this documentation might not provide a full description of the behaviour in all circumstances.
If this rule is not applied, the decision to use each language extension should be justified in the project's design documentation.
The methods by which valid use of each extension will be assured, for example checking the compiler and its diagnostics, should also be documented. It is recognized that it is necessary to use language extensions in embedded systems. The Standard requires that an extension does not alter the behaviour of any strictly conforming program. For example, a compiler might implement, as an extension, full evaluation of binary logical operators even though The Standard specifies that evaluation stops as soon as the result can be determined. Such an extension does not conform to The Standard because side effects in the right-hand operand of a logical AND operator would always occur, giving rise to a different behaviour. See also Rule 1.1 Rule 1.3 There shall be no occurrence of undefined or critical unspecified behaviour Category Required Analysis Undecidable, System Applies to C90, C99 Amplification 38 Some undefined and unspecified behaviours are dealt with by specific rules. This rule prevents all other undefined and critical unspecified behaviours. Appendix H lists the undefined behaviours and those unspecified behaviours that are considered critical. Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 Any program that gives rise to undefined or unspecified behaviour may not behave in the expected manner. In many cases, the effect is to make the program non-portable but it is also possible for more serious problems to occur. For example, undefined behaviour might affect the result of a computation. If correct operation of the software is dependent on this computation then system safety might be compromised. The problem is particularly difficult to detect if the undefined behaviour only manifests itself on rare occasions. Section 8: Rules Rationale Many of the MISRA C guidelines have been designed to avoid certain undefined and unspecified behaviours. For example, compliance with all of Rule 11.4, Rule 11.8 and Rule 19.2 ensures that it is not possible in C to create a non-const qualified pointer to an object declared with a const-qualified type. This avoids C90 [Undefined 39] and C99 [Undefined 61]. However, other behaviours are not covered by specific guidelines for example because: • It is unlikely that the behaviour will be encountered; • There is no practical guidance that can be given other than the obvious statement that the behaviour should be avoided. Instead of introducing a guideline for each undefined and critical unspecified behaviour, the MISRA C Guidelines directly address those that are considered most important and most likely to occur in practice.
Those behaviours that do not have specific guidelines are all covered together by this single rule. Appendix H lists all undefined and critical unspecified behaviours, along with the MISRA C guidelines that prevent their occurrence. It therefore indicates which behaviours are expected to be prevented by this rule and which behaviours are covered by other rules. Note: some implementations may provide well-defined behaviour for some of the undefined and unspecified behaviours listed in The Standard. If such well-defined behaviours are relied upon, including by means of a language extension, it will be necessary to deviate this rule in respect of those behaviours.
See also Dir 4.1 8.2 Unused code Rule 2.1 A project shall not contain unreachable code [IEC 61508-7 Section C.5.9], [DO-178C Section 6.4.4.3.c] Category Required Analysis Undecidable, System Applies to C90, C99 Rationale Provided that a program does not exhibit any undefined behaviour, unreachable code cannot be executed and cannot have any effect on the program's outputs. The presence of unreachable code may therefore indicate an error in the program's logic. 39 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules A compiler is permitted to remove any unreachable code although it does not have to do so. Unreachable code that is not removed by the compiler wastes resources, for example: • It occupies space in the target machine's memory; • Its presence may cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code; • Within a loop, it might prevent the entire loop from residing in an instruction cache.
It is sometimes desirable to insert code that appears to be unreachable in order to handle exceptional cases. For example, in a switch statement in which every possible value of the controlling expression is covered by an explicit case, a default clause shall be present according to Rule 16.4. The purpose of the default clause is to trap a value that should not normally occur but that may have been generated as a result of: • Undefined behaviour present in the program; • A failure of the processor hardware. If a compiler can prove that a default clause is unreachable, it may remove it, thereby eliminating the defensive action. On the assumption that the defensive action is important, it will be necessary either to demonstrate that the compiler does not eliminate the code despite it being unreachable, or to take steps to make the defensive code reachable. The former course of action requires a deviation against this rule, probably with a review of the object code or unit testing being used to support such a deviation.
The latter course of action can usually be achieved by means of a volatile access. For example, a compiler might determine that the range of values held by x is covered by the case clauses in a switch statement such as: uint16_t x; switch (x) By forcing x to be accessed by means of a volatile qualified lvalue, the compiler has to assume that the controlling expression could take any value: switch (*(volatile uint16_t *) &x) Note: code that has been conditionally excluded by pre-processor directives is not subject to this rule as it is not presented to the later phases of translation. Example enum light { red, amber, red_amber, green }; enum light next_light (enum light c) { enum light res; 40 switch (c) { case red: res = red_amber; break; case red_amber: res = green; break; case green: res = amber; break; case amber: res = red; break; Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules default: { /* * This default will only be reachable if the parameter c * holds a value that is not a member of enum light. */ error_handler ( ); break; } } return res; res = c; /* Non-compliant - this statement is * certainly unreachable */ } See also Rule 14.3, Rule 16.4 Rule 2.2 There shall be no dead code [IEC 61508-7 Section C.5.10], [ISO 26262-6 Section 9.4.5], [DO-178C Section 6.4.4.3.c] Category Required Analysis Undecidable, System Applies to C90, C99 Amplification Any operation that is executed but whose removal would not affect program behaviour constitutes dead code. Operations that are introduced by language extensions are assumed always to have an effect on program behaviour. Note: The behaviour of an embedded system is often determined not just by the nature of its actions, but also by the time at which they occur. Note: unreachable code is not dead code as it cannot be executed. Rationale The presence of dead code may be indicative of an error in the program's logic.
Since dead code may be removed by a compiler, its presence may cause confusion. Exception A cast to void is assumed to indicate a value that is intentionally not being used. The cast is therefore not dead code itself. It is treated as using its operand which is therefore also not dead code. 41 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Example In this example, it is assumed that the object pointed to by p is used in other functions. extern volatile uint16_t v; extern char *p; void f (void) { uint16_t x; (void) v; (int32_t) v; v >> 3; x = 3; *p++; (*p)++; /* * * /* /* /* * /* /* Compliant Non-compliant Non-compliant Non-compliant Non-compliant Compliant - v is accessed for its side effect and the cast to void is permitted by exception - the cast operator is dead - the >> operator is dead - the = operator is dead - x is not subsequently read - result of * operator is not used - *p is incremented */ */ */ */ */ */ } In the following compliant example, the __asm keyword is a language extension, not a function call operation, and is therefore not dead code: __asm ("NOP"); In the following example, the function g does not contain dead code, and is not itself dead code because it does not contain any operations.
However, the call to the function is dead because it could be removed without affecting program behaviour. void g (void) { /* Compliant - there are no operations in this function */ } void h (void) { g ( ); /* Non-compliant - the call could be removed } */ See also Rule 17.7 42 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 A project should not contain unused type declarations Category Advisory Analysis Decidable, System Applies to C90, C99 Rationale Section 8: Rules Rule 2.3 If a type is declared but not used, then it is unclear to a reviewer if the type is redundant or it has been left unused by mistake.
Example int16_t unusedtype (void) { typedef int16_t local_Type; /* Non-compliant */ return 67; } Rule 2.4 A project should not contain unused tag declarations Category Advisory Analysis Decidable, System Applies to C90, C99 Rationale If a tag is declared but not used, then it is unclear to a reviewer if the tag is redundant or it has been left unused by mistake. Example In the following example, the tag state is unused and the declaration could have been written without it.
void unusedtag (void) { enum state { S_init, S_run, S_sleep }; } /* Non-compliant */ In the following example, the tag record_t is used only in the typedef of record1_t which is used in the rest of the translation unit whenever the type is needed. This typedef can be written in a compliant manner by omitting the tag as shown in the definition of record2_t.
typedef struct record_t { uint16_t key; uint16_t val; } record1_t; /* Non-compliant */ typedef struct { uint16_t key; uint16_t val; } record2_t; /* Compliant */ 43 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 2.5 A project should not contain unused macro declarations Category Advisory Analysis Decidable, System Applies to C90, C99 Rationale If a macro is declared but not used, then it is unclear to a reviewer if the macro is redundant or it has been left unused by mistake. Example void use_macro (void) { #define SIZE 4 /* Non-compliant - DATA not used */ #define DATA 3 use_int16 (SIZE); } Rule 2.6 A function should not contain unused label declarations Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale If a label is declared but not used, then it is unclear to a reviewer if the label is redundant or it has been left unused by mistake. Example void unused_label (void) { int16_t x = 6; label1: use_int16 (x); } /* Non-compliant */ Rule 2.7 There should be no unused parameters in functions Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale 44 Most functions will be specified as using each of their parameters. If a function parameter is unused, it is possible that the implementation of the function does not match its specification. This rule highlights such potential mismatches. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 void withunusedpara (uint16_t *para1, int16_t unusedpara) { *para1 = 42U; } 8.3 /* Non-compliant - unused */ Section 8: Rules Example Comments Rule 3.1 The character sequences /* and // shall not be used within a comment Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale If a comment starting sequence, /* or //, occurs within a /* comment, is it quite likely to be caused by a missing */ comment ending sequence. If a comment starting sequence occurs within a // comment, it is probably because a region of code has been commented-out using //. Exception The sequence // is permitted within a // comment. Example Consider the following code fragment: /* some comment, end comment marker accidentally omitted Perform_Critical_Safety_Function(X); /* this comment is non-compliant */ In reviewing the page containing the call to the function, the assumption is that it is executed code. Because of the accidental omission of the end comment marker, the call to the safety critical function will not be executed. In the following C99 example, the presence of // comments changes the meaning of the program: x = y // /* + z // */ ; This gives x = y + z; but would have been x = y; in the absence of the two // comment start sequences. See also 45 Dir 4.4 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 3.2 Line-splicing shall not be used in // comments Category Required Analysis Decidable, Single Translation Unit Applies to C99 Amplification Line-splicing occurs when the \ character is immediately followed by a new-line character. If the source file contains multibyte characters, they are converted to the source character set before any splicing occurs. Rationale If the source line containing a // comment ends with a \ character in the source character set, the next line becomes part of the comment. This may result in unintentional removal of code. Note: line-splicing is described in Section 5.1.1.2(2) of both C90 and C99. Example In the following non-compliant example, the physical line containing the if keyword is logically part of the previous line and is therefore a comment. extern bool_t b; void f (void) { uint16_t x = 0; if (b) { ++x; } } // comment \ /* This is always executed */ See also Dir 4.4 8.4 Character sets and lexical conventions Rule 4.1 Octal and hexadecimal escape sequences shall be terminated C90 [Implementation 11], C99 [Implementation J.3.4(7, 8)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification An octal or hexadecimal escape sequence shall be terminated by either: 46 • The start of another escape sequence, or • The end of the character constant or the end of a string literal. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant '\x1f' consists of a single character whereas the character constant '\x1g' consists of the two characters '\x1' and 'g'.
The manner in which multi-character constants are represented as integers is implementation-defined. The potential for confusion is reduced if every octal or hexadecimal escape sequence in a character constant or string literal is terminated. Section 8: Rules Rationale Example In this example, each of the strings pointed to by s1, s2 and s3 is equivalent to "Ag". const char *s1 = "\x41g"; /* Non-compliant */ const char *s2 = "\x41" "g"; /* Compliant - terminated by end of literal */ const char *s3 = "\x41\x67"; /* Compliant - terminated by another escape */ int c1 = '\141t'; int c2 = '\141\t'; /* Non-compliant */ /* Compliant - terminated by another escape */ See also C90: Section 6.1.3.4, C99: Section 6.4.4.4 Rule 4.2 Trigraphs should not be used Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale Trigraphs are denoted by a sequence of two 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. Note: the so-called digraphs: %: %:%: are permitted because they are tokens. Trigraphs are replaced wherever they appear in the program prior to preprocessing. Example 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 ~~]" 47 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules 8.5 Identifiers Rule 5.1 External identifiers shall be distinct C90 [Undefined 7], C99 [Unspecified 7; Undefined 28] Category Required Analysis Decidable, System Applies to C90, C99 Amplification This rule requires that different external identifiers be distinct within the limits imposed by the implementation. The definition of distinct depends on the implementation and on the version of the C language that is being used: • In C90 the minimum requirement is that the first 6 characters of external identifiers are significant but their case is not required to be significant; • In C99 the minimum requirement is that the first 31 characters of external identifiers are significant, with each universal character or corresponding extended source character occupying between 6 and 10 characters.
In practice, many implementations provide greater limits. For example it is common for external identifiers in C90 to be case-sensitive and for at least the first 31 characters to be significant. Rationale If two identifiers differ only in non-significant characters, the behaviour is undefined. If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard. Long identifiers may impair the readability of code. While many automatic code generation systems produce long identifiers, there is a good argument for keeping identifier lengths well below this limit. Note: In C99, if an extended source character appears in an external identifier and that character does not have a corresponding universal character, The Standard does not specify how many characters it occupies. Example In the following example, the definitions all occur in the same translation unit. The implementation in question supports 31 significant case-sensitive characters in external identifiers. /* 1234567890123456789012345678901********* Characters */ int32_t engine_exhaust_gas_temperature_raw; int32_t engine_exhaust_gas_temperature_scaled; /* Non-compliant */ /* 1234567890123456789012345678901********* Characters int32_t engine_exhaust_gas_temp_raw; int32_t engine_exhaust_gas_temp_scaled; /* Compliant */ */ 48 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 /* file1.c */ int32_t abc = 0; Section 8: Rules In the following non-compliant example, the implementation supports 6 significant case-insensitive characters in external identifiers. The identifiers in the two translation units are different but are not distinct in their significant characters. /* file2.c */ int32_t ABC = 0; See also Dir 1.1, Rule 5.2, Rule 5.4, Rule 5.5 Rule 5.2 Identifiers declared in the same scope and name space shall be distinct C90 [Undefined 7], C99 [Undefined 28] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule does not apply if both identifiers are external identifiers because this case is covered by Rule 5.1. This rule does not apply if either identifier is a macro identifier because this case is covered by Rule 5.4 and Rule 5.5. The definition of distinct depends on the implementation and on the version of the C language that is being used: • In C90 the minimum requirement is that the first 31 characters are significant; • In C99 the minimum requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character. Rationale If two identifiers differ only in non-significant characters, the behaviour is undefined. If portabilit y is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard. Long identifiers may impair the readability of code. While many automatic code generation systems produce long identifiers, there is a good argument for keeping identifier lengths well below this limit. Example In the following example, the implementation in question supports 31 significant case-sensitive characters in identifiers that do not have external linkage. The identifier engine_exhaust_gas_temperature_local is compliant with this rule. Although it is not distinct from the identifier engine_exhaust_gas_temperature_raw, it is in a different scope. However, it is not compliant with Rule 5.3. Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 49 Section 8: Rules /* 1234567890123456789012345678901********* Characters */ extern int32_t engine_exhaust_gas_temperature_raw; static int32_t engine_exhaust_gas_temperature_scaled; /* Non-compliant */ void f (void) { /* 1234567890123456789012345678901********* int32_t engine_exhaust_gas_temperature_local; } Characters */ /* Compliant */ /* 1234567890123456789012345678901********* Characters */ static int32_t engine_exhaust_gas_temp_raw; static int32_t engine_exhaust_gas_temp_scaled; /* Compliant */ See also Dir 1.1, Rule 5.1, Rule 5.3, Rule 5.4, Rule 5.5 Rule 5.3 An identifier declared in an inner scope shall not hide an identifier declared in an outer scope Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification An identifier declared in an inner scope shall be distinct from any identifier declared in an outer scope. The definition of distinct depends on the implementation and the version of the C language that is being used: • In C90 the minimum requirement is that the first 31 characters are significant; • In C99 the minimum requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.
Rationale If an identifier is declared in an inner scope but is not distinct from an identifier that already exists in an outer scope, then the inner-most declaration will "hide" the outer one. This may lead to developer confusion. Note: An identifier declared in one name space does not hide an identifier declared in a different name space. The terms outer and inner scope are defined as follows: • Identifiers that have file scope can be considered as having the outermost scope; • Identifiers that have block scope have a more inner scope; • Successive, nested blocks, introduce more inner scopes.
50 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 void fn1 (void) { int16_t i; /* Declare an object "i" */ int16_t i; /* Non-compliant - hides previous "i" */ i = 3; /* Could be confusing as to which "i" this refers */ { Section 8: Rules Example } } struct astruct { int16_t m; }; extern void g (struct astruct *p); int16_t xyz = 0; /* Declare an object "xyz" */ void fn2 (struct astruct xyz) /* Non-compliant - outer "xyz" is * now hidden by parameter name */ { g (&xyz); } uint16_t speed; void fn3 (void) { typedef float32_t speed; } /* Non-compliant - type hides object */ See also Rule 5.2, Rule 5.8 Rule 5.4 Macro identifiers shall be distinct C90 [Undefined 7], C99 [Unspecified 7; Undefined 28] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule requires that, when a macro is being defined, its name be distinct from: • the names of the other macros that are currently defined; and • the names of their parameters. It also requires that the names of the parameters of a given macro be distinct from each other but does not require that macro parameters names be distinct across two different macros.
51 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules The definition of distinct depends on the implementation and on the version of the C language that is being used: • In C90 the minimum requirement is that the first 31 characters of macro identifiers are significant; • In C99 the minimum requirement is that the first 63 characters of macro identifiers are significant. In practice, implementations may provide greater limits. This rule requires that macro identifiers be distinct within the limits imposed by the implementation. Rationale If two macro identifiers differ only in non-significant characters, the behaviour is undefined. Since macro parameters are active only during the expansion of their macro, there is no issue with parameters in one macro being confused with parameters in another macro. If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard. Long macro identifiers may impair the readability of code. While many automatic code generation systems produce long macro identifiers, there is a good argument for keeping macro identifier lengths well below this limit. Note: In C99, if an extended source character appears in a macro name and that character does not have a corresponding universal character, The Standard does not specify how many characters it occupies. Example In the following example, the implementation in question supports 31 significant case-sensitive characters in macro identifiers. /* #define #define 1234567890123456789012345678901********* engine_exhaust_gas_temperature_raw egt_r engine_exhaust_gas_temperature_scaled egt_s /* #define #define 1234567890123456789012345678901********* engine_exhaust_gas_temp_raw egt_r engine_exhaust_gas_temp_scaled egt_s Characters */ /* Non-compliant */ Characters */ /* Compliant */ See also Rule 5.1, Rule 5.2, Rule 5.5 52 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Identifiers shall be distinct from macro names C90 [Undefined 7], C99 [Unspecified 7; Undefined 28] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 5.5 Amplification This rule requires that the names of macros that exist prior to preprocessing be distinct from the identifiers that exist after preprocessing. It applies to identifiers, regardless of scope or name space, and to any macros that have been defined regardless of whether the definition is still in force when the identifier is declared. The definition of distinct depends on the implementation and the version of the C language that is being used: • In C90 the minimum requirement is that the first 31 characters are significant; • In C99 the minimum requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character. Rationale Keeping macro names and identifiers distinct can help to avoid developer confusion. Example In the following non-compliant example, the name of the function-like macro Sum is also used as an identifier. The declaration of the object Sum is not subject to macro-expansion because it is not followed by a (character. The identifier therefore exists after preprocessing has been performed. #define Sum(x, y) ((x) + (y)) int16_t Sum; The following example is compliant because there is no instance of the identifier Sum after preprocessing. #define Sum(x, y) ((x) + (y)) int16_t x = Sum (1, 2); In the following example, the implementation in question supports 31 significant case-sensitive characters in identifiers that do not have external linkage. The example is non-compliant because the macro name is not distinct from an identifier name with internal linkage in the first 31 characters. /* 1234567890123456789012345678901********* Characters */ #define low_pressure_turbine_temperature_1 lp_tb_temp_1 static int32_t low_pressure_turbine_temperature_2; See also Rule 5.1, Rule 5.2, Rule 5.4 53 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 5.6 A typedef name shall be a unique identifier Category Required Analysis Decidable, System Applies to C90, C99 Amplification A typedef name shall be unique across all name spaces and translation units. Multiple declarations of the same typedef name are only permitted by this rule if the type definition is made in a header file and that header file is included in multiple source files. Rationale Reusing a typedef name either as another typedef name or as the name of a function, object or enumeration constant, may lead to developer confusion.
Exception The typedef name may be the same as the structure, union or enumeration tag name associated with the typedef. Example void func (void) { { typedef unsigned char u8_t; } { typedef unsigned char u8_t; } } /* Non-compliant - reuse */ typedef float mass; void func1 (void) { float32_t mass = 0.0f; } typedef struct list { struct list *next; uint16_t element; } list; typedef struct { struct chain { struct chain *list; uint16_t element; } s1; uint16_t length; } chain; 54 /* Non-compliant - reuse */ /* Compliant - exception */ /* Non-compliant - tag "chain" not * associated with typedef */ See also Rule 5.7 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 A tag name shall be a unique identifier Category Required Analysis Decidable, System Applies to C90, C99 Section 8: Rules Rule 5.7 Amplification The tag shall be unique across all name spaces and translation units.
All declarations of the tag shall specify the same type. Multiple complete declarations of the same tag are only permitted by this rule if the tag is declared in a header file and that header file is included in multiple source files. Rationale Reusing a tag name may lead to developer confusion. There is also undefined behaviour associated with reuse of tag names in C90 although this is not listed in The Standard's Annex.
This undefined behaviour was recognized in C99 as a constraint in Section 6.7.2.3. Exception The tag name may be the same as the typedef name with which it is associated. Example struct stag { uint16_t a; uint16_t b; }; struct stag a1 = { 0, 0 }; union stag a2 = { 0, 0 }; /* /* * * Compliant - compatible with above Non-compliant - declares different type from struct stag. Constraint violation in C99 */ */ The following example also violates Rule 5.3 struct deer { uint16_t a; uint16_t b; }; void foo (void) { struct deer { uint16_t a; }; } typedef struct coord { uint16_t x; uint16_t y; } coord; /* Non-compliant - tag "deer" reused */ /* Compliant by Exception */ Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 55 Section 8: Rules struct elk { uint16_t x; }; struct elk /* Non-compliant - declaration of different type * Constraint violation in C99 */ { uint32_t x; }; See also Rule 5.6 Rule 5.8 Identifiers that define objects or functions with external linkage shall be unique Category Required Analysis Decidable, System Applies to C90, C99 Amplification An identifier used as an external identifier shall not be used for any other purpose in any name space or translation unit, even if it denotes an object with no linkage. Rationale Enforcing uniqueness of identifier names in this manner helps avoid confusion. Identifiers of objects that have no linkage need not be unique since there is minimal risk of such confusion.
Example In the following example, file1.c and file2.c are both part of the same project.
/* file1.c */ int32_t count; void foo (void) { int16_t index; } /* file2.c */ static void foo (void) { int16_t count; int32_t index; /* "count" has external linkage /* "foo" has external linkage */ */ /* "index" has no linkage */ /* * * /* * * /* Non-compliant - "foo" is not unique (it is already defined with external linkage in file1.c) Non-compliant - "count" has no linkage but clashes with an identifier with external linkage Compliant - "index" has no linkage */ */ */ } See also Rule 5.3 56 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Identifiers that define objects or functions with internal linkage should be unique Category Advisor y Analysis Decidable, System Applies to C90, C99 Section 8: Rules Rule 5.9 Amplification The identifier name should be unique across all name spaces and translation units. Any identifier used in this way should not have the same name as any other identifier, even if that other identifier denotes an object with no linkage. Rationale Enforcing uniqueness of identifier names in this manner helps avoid confusion. Exception An inline function with internal linkage may be defined in more than one translation unit provided that all such definitions are made in the same header file that is included in each translation unit. Example In the following example, file1.c and file2.c are both part of the same project. /* file1.c */ static int32_t count; static void foo (void) { int16_t count; int16_t index; /* "count" has internal linkage /* "foo" has internal linkage */ */ /* * * /* Non-compliant - "count" has no linkage but clashes with an identifier with internal linkage "index" has no linkage */ */ /* * * /* * Non-compliant - "count" has no linkage but clashes with an identifier with internal linkage Compliant - "index" is not unique but has no linkage /* * * Non-compliant - "count" has internal linkage but clashes with other identifiers of the same name Non-compliant - "foo" has internal linkage but clashes with a function of the same name } void bar1 (void) { static int16_t count; int16_t index; */ */ foo ( ); } /* End of file1.c */ /* file2.c */ static int8_t count; static void foo (void) */ */ { int32_t index; int16_t nbytes; /* Compliant - both "index" and "nbytes" /* are not unique but have no linkage */ */ } 57 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules void bar2 (void) { static uint8_t nbytes; /* Compliant - "nbytes" is not unique but * has no linkage and the storage class is * irrelevant */ } /* End of file2.c */ See also Rule 8.10 8.6 Types Rule 6.1 Bit-fields shall only be declared with an appropriate type C90 [Undefined 38; Implementation 29], C99 [Implementation J.3.9(1, 2)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification The appropriate bit-field types are: • C90: either unsigned int or signed int; • C99: one of: - either unsigned int or signed int; - another explicitly signed or explicitly unsigned integer type that is permitted by the implementation; - _Bool.
Note: It is permitted to use typedefs to designate an appropriate type. Rationale Using int is implementation-defined because bit-fields of type int can be either signed or unsigned. The use of enum, short, char or any other type for bit-fields is not permitted in C90 because the behaviour is undefined. In C99, the implementation may define other integer types that are permitted in bit-field declarations.
58 Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 The following example is applicable to C90 and to C99 implementations that do not provide any additional bit-field types. It assumes that the int type is 16-bit. typedef unsigned int UINT_16; struct s { unsigned int int UINT_16 signed long b1:2; b2:2; b3:2; b4:2; /* /* /* /* * Compliant Non-compliant - plain int not permitted Compliant - typedef designating unsigned int Non-compliant even if long and int are the same size */ */ */ Section 8: Rules Example */ }; Rule 6.2 Single-bit named bit fields shall not be of a signed type Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has a single (one) sign bit and no (zero) value bits. In any representation of integers, 0 value bits cannot specify a meaningful value. A single-bit signed bit-field is therefore unlikely to behave in a useful way and its presence is likely to indicate programmer confusion. Although the C90 Standard does not provide so much detail regarding the representation of types, the same considerations apply as for C99. Note: this rule does not apply to unnamed bit fields as their values cannot be accessed. 8.7 Literals and constants Rule 7.1 Octal constants shall not be used [Koenig 9] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale Developers writing constants that have a leading zero might expect them to be interpreted as decimal constants. Note: this rule does not apply to octal escape sequences because the use of a leading \ character means that there is less scope for confusion. 59 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Exception The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule. Example extern uint16_t code[10]; code[code[code[code[1 2 3 4]]]] = = = = 109; 100; 052; 071; /* /* /* /* Compliant Compliant Non-Compliant Non-Compliant - decimal decimal decimal decimal 109 100 42 57 */ */ */ */ Rule 7.2 A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule applies to: • Integer constants that appear in the controlling expressions of #if and #elif preprocessing directives; • Any other integer constants that exist after preprocessing. Note: during preprocessing, the type of an integer constant is determined in the same manner as after preprocessing except that: • All signed integer types behave as if they were long (C90) or intmax_t (C99); • All unsigned integer types behave as if they were unsigned long (C90) or uintmax_t (C99).
Rationale The type of an integer constant is a potential source of confusion, because it is dependent on a complex combination of factors including: • The magnitude of the constant; • The implemented sizes of the integer types; • The presence of any suffixes; • The number base in which the value is expressed (i.e. decimal, octal or hexadecimal). For example, the integer constant 40000 is of type signed int in a 32-bit environment but of type signed long in a 16-bit environment. The value 0x8000 is of type unsigned int in a 16-bit environment, but of type signed int in a 32-bit environment.
Note: 60 • Any value with a "U" suffix is of unsigned type; • An unsuffixed decimal value less than 231 is of signed type. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 • An unsuffixed hexadecimal value greater than or equal to 215 may be of signed or unsigned type; • For C90, an unsuffixed decimal value greater than or equal to 231 may be of signed or unsigned type. Signedness of constants should be explicit.
If a constant is of an unsigned type, applying a "U" suffix makes it clear that the programmer understands that the constant is unsigned. Section 8: Rules But: Note: this rule does not depend on the context in which a constant is used; promotion and other conversions that may be applied to the constant are not relevant in determining compliance with this rule. Example The following example assumes a machine with a 16-bit int type and a 32-bit long type. It shows the type of each integer constant determined in accordance with The Standard. The integer constant 0x8000 is non-compliant because it has an unsigned type but does not have a "U" suffix. Constant Type Compliance 32767 signed int Compliant 0x7fff signed int Compliant 32768 signed long Compliant 32768u unsigned int Compliant 0x8000 unsigned int Non-compliant 0x8000u unsigned int Compliant Rule 7.3 The lowercase character "l" shall not be used in a literal suffix Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale Using the uppercase suffix "L" removes the potential ambiguity between "1" (digit 1) and "l" (letter "el") when declaring literals.
61 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Example Note: the examples containing the long long suffix are applicable only to C99. const const const const const const const const const const int64_t int64_t uint64_t uint64_t uint64_t uint64_t int128_t int128_t float128_t float128_t Rule 7.4 a b c d e f g h m n = = = = = = = = = = 0L; 0l; 0Lu; 0lU; 0ULL; 0Ull; 0LL; 0ll; 1.2L; 2.4l; /* Non-compliant */ /* Non-compliant */ /* Non-compliant */ /* Non-compliant */ /* Non-compliant */ A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char" C90 [Undefined 12], C99 [Unspecified 14; Undefined 30] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification No attempt shall be made to modify a string literal or wide string literal directly. The result of the address-of operator, &, applied to a string literal shall not be assigned to an object unless that object's type is "pointer to array of const-qualified char". The same considerations apply to wide string literals. A wide string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified wchar_t". The result of the address-of operator, &, applied to a wide string literal shall not be assigned to an object unless that object's type is "pointer to array of const-qualified wchar_t". Rationale Any attempt to modify a string literal results in undefined behaviour. For example, some implementations may store string literals in read-only memory in which case an attempt to modify the string literal will fail and may also result in an exception or crash.
This rule, when applied in conjunction with others, prevents a string literal from being modified. It is explicitly unspecified in C99 whether string literals that share a common ending are stored in distinct memory locations. Therefore, even if an attempt to modify a string literal appears to succeed, it is possible that another string literal might be inadvertently altered. Example The following example shows an attempt to modify a string literal directly.
"0123456789"[0] = '*'; /* Non-compliant */ These examples show how to prevent modification of string literals indirectly. 62 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 /* Compliant - p is const-qualified; additional qualifiers are permitted */ const volatile char *p = "string"; extern void f1 (char *s1); extern void f2 (const char *s2); void g (void) { f1 ("string"); f2 ("string"); /* Non-compliant - parameter s1 is not * const-qualified /* Compliant */ */ /* Non-compliant - return type is not * const-qualified */ Section 8: Rules /* Non-compliant - s is not const-qualified */ char *s = "string"; } char *name1 (void) { return ("MISRA"); } const char *name2 (void) { return ("MISRA"); /* Compliant } */ See also Rule 11.4, Rule 11.8 8.8 Declarations and definitions Rule 8.1 Types shall be explicitly specified Category Required Analysis Decidable, Single Translation Unit Applies to C90 Rationale The C90 standard permits types to be omitted in some circumstances, in which case the int type is implicitly specified. Examples of the circumstances in which an implicit int might be used are: • Object declarations; • Parameter declarations; • Member declarations; • typedef declarations; • Function return types. The omission of an explicit type might lead to confusion. For example, in the declaration: extern void g (char c, const k); the type of k is const int whereas const char might have been expected. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 63 Section 8: Rules Example The following examples show compliant and non-compliant object declarations: extern extern int16_t const const int16_t x; x; y; y; /* /* /* /* Non-compliant - implicit int type */ Compliant - explicit type */ Non-compliant - implicit int type */ Compliant - explicit type */ The following examples show compliant and non-compliant function type declarations: extern f (void); /* Non-compliant - implicit * int return type */ /* Compliant */ extern int16_t f (void); extern void g (char c, const k); extern void g (char c, const int16_t k); /* Non-compliant - implicit * int for parameter k */ /* Compliant */ The following examples show compliant and non-compliant type definitions: typedef (*pfi) (void); typedef int16_t (*pfi) (void); typedef void (*pfv) (const x); typedef void (*pfv) (int16_t x); /* * /* /* * /* Non-compliant - implicit int return type Compliant Non-compliant - implicit int for parameter x Compliant */ */ */ */ The following examples show compliant and non-compliant member declarations: struct str { int16_t x; const y; } s; /* Compliant */ /* Non-compliant - implicit int for member y */ See also Rule 8.2 Rule 8.2 Function types shall be in prototype form with named parameters C90 [Undefined 22–25], C99 [Undefined 36–39, 73, 79] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale The early version of C, commonly referred to as K&R C [30], did not provide a mechanism for checking the number of arguments or their types against the corresponding parameters. The type of an object or function did not have to be declared in K&R C since the default type of an object and the default return type of a function was int. 64 The C90 standard introduced function prototypes, a form of function declarator in which the parameter types were declared. This permitted argument types to be checked against parameter types. It also allowed the number of arguments to be checked except when a function prototype specified that a variable number of arguments was expected.
The C90 standard did not require the use of function prototypes for reasons of backward compatibility with existing code. For the same reason, it continued to permit types to be omitted in which case the type would default to int. Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 The mismatch between the number of arguments and parameters, their types and the expected and actual return type of a function provides potential for undefined behaviour. The purpose of this rule along with Rule 8.1 and Rule 8.4 is to avoid this undefined behaviour by requiring parameter types and function return types to be specified explicitly. Rule 17.3 ensures that this information is available at the time of a function call, thereby requiring the compiler to diagnose any mismatch that is detected. Section 8: Rules The C99 standard removed the default int type from the language but continued to allow K&R-style function types in which there was no means to supply parameter type information in a declaration and it was optional to supply parameter type information in a definition. This rule also requires that names be specified for all the parameters in a declaration. The parameter names can provide useful information regarding the function interface and a mismatch between a declaration and definition might be indicative of a programming error. Note: An empty parameter list is not valid in a prototype. If a function type has no parameters its prototype form uses the keyword void. Example The first example shows declarations of some functions and the corresponding definitions for some of those functions.
/* Compliant extern int16_t func1 (int16_t n); */ /* Non-compliant - parameter name not specified extern void func2 (int16_t); */ /* Non-compliant - not in prototype form static int16_t func3 ( ); */ /* Compliant - prototype specifies 0 parameters static int16_t func4 (void); */ /* Compliant int16_t func1 (int16_t n) { return n; } */ /* Non-compliant - old style identifier and declaration list */ static int16_t func3 (vec, n) int16_t *vec; int16_t n; { return vec[n - 1]; } 65 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules This example section shows the application of the rule to function types other than in function declarations and definitions.
/* Non-compliant - no prototype int16_t (*pf1) ( ); */ /* Compliant - prototype specifies 0 parameters int16_t (*pf1) (void); */ /* Non-compliant - parameter name not specified typedef int16_t (*pf2_t) (int16_t); */ /* Compliant typedef int16_t (*pf3_t) (int16_t n); */ See also Rule 8.1, Rule 8.4, Rule 17.3 Rule 8.3 All declarations of an object or function shall use the same names and type qualifiers C90 [Undefined 10], C99 [Undefined 14], [Koenig 59–62] Category Required Analysis Decidable, System Applies to C90, C99 Amplification Storage class specifiers are not included within the scope of this rule. Rationale Using types and qualifiers consistently across declarations of the same object or function encourages stronger typing. Specifying parameter names in function prototypes allows the function definition to be checked for interface consistency with its declarations. Exception Compatible versions of the same basic type may be used interchangeably. For example, int, signed and signed int are all equivalent. Example extern void void extern void void f f g g (signed int); (int); (int * const); (int *); /* Compliant - Exception */ /* Non-compliant - type qualifiers */ Note: all the above are not compliant with Dir 4.6. 66 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 /* Non-compliant - parameter names do not match */ int16_t func (int16_t den, int16_t num) { return num / den; } In this example the definition of area uses a different type name for the parameter h from that used in the declaration. This does not comply with the rule even though width_t and height_t are the same basic type. Section 8: Rules extern int16_t func (int16_t num, int16_t den); typedef uint16_t width_t; typedef uint16_t height_t; typedef uint32_t area_t; extern area_t area (width_t w, height_t h); area_t area (width_t w, width_t h) { return (area_t) w * h; } This rule does not require that a function pointer declaration use the same names as a function declaration. The following example is therefore compliant. extern void f1 (int16_t x); extern void f2 (int16_t y); void f (bool_t b) { void (*fp1) (int16_t z) = b ? f1 : f2; } See also Rule 8.4 Rule 8.4 A compatible declaration shall be visible when an object or function with external linkage is defined C90 [Undefined 24], C99 [Undefined 39] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification A compatible declaration is one which declares a compatible type for the object or function being defined. Rationale If a declaration for an object or function is visible when that object or function is defined, a compiler must check that the declaration and definition are compatible. In the presence of function prototypes, as required by Rule 8.2, checking extends to the number and type of function parameters. 67 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules The recommended method of implementing declarations of objects and functions with external linkage is to declare them in a header file, and then include the header file in all those code files that need them, including the one that defines them (See Rule 8.5).
Example In these examples there are no declarations or definitions of objects or functions other than those present in the code. extern int16_t count; int16_t count = 0; extern uint16_t speed = 6000u; uint8_t pressure = 101u; /* Compliant /* * /* * * */ Non-compliant prior to this Non-compliant prior to this - no declaration definition - no declaration definition */ */ extern void func1 (void); extern void func2 (int16_t x, int16_t y); extern void func3 (int16_t x, int16_t y); void func1 (void) { /* Compliant */ } The following non-compliant definition of func3 also violates Rule 8.3. void func2 (int16_t x, int16_t y) { /* Compliant } */ void func3 (int16_t x, uint16_t y) { /* Non-compliant - parameter types different */ } void func4 (void) { /* Non-compliant - no declaration of func4 before this definition } static void func5 (void) { /* Compliant - rule does not apply to objects/functions with internal * linkage } */ */ See also Rule 8.2, Rule 8.3, Rule 8.5, Rule 17.3 68 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 An external object or function shall be declared once in one and only one file [Koenig 66] Category Required Analysis Decidable, System Applies to C90, C99 Section 8: Rules Rule 8.5 Amplification This rule applies to non-defining declarations only. Rationale Typically, a single declaration will be made in a header file that will be included in any translation unit in which the identifier is defined or used. This ensures consistency between: • The declaration and the definition; • Declarations in different translation units. Note: there may be many header files in a project, but each external object or function shall only be declared in one header file. Example /* featureX.h */ extern int16_t a; /* Declare a */ /* file.c */ #include "featureX.h" int16_t a = 0; /* Define a */ See also Rule 8.4 Rule 8.6 An identifier with external linkage shall have exactly one external definition C90 [Undefined 44], C99 [Undefined 78], [Koenig 55, 63–65] Category Required Analysis Decidable, System Applies to C90, C99 Rationale The behaviour is undefined if an identifier is used for which multiple definitions exist (in different files) or no definition exists at all. Multiple definitions in different files are not permitted by this rule even if the definitions are the same. It is undefined behaviour if the declarations are different, or initialize the identifier to different values. 69 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Example In this example the object i is defined twice. /* file1.c */ int16_t i = 10; /* file2.c */ int16_t i = 20; /* Non-compliant - two definitions of i */ In this example the object j has one tentative definition and one external definition. /* file3.c */ int16_t j; int16_t j = 1; /* Tentative definition /* Compliant - external definition */ */ The following example is non-compliant because the object k has two external definitions. The tentative definition in file4.c becomes an external definition at the end of the translation unit. /* file4.c */ int16_t k; /* Tentative definition - becomes external */ /* file5.c */ int16_t k = 0; /* External definition Rule 8.7 */ Functions and objects should not be defined with external linkage if they are referenced in only one translation unit [Koenig 56, 57] Category Advisory Analysis Decidable, System Applies to C90, C99 Rationale Restricting the visibility of an object by giving it internal linkage or no linkage reduces the chance that it might be accessed inadvertently. Similarly, reducing the visibilit y of a function by giving it internal linkage reduces the chance of it being called inadvertently. Compliance with this rule also avoids any possibility of confusion between an identifier and an identical identifier in another translation unit or a library. Rule 8.8 The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification Since definitions are also declarations, this rule applies equally to definitions. Rationale 70 The Standard states that if an object or function is declared with the extern storage class specifier and another declaration of the object or function is already visible, the linkage is that specified by the Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Example static int32_t x = 0; extern int32_t x; /* definition: internal linkage /* Non-compliant */ */ static int32_t f (void); int32_t f (void) { return 1; } /* declaration: internal linkage */ /* Non-compliant */ static int32_t g (void); extern int32_t g (void) { return 1; } /* declaration: internal linkage */ /* Non-compliant */ Rule 8.9 An object should be defined at block scope if its identifier only appears in a single function Category Advisory Analysis Decidable, System Applies to C90, C99 Section 8: Rules earlier declaration. This can be confusing because it might be expected that the extern storage class specifier creates external linkage. The static storage class specifier shall therefore be consistently applied to objects and functions with internal linkage. Rationale Defining an objec t at block scope reduces the possibility that the object might be accessed inadvertently and makes clear the intention that it should not be accessed elsewhere. Within a function, whether objects are defined at the outermost or innermost block is largely a matter of style. It is recognized that there are situations in which it may not be possible to comply with this rule. For example, an object with static storage duration declared at block scope cannot be accessed directly from outside the block. This makes it impossible to set up and check the results of unit test cases without using indirect accesses to the object. In this kind of situation, some projects may prefer not to apply this rule. Example In this compliant example, i is declared at block scope because it is a loop counter. There is no need for other functions in the same file to use the same object for any other purpose.
void func (void) { int32_t i; for (i = 0; i < N; ++i) { } } In this compliant example, the function count keeps track of the number of times it has been called and returns that number. No other function needs to know the details of the implementation of count so the call counter is defined with block scope. Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 71 Section 8: Rules uint32_t count (void) { static uint32_t call_count = 0; ++call_count; return call_count; } Rule 8.10 An inline function shall be declared with the static storage class C99 [Unspecified 20; Undefined 67] Category Required Analysis Decidable, Single Translation Unit Applies to C99 Rationale If an inline function is declared with external linkage but not defined in the same translation unit, the behaviour is undefined. A call to an inline function declared with external linkage may call the external definition of the function, or it may use the inline definition. Although this should not affect the behaviour of the called function, it might affect execution timing and therefore have an impact on a real-time program. Note: an inline function can be made available to several translation units by placing its definition in a header file. See also Rule 5.9 Rule 8.11 When an array with external linkage is declared, its size should be explicitly specified Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule applies to non-defining declarations only. It is possible to define an array and specify its size implicitly by means of initialization. Rationale Although it is possible to declare an array with incomplete type and access its elements, it is safer to do so when the size of the array may be explicitly determined. Providing size information for each declaration permits them to be checked for consistency.
It may also permit a static checker to perform some array bounds analysis without needing to analyse more than one translation unit. Example 72 extern int32_t array1[10]; extern int32_t array2[ ]; /* Compliant */ /* Non-compliant */ Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 8.12 Rationale An implicitly-specified enumeration constant has a value 1 greater than its predecessor. If the first enumeration constant is implicitly-specified then its value is 0. An explicitly-specified enumeration constant has the value of the associated constant expression. If implicitly-specified and explicitly-specified constants are mixed within an enumeration list, it is possible for values to be replicated. Such replication may be unintentional and may give rise to unexpected behaviour. This rule requires that any replication of enumeration constants be made explicit, thus making the intent clear. Example In the following examples the green and yellow enumeration constants are given the same value. /* Non-compliant - yellow replicates implicit green */ enum colour { red = 3, blue, green, yellow = 5 }; /* Compliant */ enum colour { red = 3, blue, green = 5, yellow = 5 }; Rule 8.13 A pointer should point to a const-qualified type whenever possible Category Advisory Analysis Undecidable, System Applies to C90, C99 Amplification A pointer should point to a const-qualified type unless either: • It is used to modify an object, or • It is copied to another pointer that points to a type that is not const-qualified by means of either: - Assignment, or - Memory move or copying functions. For the purposes of simplicity, this rule is written in terms of pointers and the types that they point to.
However, it applies equally to arrays and the types of the elements that they contain. An array should have elements with const-qualified type unless either: • Any element of the array is modified, or • It is copied to a pointer that points to a type that is not const-qualified by the means described above. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 73 Section 8: Rules Rationale This rule encourages best practice by ensuring that pointers are not inadvertently used to modify objects. Conceptually, it is equivalent to initially declaring: • All arrays to have elements with const-qualified type, and • All pointers to point to const-qualified types. and then removing const-qualification only where it is necessary to comply with the constraints of the language standard. Example In the following non-compliant example, p is not used to modify an object but the type to which it points is not const-qualified. uint16_t f (uint16_t *p) { return *p; } The code would be compliant if the function were defined with: uint16_t g (const uint16_t *p) The following example violates a constraint because an attempt is made to use a const-qualified pointer to modify an object. void h (const uint16_t *p) { *p = 0; } In the following example, the pointer s is const-qualified but the type it points to is not. Since s is not used to modify an object, this is non-compliant.
#include char last_char (char * const s) { return s[strlen (s) - 1u]; } The code would be compliant if the function were defined with: char last_char (const char * const s) In this non-compliant example, none of the elements of the array a are modified but the element type is not const-qualified. uint16_t first (uint16_t a[5]) { return a[0]; } The code would be compliant if the function were defined with: uint16_t first (const uint16_t a[5]) 74 Licensed to: LEE CHING MIN.
5 Mar 2019.
Copy 1 of 1 The restrict type qualifier shall not be used C99 [Undefined 65, 66] Category Required Analysis Decidable, Single Translation Unit Applies to C99 Section 8: Rules Rule 8.14 Rationale When used with care the restrict type qualifier may improve the efficiency of code generated by a compiler. It may also allow improved static analysis. However, to use the restrict type qualifier the programmer must be sure that the memory areas operated on by two or more pointers do not overlap. There is a significant risk that a compiler will generate code that does not behave as expected if restrict is used incorrectly. Example The following example is compliant because the MISRA C Guidelines do not apply to The Standard Library functions. The programmer must ensure that the areas defined by p, q and n do not overlap. #include void f (void) { /* memcpy has restrict-qualified parameters */ memcpy (p, q, n); } The following example is non-compliant because a function has been defined using restrict.
void user_copy (void * restrict p, void * restrict q, size_t n) { } 8.9 Initialization Rule 9.1 The value of an object with automatic storage duration shall not be read before it has been set C90 [Undefined 41], C99 [Undefined 10, 17] Category Mandatory Analysis Undecidable, System Applies to C90, C99 Amplification For the purposes of this rule, an array element or structure member shall be considered as a discrete object. 75 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rationale According to The Standard, objects with static storage duration are automatically initialized to zero unless initialized explicitly. Objects with automatic storage duration are not automatically initialized and can therefore have indeterminate values. Note: it is sometimes possible for the explicit initialization of an automatic object to be ignored. This will happen when a jump to a label using a goto or switch statement "bypasses" the declaration of the object; the object will be declared as expected but any explicit initialization will be ignored. Example void f (bool_t b, uint16_t *p) { if (b) { *p = 3U; } } void g (void) { uint16_t u; f (false, &u); if (u == 3U) { /* Non-compliant - u has not been assigned a value */ } } In the following non-compliant C99 example, the goto statement jumps past the initialization of x. Note: This example is also non-compliant with Rule 15.1. { goto L1; uint16_t x = 10u; L1: x = x + 1u; } /* Non-compliant - x has not been assigned a value */ See also Rule 15.1, Rule 15.3 76 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The initializer for an aggregate or union shall be enclosed in braces C90 [Undefined 42], C99 [Undefined 76, 77] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 9.2 Amplification This rule applies to initializers for both objects and subobjects. An initializer of the form { 0 }, which sets all values to 0, may be used to initialize subobjects without nested braces. Note: this rule does not itself require explicit initialization of objects or subobjects. Rationale Using braces to indicate initialization of subobjects improves the clarity of code and forces programmers to consider the initialization of elements in complex data structures such as multi-dimensional arrays or arrays of structures. Exception 1. An array may be initialized using a string literal. 2. An automatic structure or union may be initialized using an expression with compatible structure or union type. 3. A designated initializer may be used to initialize part of a subobject. Example The following three initializations, which are permitted by The Standard, are equivalent. The first form is not permitted by this rule because it does not use braces to show subarray initialization explicitly. int16_t y[3][2] = { 1, 2, 0, 0, 5, 6 }; /* Non-compliant */ int16_t y[3][2] = { { 1, 2 }, { 0 }, { 5, 6 } }; /* Compliant */ int16_t y[3][2] = { { 1, 2 }, { 0, 0 }, { 5, 6 } }; /* Compliant */ In the following example, the initialization of z1 is compliant by virtue of Exception 3 because a designated initializer is used to initialize the subobject z1[1]. The initialization of z2 is also compliant for the same reason. The initialization of z3 is non-compliant because part of the subobject z3[1] is initialized with a positional initializer but is not enclosed in braces. The initialization of z4 is compliant because a designated initializer is used to initialize the subobject z4[0] and the initializer for subobject z4[1] is brace-enclosed. int16_t z1[2][2] = { { int16_t z2[2][2] = { { [ ]; int16_t z3[2][2] = { { int16_t z4[2][2] = { [0], [1][1] = 1 }; /* 0 }, 1][1] = 1, [1][0] = 0 /* 0 }, [1][0] = 0, 1 }; /* 0][1] = 0, { 0, 1 } }; /* Compliant */ Compliant */ Non-compliant */ Compliant */ 77 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules The first line in the following example initializes 3 subarrays without using nested braces. The second and third lines show equivalent ways to write the same initializer.
float32_t a[3][2] = { 0 }; float32_t a[3][2] = { { 0 }, { 0 }, { 0 } }; float32_t a[3][2] = { { 0.0f, 0.0f }, { 0.0f, 0.0f }, { 0.0f, 0.0f } }; union u1 { int16_t i; float32_t f; } u = { 0 }; /* Compliant /* Compliant */ */ /* Compliant */ /* Compliant */ struct s1 { uint16_t len; char buf[8]; } s[3] = { { 5u, { 'a', 'b', 'c', 'd', 'e', '\0', '\0', '\0' } }, { 2u, { 0 } }, {

.len = 0u } /* Compliant - buf initialized implicitly }; /* Compliant - s[] fully initialized Rule 9.3 Arrays shall not be partially initialized Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 */ */ Amplification If any element of an array object or subobject is explicitly initialized, then the entire object or subobject shall be
explicitly initialized. Rationale Providing an explicit initialization for each element of an array makes it clear that every element has been considered. Exception 1. An initializer of the form { 0 } may be used to explicitly initialize all elements of an array object or subobject. 2. An array whose initializer consists only of designated initializers may be
used, for example to perform a sparse initialization. 3. An array initialized using a string literal does not need an initializer for every element.
Example /* Compliant */ int32_t x[3] = { 0, 1, 2 }; /* Non-compliant - y[2] is implicitly initialized */ int32_t y[3] = { 0, 1 }; 78 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 /* Compliant - designated initializers for sparse matrix */ float32_t z[50] = { [1] = 1.0f, [25] = 2.0f }; In the following compliant example, each element of the array
arr is initialized: float32_t arr[3][2] = { { 0.0f, 0.0f }, { PI / 4.0f, -PI / 4.0f }, { 0 } /* initializes all elements of array subobject arr[2] */ }; Section 8: Rules /* Non-compliant - t[0] and t[3] are implicitly initialized */ float32_t t[4] = { [1] = 1.0f, 2.0f }; In the following example, array elements 6 to 9 are implicitly initialized to '\0': char h[10] =
"Hello"; /* Compliant by Exception 3 */ Rule 9.4 An element of an object shall not be initialized more than once Category Required Analysis Decidable, Single Translation Unit Applies to C99 Amplification This rule applies to initializers for both objects and subobjects. The provision of designated initializers in C99 allows the naming of the components
of an aggregate (structure or array) or of a union to be initialized within an initializer list and allows the object’s elements to be initialized in any order by specifying the array indices or structure member names they apply to (elements having no initialization value assume the default for uninitialized objects). Rationale Care is required when using
designated initializers since the initialization of object elements can be inadvertently repeated leading to overwriting of previously initialized elements.
The C99 Standard does not specify whether the side effects in an overwritten initializer occur or not although this is not listed in Annex J. In order to allow sparse arrays and structures, it is acceptable to only initialize those which are necessary to the application. Example Array initialization: /* * Required behaviour using positional initialization *
Compliant - a1 is -5, -4, -3, -2, -1 */ int16_t a1[5] = { -5, -4, -3, -2, -1 }; /* * Similar behaviour using designated initializers * Compliant - a2 is -5, -4, -3, -2, -1 */ int16_t a2[5] = { [0] = -5, [1] = -4, [2] = -3, [3] = -2, [4] = -1 }; Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 79 Section 8: Rules /* * Repeated designated initializer element values overwrite earlier ones * Non-compliant - a3 is -5, -4, -2, 0, -1 */ int16_t a3[5] = { [0] = -5, [1] = -4, [2] = -3, [2] = -2, [4] = -1 }; In the following non-compliant example, it is unspecified whether the side effect occurs or not: uint16_t *p; void f (void) { uint16_t a[
2] = { [0] = *p++, [0] = 1 }; } Structure initialization: struct mystruct { int32_t a; int32_t b; int32_t c; int32_t d; }; /* * Required behaviour using positional initialization * Compliant - s1 is 100, -1, 42, 999 */ struct mystruct s1 = { 100, -1, 42, 999 }; /* * Similar behaviour using designated initializers * Compliant - s2 is 100, -1, 42, 999 */ struct
mystruct s2 = { .a = 100, .b = -1, .c = 42, .d = 999 }; /* * Repeated designated initializer element values overwrite earlier ones * Non-compliant - s3 is 42, -1, 0, 999 */ struct mystruct s3 = { .a = 100, .b = -1, .a = 42, .d = 999 }; Rule 9.5 Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly
Category Required Analysis Decidable, Single Translation Unit Applies to C99 Amplification The rule applies equally to an array subobject that is a flexible array member. Rationale 80 If the size of an array is not specified explicitly, it is determined by the highest index of any of the elements that are initialized. When using designated initializers it
may not always be clear which initializer has the highest index, especially when the initializer contains a large number of elements. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Example /* Non-compliant - probably unintentional to have single element */ int a1[ ] = { [0] = 1 }; /* Compliant int a2[10] = { [0] = 1 }; Section 8: Rules To
make the intent clear, the array size shall be declared explicitly. This provides some protection if, during development of the program, the indices of the initialized elements are changed as it is a constraint violation (C99 Section 6.7.8) to initialize an element outside the bounds of an array.
*/ 8.10 The essential type model 8.10.1 Rationale The rules in this section collectively define the essential type model and restrict the C type system so as to: 1. Support a stronger system of type-checking; 2. Provide a rational basis for defining rules to control the use of implicit and explicit type conversions; 3. Promote portable coding practices; 4.
Address some of the type conversion anomalies found within ISO C. The essential type model does this by allocating an essential type to those objects and expressions which ISO C considers to be of arithmetic type. For example, adding an int to a char gives a result having essentially character type rather than the int type that is actually produced by
integer promotion. The full rationale behind the essential type model is given in Appendix C with Appendix D providing a comprehensive definition of the essential type of any arithmetic expression. 8.10.2 Essential type The essential type of an object or expression is defined by its essential type category and size. The essential type category of an
expression reflects its underlying behaviour and may be: • Essentially Boolean; • Essentially character; • Essentially enum; • Essentially signed; • Essentially unsigned; • Essentially floating. Note: each enumerated type is a unique essentially enum type identified as enum. This allows different enumerated types to be handled as distinct types, which
supports a stronger system of type-checking. One exception is the use of an enumerated type to define a Boolean value in C90. Such types are considered to have essentially Boolean type. Another exception is the use of anonymous enumerations as defined in Appendix D. Anonymous enumerations are a way of defining a set of related constant
integers and are considered to have an essentially signed type. Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 81 Section 8: Rules When comparing two types of the same type category, the terms wider and narrower are used to describe their relative sizes as measured in bytes. Two different types are sometimes implemented with the same size. The following table shows how the standard integer types map on to essential type categories. Essential
type category Boolean character signed unsigned enum floating _Bool char signed char signed short signed int signed long signed long long unsigned char unsigned short unsigned int unsigned long unsigned long long named enum float double long double Note: C99 implementations may provide extended integer types each of which would be
allocated a location appropriate to its rank and signedness (see C99 Section 6.3.1.1). The restrictions enforced by the rules in this section also apply to the compound assignment operators (e.g. ^=) as these are equivalent to assigning the result obtained from the use of one of the arithmetic, bitwise or shift operators. For example: u8a += u8b + 1U;
is equivalent to: u8a = u8a + (u8b + 1U); Rule 10.1 Operands shall not be of an inappropriate essential type C90 [Unspecified 23; Implementation 14, 17, 19, 32] C99 [Undefined 13, 49; Implementation J3.4(2, 5), J3.5(5), J3.9(6)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification In the following table a
number within a cell indicates where a restriction applies to the use of an essential type as an operand to an operator. These numbers correspond to paragraphs in the Rationale section below and indicate why each restriction is imposed. 82 Operator Operand [ ] integer Essential type category of arithmetic operand Boolean character enum 3 4 +
(unary) 3 4 5 - (unary) 3 4 5 signed unsigned floating 1 + - either 3 * / either 3 4 5 % either 3 4 5 < > = either 3 == != either ! && || any 2 2 8 5 1 2 2 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 2 Operand > Essential type category of arithmetic operand Boolean character enum signed unsigned floating left 3 4 5, 6 6 1 > right 3 4 7 7 1 ~
& | ^ any 3 4 5, 6 6 1 ?: 1st 2 2 2 ?: 2nd and 3rd 2 2 Section 8: Rules Operator Under this rule the ++ and -- operators behave the same way as the binary + and - operators.
Other rules place further restrictions on the combination of essential types that may be used within an expression. Rationale 1.
The use of an expression of essentially floating type for these operands is a constraint violation. 2. An expression of essentially Boolean type should always be used where an operand is interpreted as a Boolean value. 3. An operand of essentially Boolean type should not be used where an operand is interpreted as a numeric value. 4. An operand of
essentially character type should not be used where an operand is interpreted as a numeric value. The numeric values of character data are implementation-defined. 5. An operand of essentially enum type should not be used in an arithmetic operation because an enum object uses an implementation-defined integer type. An operation involving an
enum object may therefore yield a result with an unexpected type. Note that an enumeration constant from an anonymous enum has essentially signed type. 6. Shift and bitwise operations should only be performed on operands of essentially unsigned type. The numeric value resulting from their use on essentially signed types is
implementationdefined. 7. The right hand operand of a shift operator should be of essentially unsigned type to ensure that undefined behaviour does not result from a negative shift. 8. An operand of essentially unsigned type should not be used as the operand to the unary minus operator, as the signedness of the result is determined by the
implemented size of int. Exception A non-negative integer constant expression of essentially signed type may be used as the right hand operand to a shift operator.
Example enum enuma { a1, a2, a3 } ena, enb; enum { K1 = 1, K2 = 2 }; /* Essentially enum /* Essentially signed */ */ The following examples are non-compliant. The comments refer to the numbered rationale item that results in the non-compliance. f32a & 2U f32a blb /* Rationale 3 - Boolean used as a numeric value /* Rationale 3 - Boolean used as a
numeric value */ */ cha & chb cha 0U u8a u8c); /* /* /* pu8a = pu8b; u8a = u8b + u8c + u8d; u8a = (uint8_t) s8a; /* By exception */ /* true is essentially Boolean */ Constant value fits By exception By exception cha = cha + 1 assigns character to character */ */ */ */ /* Same essential type /* Same essential type /* Cast gives same essential type */ */
/ Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 / /* /* /* Assignment Assignment Assignment Assignment to to to to a wider essential type a wider essential type a wider essential type same essential type */ */ */ */ The following are non-compliant as they have different essential type categories: uint8_t u8a = 1.0f; bool_t bla = 0; cha = 7; u8a
= 'a'; u8b = 1 - 2; u8c += 'a'; use_uint32 (s32a); /* /* /* /* /* /* unsigned and floating */ boolean and signed */ character and signed */ unsigned and signed */ u8c = u8c + 'a' assigns character to unsigned */ signed and unsigned */ Section 8: Rules u32a = u16a; u32a = 2U + 125U; use_uint16 (u8a); use_uint16 (u8a +
u16b); The following are non-compliant as they contain assignments to a narrower essential type: s8a = K2; u16a = u32a; /* Constant value does not fit /* uint32_t to uint16_t */ */ use_uint16 (u32a); /* uint32_t to uint16_t */ /* uint16_t to uint8_t */ uint8_t foo1 (uint16_t x) { return x; } See also Rule 10.4, Rule 10.5, Rule 10.6 Rule 10.4 Both
operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category C90 [Implementation 21], C99 [Implementation 3.6(4)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule applies to operators that are described in usual arithmetic
conversions (see C90 Section 6.2.1.5, C99 Section 6.3.1.8). This includes all the binary operators, excluding the shift, logical &&, logical || and comma operators. In addition, the second and third operands of the ternary operator are covered by this rule. Note: the increment and decrement operators are not covered by this rule. Rationale The
C language allows the programmer considerable freedom and will permit conversions between different arithmetic types to be performed automatically. However, the use of these implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. Further details of concerns with the C type system can be found in
Appendix C. The use of stronger typing, as enforced by the MISRA essential type model, allows implicit conversions to be restricted to those that should then produce the answer expected by the developer. 87 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Exception The following are permitted to allow a common form of
character manipulation to be used: 1. The binary + and += operators may have one operand with essentially character type and the other operand with an essentially signed or essentially unsigned type; 2. The binary - and -= operators may have a left-hand operand with essentially character type and a right-hand operand with an essentially signed or
essentially unsigned type. Example enum enuma { A1, A2, A3 } ena; enum enumb { B1, B2, B3 } enb; The following are compliant as they have the same essential type category: ena > A1 u8a + u16b The following is compliant by exception 1: cha += u8a The following is non-compliant with this rule and also violates Rule 10.3: s8a += u8a /* signed
and unsigned */ The following are non-compliant: u8b enb ena u8a + 2 > A1 == enb += cha /* /* /* /* unsigned and signed enum and enum enum unsigned and char */ */ */ */ See also Rule 10.3, Rule 10.7 88 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The value of an expression should not be cast to an inappropriate essential
type Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 10.5 Amplification The casts which should be avoided are shown in the following table, where values are cast (explicitly converted) to the essential type category of the first column. Essential type category to from Boolean Boolean character
Avoid enum Avoid signed Avoid unsigned Avoid floating Avoid character enum signed unsigned floating Avoid Avoid Avoid Avoid Avoid Avoid Avoid Avoid* Avoid Avoid Avoid Avoid * Note: an enumerated type may be cast to an enumerated type provided that the cast is to the same essential enumerated type. Such casts are redundant. Casting from
void to any other type is not permitted as it results in undefined behaviour. This is covered by Rule 1.3. Rationale An explicit cast may be introduced for legitimate functional reasons, for example: • To change the type in which a subsequent arithmetic operation is performed; • To truncate a value deliberately; • To make a type conversion explicit in
the interests of clarity. However, some explicit casts are considered inappropriate: • In C99, the result of a cast or assignment to _Bool is always 0 or 1. This is not necessarily the case when casting to another type which is defined as essentially Boolean; • A cast to an essentially enum type may result in a value that does not lie within the set of
enumeration constants for that type; • A cast from essentially Boolean to any other type is unlikely to be meaningful; • Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.
Exception An integer constant expression with the value 0 or 1 of either signedness may be cast to a type which is defined as essentially Boolean. This allows the implementation of non-C99 Boolean models. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 89 Section 8: Rules Example ((((bool_t) false int32_t) 3U bool_t) 0 bool_t) 3U (
int32_t) ena (enum enuma) 3 (char) enc /* /* /* Compliant - C99 'false' is essentially Boolean */ Compliant */ Compliant - by exception */ Non-compliant */ /* Compliant /* Non-compliant /* Compliant */ */ */ See also Rule 10.3, Rule 10.8 8.10.3 Composite operators and expressions Some of the concerns mentioned in Appendix C can be avoided by
restricting the implicit and explicit conversions that may be applied to non-trivial expressions. These include: • The confusion about the type in which integer expressions are evaluated, as this depends on the type of the operands after any integer promotion. The type of the result of an arithmetic operation depends on the implemented size of int; •
The common misconception among programmers that the type in which a calculation is conducted is influenced by the type to which the result is assigned or cast. This false expectation may lead to unintended results. In addition to the previous rules, the essential type model places further restrictions on expressions whose operands are composite
expressions, as defined below. The following are defined as composite operators in this document: • Multiplicative (*, /, %) • Additive (binary +, binary -) • Bitwise (&, |, ^) • Shift () • Conditional (?:) if either the second or third operand is a composite expression A compound assignment is equivalent to an assignment of the result of its corresponding
composite operator.
A composite expression is defined in this document as a non-constant expression which is the direct result of a composite operator. Note: • The result of a compound assignment operator is not a composite expression; • A parenthesized composite expression is also a composite expression; • A constant expression is not a composite expression. 90
Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The value of a composite expression shall not be assigned to an object with wider essential type Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 10.6 Amplification This rule covers the assigning operations described in Rule 10.3. Rationale
The rationale is described in the introduction on composite operators and expressions (see Section 8.10.3). Example The following are compliant: u16c = u16a + u16b; u32a = (uint32_t) u16a + u16b; /* Same essential type /* Cast causes addition in uint32_t */ */ The following are non-compliant: u32a = u16a + u16b; use_uint32 (u16a + u16b); /*
Implicit conversion on assignment */ /* Implicit conversion of fn argument */ See also Rule 10.3, Rule 10.7, Section 8.10.3 Rule 10.7 If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type Category Required Analysis Decidable,
Single Translation Unit Applies to C90, C99 Rationale The rationale is described in the introduction on composite operators and expressions (see Section 8.10.3).
Restricting implicit conversions on composite expressions means that sequences of arithmetic operations within an expression must be conducted in exactly the same essential type. This reduces possible developer confusion. Note: this does not imply that all operands in an expression are of the same essential type. The expression u32a + u16b + u16c
is compliant as both additions will notionally be performed in type uint32_t. In this case only non-composite expressions are implicitly converted.
The expression (u16a + u16b) + u32c is non-compliant as the left addition is notionally performed in type uint16_t and the right in type uint32_t, requiring an implicit conversion of the composite expression u16a + u16b to uint32_t.
Licensed to: LEE CHING MIN. 5 Mar 2019.
Copy 1 of 1 91 Section 8: Rules Example The following are compliant: u32a * u16a + u16b (u32a * u16a) + u16b u32a * ((uint32_t) u16a + u16b) u32a += (u32b + u16b) /* /* /* * /* No composite conversion No composite conversion Both operands of * have same essential type No composite conversion */ */ */ */ The following are non-compliant:
u32a * (u16a + u16b) u32a += (u16a + u16b) /* Implicit conversion of (u16a + u16b) /* Implicit conversion of (u16a + u16b) */ */ See also Rule 10.4, Rule 10.6, Section 8.10.3 Rule 10.8 The value of a composite expression shall not be cast to a different essential type category or a wider essential type Category Required Analysis Decidable,
Single Translation Unit Applies to C90, C99 Rationale The rationale is described in the introduction on composite operators and expressions (see Section 8.10.3). Casting to a wider type is not permitted as the result may vary between implementations. Consider the following: (uint32_t) (u16a + u16b); On a 16-bit machine the addition will be
performed in 16 bits with the result wrapping modulo-2 before it is cast to 32 bits. However, on a 32-bit machine the addition will take place in 32 bits and would preserve high-order bits that would have been lost on a 16-bit machine. Casting to a narrower type with the same essential type category is acceptable as the explicit truncation of the result
always leads to the same loss of information. Example (uint16_t) (u32a + u32b) (uint16_t) (s32a + s32b) (uint16_t) s32a (uint32_t) (u16a + u16b) /* /* * /* /* * Compliant Non-compliant - different essential type category Compliant - s32a is not composite Non-compliant - cast to wider essential type */ */ */ */ See also Rule 10.5, Section 8.10.3 92
Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules 8.11 Pointer type conversions Pointer types can be classified as follows: • Pointer to object; • Pointer to function; • Pointer to incomplete; • Pointer to void; • A null pointer constant, i.e. the value 0, optionally cast to void *. The only conversions involving pointers that are
permitted by The Standard are: • A conversion from a pointer type into void; • A conversion from a pointer type into an arithmetic type; • A conversion from an arithmetic type into a pointer type; • A conversion from one pointer type into another pointer type.
Although permitted by the language constraints, conversion between pointers and any arithmetic types other than integer types is undefined. The following permitted pointer conversions do not require an explicit cast: • A conversion from a pointer type into _Bool (C99 only); • A conversion from a null pointer constant into a pointer type; • A
conversion from a pointer type into a compatible pointer type provided that the destination type has all the type qualifiers of the source type; • A conversion between a pointer to an object or incomplete type and void *, or qualified version thereof, provided that the destination type has all the type qualifiers of the source type. In C99, any implicit
conversion that does not fall into this subset of pointer conversions violates a constraint (C99 Sections 6.5.4 and 6.5.16.1). In C90, any implicit conversion that does not fall into this subset of pointer conversions leads to undefined behaviour (C90 Sections 6.3.4 and 6.3.16.1). The conversion between pointer types and integer types is implementation-
defined. Rule 11.1 Conversions shall not be performed between a pointer to a function and any other type C90 [Undefined 24, 27–29], C99 [Undefined 21, 23, 39, 41] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification A pointer to a function shall only be converted into or from a pointer to a function with a
compatible type. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 93 Section 8: Rules Rationale The conversion of a pointer to a function into or from any of: • Pointer to object; • Pointer to incomplete; • void * results in undefined behaviour. If a function is called by means of a pointer whose type is not compatible with the called function, the
behaviour is undefined. Conversion of a pointer to a function into a pointer to a function with a different type is permitted by The Standard. Conversion of an integer into a pointer to a function is also permitted by The Standard. However, both are prohibited by this rule in order to avoid the undefined behaviour that would result from calling a function
using an incompatible pointer type. Exception 1. A null pointer constant may be converted into a pointer to a function; 2. A pointer to a function may be converted into void; 3.
A function type may be implicitly converted into a pointer to that function type. Note: exception 3 covers the implicit conversions described in C90 Section 6.2.2.1 and C99 Section 6.3.2.1. These conversions commonly occur when: • A function is called directly, i.e. using a function identifier to denote the function to be called; • A function is assigned
to a function pointer. Example typedef void (*fp16) (int16_t n); typedef void (*fp32) (int32_t n); #include /* To obtain macro NULL */ fp16 fp1 = NULL; fp32 fp2 = (fp32) fp1; /* /* * * */ Compliant - exception 1 Non-compliant - function pointer into different function pointer if (fp2 != NULL) { } /* Compliant - exception 1 fp16 fp3 = (fp16)
0x8000; /* * /* * fp16 fp4 = (fp16) 1.0e6F; Non-compliant - integer into function pointer Non-compliant - float into function pointer */ */ */ */ In the following example, the function call returns a pointer to a function type. Casting the return value into void is compliant with this rule. typedef fp16 (*pfp16) (void); pfp16 pfp1; 94 (void) (*pfp1 ( )); /*
Compliant - exception 2 - cast function * pointer into void */ Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 extern void f (int16_t n); f (1); /* Compliant - exception 3 - implicit conversion * of f into pointer to function /* Compliant - exception 3 fp16 fp5 = f; Rule 11.2 */ */ Section 8: Rules The following examples show compliant implicit
conversions from a function type into a pointer to that function type. Conversions shall not be performed between a pointer to an incomplete type and any other type C90 [Undefined 29], C99 [Undefined 21, 22, 41] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification A pointer to an incomplete type shall
not be converted into another type. A conversion shall not be made into a pointer to incomplete type. Although a pointer to void is also a pointer to an incomplete type, this rule does not apply to pointers to void as they are covered by Rule 11.5. Rationale Conversion into or from a pointer to an incomplete type may result in a pointer that is not
correctly aligned, resulting in undefined behaviour. Conversion of a pointer to an incomplete type into or from a floating type always results in undefined behaviour. Pointers to an incomplete type are sometimes used to hide the representation of an object. Converting a pointer to an incomplete type into a pointer to object would break this
encapsulation. Exception 1. A null pointer constant may be converted into a pointer to an incomplete type. 2. A pointer to an incomplete type may be converted into void. Example struct s; struct t; struct s *sp; struct t *tp; int16_t *ip; /* Incomplete type /* A different incomplete type */ */ #include /* To obtain macro NULL */ ip = (int16_t *) sp; sp = (
struct s *) 1234; tp = (struct t *) sp; /* /* /* * /* */ */ sp = NULL; Non-compliant Non-compliant Non-compliant - casting pointer into a different incomplete type Compliant - exception 1 */ */ Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 95 Section 8: Rules struct s *f (void); (void) f ( ); /* Compliant - exception 2 */ See also Rule 11.5 Rule
11.3 A cast shall not be performed between a pointer to object type and a pointer to a different object type C90 [Undefined 20], C99 [Undefined 22, 34] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule applies to the unqualified types that are pointed to by the pointers. Rationale Casting a
pointer to object into a pointer to a different object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. Even if conversion is known to produce a pointer that is correctly aligned, the behaviour may be undefined if that pointer is used to access an object. For example, if an object whose type is int is accessed as a short
the behaviour is undefined even if int and short have the same representation and alignment requirements. See C90 Section 6.3, C99 Section 6.5, paragraph 7 for details. Exception It is permitted to convert a pointer to object type into a pointer to one of the object types char, signed char or unsigned char.
The Standard guarantees that pointers to these types can be used to access the individual bytes of an object. Example uint8_t *p1; uint32_t *p2; /* Non-compliant - possible incompatible alignment */ p2 = (uint32_t *) p1; extern uint32_t read_value (void); extern void print (uint32_t n); void f (void) { uint32_t u = read_value ( ); uint16_t *hi_p = (
uint16_t *) &u; *hi_p = 0; print (u); /* Non-compliant even though * probably correctly aligned */ /* Attempt to clear high 16-bits on big-endian machine */ /* Line above may appear not to have been performed */ } 96 The following example is compliant because the rule applies to the unqualified pointer types.
It does not prevent type qualifiers from being added to the object type. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 q = (const volatile short *) p; /* Compliant */ The following example is non-compliant because the unqualified pointer types are different, namely “pointer to const-qualified int” and “pointer to int”. int * const * pcpi; const int
* const * pcpci; Section 8: Rules const short *p; const volatile short *q; pcpci = (const int * const *) pcpi; See also Rule 11.4, Rule 11.5, Rule 11.8 Rule 11.4 A conversion should not be performed between a pointer to object and an integer type C90 [Undefined 20; Implementation 24] C99 [Undefined 21, 34; Implementation J.3.7(1)] Category Advisory
Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification A pointer should not be converted into an integer. An integer should not be converted into a pointer. Rationale Conversion of an integer into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. Conversion of a pointer to
object into an integer may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour. Note: the C99 types intptr_t and uintptr_t, declared in , are respectively signed and unsigned integer types capable of representing pointer values. Despite this, conversions between a pointer to object and these types is
not permitted by this rule because their use does not avoid the undefined behaviour associated with misaligned pointers.
Casting between a pointer and an integer type should be avoided where possible, but may be necessary when addressing memory mapped registers or other hardware specific features. If casting between integers and pointers is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed
under Rule 11.3. Exception A null pointer constant that has integer type may be converted into a pointer to object. 97 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Example uint8_t *PORTA = (uint8_t *) 0x0002; /* Non-compliant */ uint16_t *p; int32_t uint8_t bool_t addr = (int32_t) &p; *q = (uint8_t *) addr; b = (bool_t)
p; enum etag { A, B } e = (enum etag) p; /* Non-compliant */ /* Non-compliant */ /* Non-compliant */ /* Non-compliant */ See also Rule 11.3, Rule 11.7, Rule 11.9 Rule 11.5 A conversion should not be performed from pointer to void into pointer to object C90 [Undefined 20], C99 [Undefined 22, 34] Category Advisory Analysis Decidable, Single
Translation Unit Applies to C90, C99 Rationale Conversion of a pointer to void into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. It should be avoided where possible but may be necessary, for example when dealing with memory allocation functions. If conversion from a pointer to object into a
pointer to void is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed under Rule 11.3. Exception A null pointer constant that has type pointer to void may be converted into pointer to object. Example uint32_t *p32; void *p; uint16_t *p16; p = p32; p16 = p; p = (void *) p16; p32 = (
uint32_t *) p; /* * /* /* Compliant - pointer to uint32_t into pointer to void Non-compliant Compliant Non-compliant */ */ */ */ See also Rule 11.2, Rule 11.3 98 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 A cast shall not be performed between pointer to void and an arithmetic type C90 [Undefined 29; Implementation 24] C99 [Undefined
21, 41; Implementation J.3.7(1)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 11.6 Rationale Conversion of an integer into a pointer to void may result in a pointer that is not correctly aligned, resulting in undefined behaviour. Conversion of a pointer to void into an integer may produce a
value that cannot be represented in the chosen integer type resulting in undefined behaviour. Conversion between any non-integer arithmetic type and pointer to void is undefined. Exception An integer constant expression with value 0 may be cast into pointer to void. Example void *p; uint32_t u; /* Non-compliant - implementation-defined */ p = (
void *) 0x1234u; /* Non-compliant - undefined p = (void *) 1024.0f; */ /* Non-compliant - implementation-defined */ u = (uint32_t) p; Rule 11.7 A cast shall not be performed between pointer to object and a noninteger arithmetic type C90 [Undefined 29; Implementation 24] C99 [Undefined 21, 41; Implementation J.3.7(1)] Category Required
Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification For the purposes of this rule a non-integer arithmetic type means one of: • Essentially Boolean; • Essentially character; • Essentially enum; • Essentially floating. 99 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rationale Conversion of an
essentially Boolean, essentially character or essentially enum type into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. Conversion of a pointer to object into an essentially Boolean, essentially character or essentially enum type may produce a value that cannot be represented in the chosen
integer type resulting in undefined behaviour.
Conversion of a pointer to object into or from an essentially floating type results in undefined behaviour. Example int16_t *p; float32_t f; f = (float32_t) p; p = (int16_t *) f; /* Non-compliant */ /* Non-compliant */ See also Rule 11.4 Rule 11.8 A cast shall not remove any const or volatile qualification from the type pointed to by a pointer C90
[Undefined 12, 39, 40], C99 [Undefined 30, 61, 62] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale Any attempt to remove the qualification associated with the addressed type by using casting is a violation of the principle of type qualification. Note: the qualification referred to here is not the same as any
qualification that may be applied to the pointer itself.
Some of the problems that might arise if a qualifier is removed from the addressed object are: • Removing a const qualifier might circumvent the read-only status of an object and result in it being modified; • Removing a const qualifier might result in an exception when the object is accessed; • Removing a volatile qualifier might result in accesses to
the object being optimized away. Note: removal of the C99 restrict type qualifier is benign. 100 Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 uint16_t x; uint16_t * const cpi = &x; uint16_t * const *pcpi; uint16_t * *ppi; const uint16_t *pci; volatile uint16_t *pvi; uint16_t *pi; pi = cpi; pi = (uint16_t *)pci; pi = (uint16_t *)pvi; ppi = (uint16_t * *)pcpi; /* const pointer /* pointer to const pointer */ */ /* pointer to const /* pointer to volatile */ */ /* Compliant - no conversion
no cast required /* Non-compliant /* Non-compliant /* Non-compliant */ */ */ */ Section 8: Rules Example See also Rule 11.3 Rule 11.9 The macro NULL shall be the only permitted form of integer null pointer constant Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification An integer constant expression with
the value 0 shall be derived from expansion of the macro NULL if it appears in any of the following contexts: • As the value being assigned to a pointer; • As an operand of an == or != operator whose other operand is a pointer; • As the second operand of a ?: operator whose third operand is a pointer; • As the third operand of a ?: operator whose
second operand is a pointer. Ignoring whitespace and any surrounding parentheses, any such integer constant expression shall represent the entire expansion of NULL. Note: a null pointer constant of the form (void *)0 is permitted, whether or not it was expanded from NULL. Rationale Using NULL rather than 0 makes it clear that a null pointer
constant was intended. Example In the following example, the initialization of p2 is compliant because the integer constant expression 0 does not appear in one of the contexts prohibited by this rule. int32_t *p1 = 0; /* Non-compliant */ int32_t *p2 = (void *) 0; /* Compliant */ 101 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8:
Rules In the following example, the comparison between p2 and (void *)0 is compliant because the integer constant expression 0 appears as the operand of a cast and not in one of the contexts prohibited by this rule. #define MY_NULL_1 0 #define MY_NULL_2 (void *) 0 if (p1 == MY_NULL_1) { } if (p2 == MY_NULL_2) { } /* Non-compliant */ /*
Compliant */ The following example is compliant because use of the macro NULL provided by the implementation is always permitted, even if it expands to an integer constant expression with value 0. /* Could also be stdio.h, stdlib.h and others */ #include extern void f (uint8_t *p); /* Compliant for any conforming definition of NULL, such as: * 0 * (
void *) 0 * (((0))) * (((1 - 1))) */ f (NULL); See also Rule 11.4 102 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Rule 12.1 The precedence of operators within expressions should be made explicit Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules 8.12 Expressions Amplification The
following table is used in the definition of this rule. Description Operator or Operand Precedence Primary identifier, constant, string literal, (expression) 16 (high) Postfix [] () (function call) . -> ++ (post-increment) -- (post-decrement) () {} (C99: compound literal) 15 Unary ++ (pre-increment) -- (pre-decrement) & * + - ~ ! sizeof defined
(preprocessor) 14 Cast () 13 Multiplicative */% 12 Additive +- 11 Bitwise shift > 10 Relational < > = 9 Equality == != 8 Bitwise AND & 7 Bitwise XOR ^ 6 Bitwise OR | 5 Logical AND && 4 Logical OR || 3 Conditional ?: 2 Assignment = *= /= %= += -= = &= ^= |= 1 Comma , 0 (low) The precedences used in this table are chosen to allow a concise
description of the rule. They are not necessarily the same as those that might be encountered in other descriptions of operator precedence. For the purposes of this rule, the precedence of an expression is the precedence of the element (operand or operator) at the root of the parse tree for that expression. For example: the parse tree for the
expression a n Compliant - no need to write *(p++) Non-compliant - write either sizeof (x) + y or sizeof (x + y) */ */ */ The following example shows expressions containing operators at the same precedence level. All of these are compliant but, depending on the types of a, b and c, any expression with more than one operator may violate other rules.
a a (a a (+ + a + + a b; b + + b (b b + b c;) + + c c +) - c);) ; d; (c + d); The following example shows a variety of mixed-operator expressions: /* Compliant - no need to write f ((a + b), c) */ x = f (a + b, c); /* Non-compliant * Operands of conditional operator (precedence 2) are: * == precedence 8 needs parentheses * a precedence 16 does not
need parentheses * - precedence 11 needs parentheses */ x = a == b ? a : a - b; /* Compliant */ x = (a == b) ? a : (a - b); 104 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules /* Compliant * Operands of len * sizeof (uint32_t))); *s2 = *s1; /* Only copies s1->len */ return s2; } See also Dir 4.12, Rule 21.3 151 Licensed to: LEE
CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 18.8 Variable-length array types shall not be used C99 [Unspecified 21; Undefined 69, 70] Category Required Analysis Decidable, Single Translation Unit Applies to C99 Rationale Variable-length array types are specified when the size of an array declared in a block or a function prototype is
not an integer constant expression. They are typically implemented as a variable size object stored on the stack. Their use can therefore make it impossible to determine statically the amount of memory that must be reserved for a stack. If the size of a variable-length array is negative or zero, the behaviour is undefined. If a variable-length array is
used in a context in which it is required to be compatible with another array type, possibly itself variable-length, then the size of the array types shall be identical. Further, all sizes shall evaluate to positive integers. If these requirements are not met, the behaviour is undefined. If a variable-length array type is used in the operand of a sizeof operator,
under some circumstances it is unspecified whether the array size expression is evaluated or not. Each instance of a variable-length array type has its size fixed at the start of its lifetime. This gives rise to behaviour that might be confusing, for example: void f (void) { uint16_t n = 5; typedef uint16_t Vector[n]; /* An array type with 5 elements */ /*
An array type with 5 elements */ /* An array type with 7 elements */ n = 7; Vector a1; uint16_t a2[n]; } Example There is no use of variable-length arrays that is compliant with this rule. The examples show some of the undefined behaviour that can arise from their use. void f (int16_t n) { uint16_t vla[n]; } void g (void) { f (0); f (-1); f (10); } /*
Non-compliant - Undefined if n B #endif */ Section 8: Rules #if ! defined (X) /* Compliant #endif /* Compliant assuming A and B are numeric */ See also Rule 14.4 Rule 20.9 All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define’d before evaluation Category Required Analysis Decidable, Single
Translation Unit Applies to C90, C99 Amplification As well as using a #define preprocessor directive, identifiers may effectively be #define’d in other, implementation-defined, ways. For example some implementations support: • Using a compiler command-line option, such as -D to allow identifiers to be defined prior to translation; • Using
environment variables to achieve the same effect; • Pre-defined identifiers provided by the compiler. Rationale If an attempt is made to use a macro identifier in a preprocessor directive, and that identifier has not been defined, then the preprocessor will assume that it has a value of zero. This may not meet developer expectations. Example The
following examples assume that the macro M is undefined. #if M == 0 /* Non-compliant /* Does 'M' expand to zero or is it undefined? */ */ #endif #if defined (M) /* Compliant - M is not evaluated #if M == 0 /* Compliant - M is known to be defined /* 'M' must expand to zero. #endif #endif */ */ */ /* Compliant - B is only evaluated in (B == 0) if it is
defined */ #if defined (B) && (B == 0) #endif 161 Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 20.10 The # and ## preprocessor operators should not be used C90 [Unspecified 12; Undefined 51, 52], C99 [Unspecified 25; Undefined 3, 88, 89] Category Advisory Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale The order of evaluation associated with multiple #, multiple ##
or a mix of # and ## preprocessor operators is unspecified. In some cases it is therefore not possible to predict the result of macro expansion. The use of the ## operator can result in code that is obscure.
Note: Rule 1.3 covers the undefined behaviour that arises if either: • The result of a # operator is not a valid string literal; or • The result of a ## operator is not a valid preprocessing token. See also Rule 20.11 Rule 20.11 A macro parameter immediately following a # operator shall not immediately be followed by a ## operator C90 [Unspecified 12],
C99 [Unspecified 25] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. The use of # and ## is discouraged by Rule 20.10.
In particular, the result of a # operator is a string literal and it is extremely unlikely that pasting this to any other preprocessing token will result in a valid token. Example #define A(x) #define B(x, y) #define C(x, y) #x x ## y #x ## y /* Compliant */ /* Compliant */ /* Non-compliant */ See also Rule 20.10 162 Licensed to: LEE CHING MIN. 5 Mar
2019. Copy 1 of 1 A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Section 8: Rules Rule 20.12 Rationale A macro parameter that is used as an
operand of a # or ## operator is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement may not meet developer expectations. Example In the following non-compliant example,
the macro parameter x is replaced with AA which is subject to further macro replacement when not used as the operand of ##. #define AA #define BB(x) 0xffff (x) + wow ## x /* Non-compliant */ void f (void) { int32_t wowAA = 0; /* Expands as wowAA = (0xffff) + wowAA; */ wowAA = BB (AA); } In the following compliant example, the macro
parameter X is not subject to further macro replacement. int32_t speed; int32_t speed_scale; int32_t scaled_speed; #define SCALE(X) ((X) * X ## _scale) /* expands to scaled_speed = ((speed) * speed_scale); */ scaled_speed = SCALE (speed); Rule 20.13 A line whose first token is # shall be a valid preprocessing directive Category Required
Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification White-space is permitted between the # and preprocessing tokens. Rationale A preprocessor directive may be used to conditionally exclude source code until a corresponding #else, #elif or #endif directive is encountered. A malformed or invalid preprocessing directive
contained Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 163 Section 8: Rules within the excluded source code may not be detected by the compiler, possibly leading to the exclusion of more code than was intended. Requiring all preprocessor directives to be syntactically valid, even when they occur within an excluded block of code, ensures
that this cannot happen. Example In the following example all the code between the #ifndef and #endif directives may be excluded if AAA is defined. The developer intended that AAA be assigned to x, but the #else directive was entered incorrectly and not diagnosed by the compiler. #define AAA 2 int32_t foo (void) { int32_t x = 0; #ifndef AAA x =
1; #else1 x = AAA; #endif /* Non-compliant */ return x; } The following example is compliant because the text #start appearing in a comment is not a token. /* #start is not a token in a comment */ Rule 20.14 All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related
Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Rationale Confusion can arise when blocks of code are included or excluded by the use of conditional compilation directives which are spread over multiple files. Requiring that a #if directive be terminated within the same file reduces the visual complexity of the code
and the chance that errors will be made during maintenance. Note: #if directives may be used within included files provided they are terminated within the same file.
Example /* file1.c */ #ifdef A #include "file1.h" #endif /* End of file1.c */ /* Compliant */ 164 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 /* file1.h */ #if 1 #endif /* End of file1.h */ Section 8: Rules /* file2.c */ #if 1 #include "file2.h" /* End of file2.c*/ /* Non-compliant */ /* Compliant */ /* file2.h */ #endif /* End of file1.h */ 8.21 Standard
libraries Rule 21.1 #define and #undef shall not be used on a reserved identifier or reserved macro name C90 [Undefined 54, 57, 58, 62, 71] C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification This rule applies to the following: • Identifiers or macro
names beginning with an underscore; • Identifiers in file scope described in Section 7, “Library”, of The Standard; • Macro names described in Section 7, “Library”, of The Standard as being defined in a standard header.
This rule also prohibits the use of #define or #undef on the identifier defined as this results in explicitly undefined behaviour.
This rule does not include those identifiers or macro names that are described in the section of the applicable C standard entitled “Future Library Directions”. The Standard states that defining a macro with the same name as: • A macro defined in a standard header, or • An identifier with file scope declared in a standard header is well-defined
provided that the header is not included. This rule does not permit such definitions on the grounds that they are likely to cause confusion. Note: the macro NDEBUG is not defined in a standard header and may therefore be #define’d. Rationale Reserved identifiers and reserved macro names are intended for use by the implementation.
Removing or changing the meaning of a reserved macro may result in undefined behaviour. Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 165 Section 8: Rules Example #undef __LINE__ #define _GUARD_H 1 #undef _BUILTIN_sqrt /* /* /* * * * Non-compliant - begins with _ Non-compliant - begins with _ Non-compliant - the implementation
may use _BUILTIN_sqrt for other purposes, e.g. generating a sqrt instruction #define defined /* Non-compliant #define errno my_errno /* Non-compliant #define isneg(x) ((x) < 0) /* Compliant * * */ */ */ reserved identifier */ library identifier */ rule doesn't include future library directions */ See also Rule 20.4 Rule 21.2 A reserved identifier or
macro name shall not be declared C90 [Undefined 57, 58, 64, 71], C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification See the Amplification for Rule 21.1 for a description of the relevant identifiers and macro names. Rationale The implementation is
permitted to rely on reserved identifiers behaving as described in The Standard and may treat them specially. If reserved identifiers are reused, the program may exhibit undefined behaviour. Example In the following non-compliant example, the function memcpy is declared explicitly. The compliant method of declaring this function is to include . /* *
Include to define size_t */ #include extern void *memcpy (void *restrict s1, const void *restrict s2, size_t n); An implementation is permitted to provide a function-like macro definition for each Standard Library function in addition to the library function itself. This feature is often used by compiler writers to generate efficient inline operations in place
of the call to a library function. Using a function-like macro, the call to a library function can be replaced with a call to a reserved function that is detected by the compiler’s code generation phase and replaced with the inline operation.
For example, the fragment of that declares sqrt might be written using a function-like macro that generates a call to _BUILTIN_sqrt which is replaced with an inline SQRT instruction on processors that support it: extern double sqrt (double x); 166 #define sqrt(x) (_BUILTIN_sqrt (x)) Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 #define
_BUILTIN_sqrt(x) (x) #include /* Non-compliant float64_t x = sqrt ((float64_t) 2.0); /* sqrt may not behave as * defined in The Standard */ Rule 21.3 */ Section 8: Rules The following non-compliant code might interfere with the compiler’s built-in mechanism for handling sqrt and therefore produce undefined behaviour: The memory allocation and
deallocation functions of shall not be used C90 [Unspecified 19; Undefined 9, 91, 92; Implementation 69] C99 [Unspecified 39, 40; Undefined 8, 9, 168–171; Implementation J.3.12(35)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification The identifiers calloc, malloc, realloc and free shall not be used and
no macro with one of these names shall be expanded. Rationale Use of dynamic memory allocation and deallocation routines provided by The Standard Library can lead to undefined behaviour, for example: • Memory that was not dynamically allocated is subsequently freed; • A pointer to freed memory is used in any way; • Accessing allocated
memory before storing a value into it. Note: this rule is a specific instance of Dir 4.12. See also Dir 4.12, Rule 18.7, Rule 22.1, Rule 22.2 167 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 21.4 The standard header file shall not be used C90 [Unspecified 14; Undefined 64–67] C99 [Unspecified 32; Undefined 118–121,
173] [Koenig 74] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification None of the facilities that are specified as being provided by shall be used. Rationale setjmp and longjmp allow the normal function call mechanisms to be bypassed. Their use may lead to undefined and unspecified behaviour. Rule 21.5
The standard header file shall not be used C90 [Undefined 67–69; Implementation 48–52] C99 [Undefined 122–127; Implementation J.3.12(12)] [Koenig 74] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification None of the facilities that are specified as being provided by shall be used. Rationale Signal
handling contains implementation-defined and undefined behaviour. Rule 21.6 The Standard Library input/output functions shall not be used C90 [Unspecified 2–5, 16–18; Undefined 77–89; Implementation 53–68] C99 [Unspecified 3–6, 34–37; Undefined 138–166, 186; Implementation J.3.12(14–32)] Category Required Analysis Decidable, Single
Translation Unit Applies to C90, C99 Amplification This rule applies to the functions that are specified as being provided by and, in C99, their wide-character equivalents specified in Sections 7.24.2 and 7.24.3 of the C99 Standard as being provided by . None of these identifiers shall be used and no macro with one of these names shall be expanded.
168 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Streams and file I/O have unspecified, undefined and implementation-defined behaviours associated with them.
See also Rule 22.1, Rule 22.3, Rule 22.4, Rule 22.5, Rule 22.6 Rule 21.7 Section 8: Rules Rationale The atof, atoi, atol and atoll functions of shall not be used C90 [Undefined 90], C99 [Undefined 113] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification The identifiers atof, atoi, atol and, for C99 only atoll,
shall not be used and no macro with one of these names shall be expanded. Rationale These functions have undefined behaviour associated with them when the string cannot be converted. Rule 21.8 The library functions abort, exit, getenv and system of shall not be used C90 [Undefined 93; Implementation 70–73] C99 [Undefined 172, 174, 175;
Implementation J.3.12(36–38)] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification The identifiers abort, exit, getenv and system shall not be used and no macro with one of these names shall be expanded. Rationale These functions have undefined and implementation-defined behaviours associated with
them. 169 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 21.9 The library functions bsearch and qsort of shall not be used C90 [Unspecified 20, 21], C99 [Unspecified 41, 42; Undefined 176–178] Category Required Analysis Decidable, Single Translation Unit Applies to C90, C99 Amplification The identifiers bsearch and
qsort shall not be used and no macro with one of these names shall be expanded. Rationale If the comparison function does not behave consistently when comparing elements, or it modifies any of the elements, the behaviour is undefined. Note: the unspecified behaviour, which relates to the treatment of elements that compare as equal, can be
avoided by ensuring that the comparison function never returns 0. When two elements are otherwise equal, the comparison function could return a value that indicates their relative order in the initial array. The implementation of qsort is likely to be recursive and will therefore place unknown demands on stack resource. This is of concern in
embedded systems as the stack is likely to be a fixed, often small, size. Rule 21.10 The Standard Library time and date functions shall not be used C90 [Unspecified 22; Undefined 80, 97; Implementation 75, 76] C99 [Unspecified 43, 44; Undefined 146, 154, 182; Implementation J.3.12(39–42)] Category Required Analysis Decidable, Single Translation
Unit Applies to C90, C99 Amplification None of the facilities that are specified as being provided by shall be used. In C99, the identifier wcsftime shall not be used and no macro with this name shall be expanded. Rationale The time and date functions have unspecified, undefined and implementation-defined behaviours associated with them. 170
Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The standard header file shall not be used C99 [Undefined 184, 185] Category Required Analysis Decidable, Single Translation Unit Applies to C99 Section 8: Rules Rule 21.11 Amplification None of the facilities that are specified as being provided by shall be used. Rationale Using the facilities of
may result in undefined behaviour.
Example #include float f1, f2; void f (void) { f1 = sqrt (f2); } /* Non-compliant - generic sqrt used */ #include float f1, f2; void f (void) { f1 = sqrtf (f2); } Rule 21.12 /* Compliant - float version of sqrt used */ The exception handling features of should not be used C99 [Unspecified 27, 28; Undefined 109–111; Implementation J.3.6(8)] Category
Advisory Analysis Decidable, Single Translation Unit Applies to C99 Amplification The identifiers feclearexcept, fegetexceptflag, feraiseexcept, fesetexceptflag and fetestexcept shall not be used and no macro with one of these names shall be expanded. The macros FE_INEXACT, FE_DIVBYZERO, FE_UNDERFLOW, FE_OVERFLOW, FE_INVALID and
FE_ALL_EXCEPT, along with any implementation-defined floating-point exception macros, shall not be used. Rationale In some circumstances, the values of the floating-point status flags are unspecified and attempts to access them may lead to undefined behaviour. 171 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules The
order in which exceptions are raised by the feraiseexcept function is unspecified and could therefore result in a program that has been designed for a certain order not operating correctly. Example #include void f (float32_t x, float32_t y) { float32_t z; feclearexcept (FE_DIVBYZERO); /* Non-compliant */ z = x / y; if (fetestexcept (FE_DIVBYZERO)
) { } /* Non-compliant */ else { #pragma STDC FENV_ACCESS ON z = x * y; } if (z > x) { #pragma STDC FENV_ACCESS OFF if (fetestexcept (FE_OVERFLOW)) { } /* Non-compliant */ } } 8.22 Resources Many of the rules in this section are applicable only when rules in other sections have been deviated. Rule 22.1 All resources obtained
dynamically by means of Standard Library functions shall be explicitly released Category Required Analysis Undecidable, System Applies to C90, C99 Amplification The Standard Library functions that allocate resources are malloc, calloc, realloc and fopen. Rationale If resources are not explicitly released then it is possible for a failure to occur due to
exhaustion of those resources. Releasing resources as soon as possible reduces the possibility that exhaustion will occur. 172 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 #include int main (void) { void *b = malloc (40); /* Non-compliant - dynamic memory not released */ return 1; Section 8: Rules Example } #include int main (void) {
FILE *fp = fopen ("tmp", "r"); /* Non-compliant - file not closed return 1; */ } In the following non-compliant example, the handle on “tmp-1” is lost when “tmp-2” is opened.
#include int main (void) { FILE *fp; fp = fopen ("tmp-1", "w"); fprintf (fp, "*"); /* File "tmp-1" should be closed here, but stream 'leaks'. */ fp = fopen ("tmp-2", "w"); fprintf (fp, "!"); fclose (fp); return (0); } See also Dir 4.12, Dir 4.13, Rule 21.3, Rule 21.6 173 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 22.2 A
block of memory shall only be freed if it was allocated by means of a Standard Library function C90 [Undefined 92], C99 [Undefined 169] Category Mandatory Analysis Undecidable, System Applies to C90, C99 Amplification The Standard Library functions that allocate memory are malloc, calloc and realloc.
A block of memory is freed when its address is passed to free and potentially freed when its address is passed to realloc. Once freed, a block of memory is no longer considered to be allocated and therefore cannot subsequently be freed again. Rationale Freeing non-allocated memory, or freeing the same allocated memory more than once leads to
undefined behaviour. Example #include void fn (void) { int32_t a; /* Non-compliant - a does not point to allocated storage free (&a); */ } void g (void) { char *p = (char *) malloc (512); char *q = p; free (p); /* Non-compliant - allocated block freed a second time free (q); */ /* Non-compliant - allocated block may be freed a third time */ p = (
char *) realloc (p, 1024); } See also Dir 4.12, Dir 4.13, Rule 21.3 174 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 The same file shall not be open for read and write access at the same time on different streams C90 [Implementation 61], C99 [Implementation J.3.12(22)] Category Required Analysis Undecidable, System Applies to C90, C99
Section 8: Rules Rule 22.3 Amplification This rule applies to files opened with The Standard Library functions. It may also apply to similar features provided by the execution environment. Rationale The Standard does not specify the behaviour if a file is both written and read via different streams. Note: it is acceptable to open a file multiple times for
read-only access. Example #include void fn (void) { FILE *fw = fopen ("tmp", "r+"); FILE *fr = fopen ("tmp", "r"); } /* "r+" opens for read/write */ /* Non-compliant */ See also Rule 21.6 175 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Section 8: Rules Rule 22.4 There shall be no attempt to write to a stream which has been opened as
read-only Category Mandatory Analysis Undecidable, System Applies to C90, C99 Rationale The Standard does not specify the behaviour if an attempt is made to write to a read-only stream. For this reason it is considered unsafe to write to a read-only stream. Example #include void fn (void) { FILE *fp = fopen ("tmp", "r"); (void) fprintf (fp, "What
happens now?"); /* Non-compliant */ (void) fclose (fp); } See also Rule 21.6 Rule 22.5 A pointer to a FILE object shall not be dereferenced Category Mandatory Analysis Undecidable, System Applies to C90, C99 Amplification A pointer to a FILE object shall not be dereferenced directly or indirectly (e.g. by a call to memcpy or memcmp). Rationale
The Standard (C90 Section 7.9.3(6), C99 Section 7.19.3(6)) states that the address of a FILE object used to control a stream may be significant and a copy of the object may not give the same behaviour. This rule ensures that such a copy cannot be made. The direct manipulation of a FILE object is prohibited as this may be incompatible with its use as
a stream designator. Example #include 176 FILE *pf1; FILE *pf2; FILE f3; Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 /* Compliant */ /* Non-compliant */ The following example assumes that FILE * specifies a complete type with a member named pos: pf1->pos = 0; /* Non-compliant */ See also Rule 21.6 Rule 22.6 Section 8: Rules pf2 = pf1; f3 = *pf2; The value of a pointer to a FILE shall not be used after the associated stream has been closed
C99 [Undefined 140] Category Mandatory Analysis Undecidable, System Applies to C90, C99 Rationale The Standard states that the value of a FILE pointer is indeterminate after a close operation on a stream. Example #include void fn (void) { FILE *fp; void *p; fp = fopen ("tmp", "w"); if (fp == NULL) { error_action ( ); } fclose (fp); fprintf (fp,
"?"); p = fp; /* Non-compliant */ /* Non-compliant */ } See also Dir 4.13, Rule 21.6 177 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 9 References
[1] MISRA Guidelines for the Use of the C Language In Vehicle Based Software, ISBN 0-9524159-9-0, Motor Industry Research Association, Nuneaton, April 1998 [2] ISO/IEC 9899:1990,
Programming languages — C, International Organization for Standardization, 1990 [3] Hatton L., Safer C – Developing Software for High-integrity and Safety-critical Systems, ISBN 0-07707640-0, McGraw-Hill, 1994 [4] ISO/IEC 9899:1990/COR 1:1995, Technical Corrigendum 1, 1995 [5] ISO/IEC 9899:1990/AMD 1:1995, Amendment 1, 1995 [6]
ISO/IEC 9899:1990/COR 2:1996, Technical Corrigendum 2, 1996 [7] ANSI X3.159-1989, Programming languages — C, American National Standards Institute, 1989 [8] ISO/IEC 9899:1999, Programming languages — C, International Organization for Standardization, 1999 [9] ISO/IEC 9899:1999/COR 1:2001, Technical Corrigendum 1, 2001 [10]
ISO/IEC 9899:1999/COR 2:2004, Technical Corrigendum 2, 2004 [11] ISO/IEC 9899:1999/COR 3:2007, Technical Corrigendum 3, 2007 [12] ISO/IEC 9899:1999 Committee Draft WG14/N1256, Programming languages — C, International Organization for Standardization, September 2007 [13] ISO/IEC 9899:2011, Programming languages — C,
International Organization for Standardization, 2011 [14] ISO/IEC 9899:2011/COR 1:2012, Technical Corrigendum 1, 2012 [15] MISRA Development Guidelines for Vehicle Based Software, ISBN 0-9524156-0-7, Motor Industry Research Association, Nuneaton, November 1994 [16] MISRA AC AGC Guidelines for the application of MISRA-C:2004 in the
context of automatic code generation, ISBN 978-1-906400-02-6, MIRA Limited, Nuneaton, November 2007 [17] MISRA AC GMG Generic modelling design and style guidelines, ISBN 978-1-906400-06-4, MIRA Limited, Nuneaton, May 2009 [18] MISRA AC SLSF Modelling design and style guidelines for the application of Simulink and Stateflow, ISBN

978-1-906400-07-1, MIRA Limited, Nuneaton, May 2009 [19] MISRA AC TL Modelling style guidelines for the application of Targetlink in the context of automatic code generation, ISBN 978-1-906400-01-9, MIRA Limited, Nuneaton, November 2007 [20] CRR80, The Use of Commercial Off-the-Shelf (COTS) Software in Safety Related Applications, ISBN 0-7176-0984-7, HSE Books [21] ISO 9001:2008, Quality management systems — Requirements, International Organization for Standardization, 2008 178 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 ISO 90003:2004, Software engineering — Guidelines for the application of ISO 9001:2000 to computer software, ISO, 2004 [23] ISO 26262:2011, Road vehicles — Functional safety, ISO, 2011 [24] DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification, RTCA, 2011 [25] The TickIT Guide, Using ISO 9001:2000 for Software Quality Management System Construction, Certification and Continual Improvement, Issue 5, British Standards Institution, 2001 [26] Straker D., C Style: Standards and Guidelines, ISBN 0–13-116898-3, Prentice Hall 1991 [27] Fenton N.E. and Pfleeger S.L., Software Metrics: A Rigorous and Practical Approach, 2nd Edition, ISBN 0-534-95429-1, PWS, 1998 [28] MISR A Report 5 Software Metrics, Motor Industry Research Association, Nuneaton, February 1995 [29] MISR A Report 6 Verification and Validation, Motor Industry Research Association, Nuneaton, February 1995 [30] Kernighan B.W., Ritchie D.M., The C programming language, 2nd edition, ISBN 0-13-110362-8, Prentice Hall, 1988 (note: The 1st edition is not a suitable reference document as it does not describe ANSI/ISO C) [31] Koenig A., C Traps and Pitfalls, ISBN 0-201-17928-8, Addison-Wesley, 1988 [32] IEC 61508:2010, Functional safety of electrical/electronic/programmable electronic safety-related systems, International Electromechanical Commission, in 7 parts published in 2010 [33] EN 50128:2011, Railway applications — Communications, signalling and processing systems — Software for railway control and protec tion, CENELEC, 2011 [34] IEC 62304:2006, Medical device software — Software life cycle processes, IEC, 2006 [35] ANSI/IEEE Std 754, IEEE Standard for Binary Floating-Point Arithmetic, 1985 [36] ISO/IEC 10646:2003, Information technology — Universal Multiple-Octet Coded Character Set (UCS), International Organization for Standardization, 2003 [37] Goldberg D., What Every Computer Scientist Should Know about Floating-Point Arithmetic, Computing Surveys, March 1991 [38] Software Engineering Center, Information-technology Promotion Agency, Japan (IPA/SEC), Embedded System development Coding Reference (ESCR) [C language edition] Version 1.1, SEC Books, 2012 Section 9: References [22] 179 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Appendix A Summary of guidelines The implementation Dir 1.1 Required Any implementation-defined behaviour on which the output of the program depends shall be documented and understood Compilation and build Dir 2.1 Required All source files shall compile without any compilation errors Requirements traceability Dir 3.1 Required All code shall be traceable to documented requirements Code design Dir 4.1 Required Run-time failures shall be minimized Dir 4.2 Advisory All usage of assembly language should be documented Dir 4.3 Required Assembly language shall be encapsulated and isolated Dir 4.4 Advisory Sections of code should not be “commented out” Dir 4.5 Advisory Identifiers in the same name space with overlapping visibility should be typographically unambiguous Dir 4.6 Advisory typedefs that indicate size and signedness should be used in place of the basic numerical types Dir 4.7 Required If a function returns error information, then that error information shall be tested Dir 4.8 Advisory If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden Dir 4.9 Advisory A function should be used in preference to a function-like macro where they are interchangeable Dir 4.10 Required Precautions shall be taken in order to prevent the contents of a header file being included more than once Dir 4.11 Required The validity of values passed to library functions shall be checked Dir 4.12 Required Dynamic memory allocation shall not be used Dir 4.13 Advisory Functions which are designed to provide operations on a resource should be called in an appropriate sequence 180 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Rule 1.1 Required The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits Rule 1.2 Advisory Language extensions should not be used Rule 1.3 Required There shall be no occurrence of undefined or critical unspecified behaviour Unused code Rule 2.1 Required A project shall not contain unreachable code Rule 2.2 Required There shall be no dead code Rule 2.3 Advisory A project should not contain unused type declarations Rule 2.4 Advisory A project should not contain unused tag declarations Rule 2.5 Advisory A project should not contain unused macro declarations Rule 2.6 Advisory A function should not contain unused label declarations Rule 2.7 Advisory There should be no unused parameters in functions Appendix A: Summary of guidelines A standard C environment Comments Rule 3.1 Required The character sequences /* and // shall not be used within a comment Rule 3.2 Required Line-splicing shall not be used in // comments Character sets and lexical conventions Rule 4.1 Required Octal and hexadecimal escape sequences shall be terminated Rule 4.2 Advisory Trigraphs should not be used Identifiers Rule 5.1 Required External identifiers shall be distinct Rule 5.2 Required Identifiers declared in the same scope and name space shall be distinct Rule 5.3 Required An identifier declared in an inner scope shall not hide an identifier declared in an outer scope Rule 5.4 Required Macro identifiers shall be distinct Rule 5.5 Required Identifiers shall be distinct from macro names Rule 5.6 Required A typedef name shall be a unique identifier Rule 5.7 Required A tag name shall be a unique identifier Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 181 Appendix A: Summary of guidelines Rule 5.8 Required Identifiers that define objects or functions with external linkage shall be unique Rule 5.9 Advisory Identifiers that define objects or functions with internal linkage should be unique Rule 6.1 Required Bit-fields shall only be declared with an appropriate type Rule 6.2 Required Single-bit named bit fields shall not be of a signed type Types Literals and constants Rule 7.1 Required Octal constants shall not be used Rule 7.2 Required A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type Rule 7.3 Required The lowercase character “l” shall not be used in a literal suffix Rule 7.4 Required A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char” Declarations and definitions 182 Rule 8.1 Required Types shall be explicitly specified Rule 8.2 Required Function types shall be in prototype form with named parameters Rule 8.3 Required All declarations of an object or function shall use the same names and type qualifiers Rule 8.4 Required A compatible declaration shall be visible when an object or function with external linkage is defined Rule 8.5 Required An external object or function shall be declared once in one and only one file Rule 8.6 Required An identifier with external linkage shall have exactly one external definition Rule 8.7 Advisory Functions and objects should not be defined with external linkage if they are referenced in only one translation unit Rule 8.8 Required The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage Rule 8.9 Advisory An object should be defined at block scope if its identifier only appears in a single function Rule 8.10 Required An inline function shall be declared with the static storage class Rule 8.11 Advisory When an array with external linkage is declared, its size should be explicitly specified Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Required Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique Rule 8.13 Advisory A pointer should point to a const-qualified type whenever possible Rule 8.14 Required The restrict type qualifier shall not be used Initialization Rule 9.1 Mandatory The value of an object with automatic storage duration shall not be read before it has been set Rule 9.2 Required The initializer for an aggregate or union shall be enclosed in braces Rule 9.3 Required Arrays shall not be partially initialized Rule 9.4 Required An element of an object shall not be initialized more than once Rule 9.5 Required Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly Appendix A: Summary of guidelines Rule 8.12 The essential type model Rule 10.1 Required Operands shall not be of an inappropriate essential type Rule 10.2 Required Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations Rule 10.3 Required The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category Rule 10.4 Required Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category Rule 10.5 Advisory The value of an expression should not be cast to an inappropriate essential type Rule 10.6 Required The value of a composite expression shall not be assigned to an object with wider essential type Rule 10.7 Required If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type Rule 10.8 Required The value of a composite expression shall not be cast to a different essential type category or a wider essential type Pointer type conversions Rule 11.1 Required Conversions shall not be performed between a pointer to a function and any other type Rule 11.2 Required Conversions shall not be performed between a pointer to an incomplete type and any other type Rule 11.3 Required A cast shall not be performed between a pointer to object type and a pointer to a different object type Licensed to: LEE CHING MIN.
5 Mar 2019. Copy 1 of 1 183 Appendix A: Summary of guidelines Rule 11.4 Advisory A conversion should not be performed between a pointer to object and an integer type Rule 11.5 Advisory A conversion should not be performed from pointer to void into pointer to object Rule 11.6 Required A cast shall not be performed between pointer to void and an arithmetic type Rule 11.7 Required A cast shall not be performed between pointer to object and a noninteger arithmetic type Rule 11.8 Required A cast shall not remove any const or volatile qualification from the type pointed to by a pointer Rule 11.9 Required The macro NULL shall be the only permitted form of integer null pointer constant Expressions Rule 12.1 Advisory The precedence of operators within expressions should be made explicit Rule 12.2 Required The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand Rule 12.3 Advisory The comma operator should not be used Rule 12.4 Advisory Evaluation of constant expressions should not lead to unsigned integer wrap-around Side effects Rule 13.1 Required Initializer lists shall not contain persistent side effects Rule 13.2 Required The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders Rule 13.3 Advisory A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator Rule 13.4 Advisory The result of an assignment operator should not be used Rule 13.5 Required The right hand operand of a logical && or || operator shall not contain persistent side effects Rule 13.6 Mandatory The operand of the sizeof operator shall not contain any expression which has potential side effects 184 Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 Rule 14.1 Required A loop counter shall not have essentially floating type Rule 14.2 Required A for loop shall be well-formed Rule 14.3 Required Controlling expressions shall not be invariant Rule 14.4 Required The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type Control flow Rule 15.1 Advisory The goto statement should not be used Rule 15.2 Required The goto statement shall jump to a label declared later in the same function Rule 15.3 Required Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement Rule 15.4 Advisory There should be no more than one break or goto statement used to terminate any iteration statement Rule 15.5 Advisory A function should have a single point of exit at the end Rule 15.6 Required The body of an iteration-statement or a selection-statement shall be a compound-statement Rule 15.7 Required All if … else if constructs shall be terminated with an else statement Appendix A: Summary of guidelines Control statement expressions Switch statements Rule 16.1 Required All switch statements shall be well-formed Rule 16.2 Required A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement Rule 16.3 Required An unconditional break statement shall terminate every switch-clause Rule 16.4 Required Every switch statement shall have a default label Rule 16.5 Required A default label shall appear as either the first or the last switch label of a switch statement Rule 16.6 Required Every switch statement shall have at least two switch-clauses Rule 16.7 Required A switch-expression shall not have essentially Boolean type Functions Rule 17.1 Required The features of shall not be used Rule 17.2 Required Functions shall not call themselves, either directly or indirectly Rule 17.3 Mandatory A function shall not be declared implicitly Licensed to: LEE CHING MIN. 5 Mar 2019. Copy 1 of 1 185 Appendix A: Summary of guidelines Rule 17.4 Mandatory All exit paths from a function with non-void return type shall have an explicit return statement with an expression Rule 17.5 Advisory The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements Rule 17.6 Mandatory The declaration of an array parameter shall not contain the static keyword between the [ ] Rule 17.7 Required The value returned by a function having non-void return type shall be used Rule 17.8 Advisory A function parameter should not be modified Pointers and arrays Rule 18.1 Required A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand Rule 18.2 Required Subtraction between pointers shall only be applied to pointers that address elements of the same array Rule 18.3 Required The relational operators >, >=, < and MISRA C:2012 TC2 We are pleased to announce the publication of MISRA C:2012 Technical Corrigendum 2 (MISRA C:2012 TC2). This document provides additional clarification on MISRA C:2012 and should be read in conjunction with either MISRA C:2012 (Third Edition, First Revision) Guidelines for the use of the C language in critical systems, as revised by MISRA C:2012 Amendment 2, Updates for ISO/IEC 9899:2011 Core functionality or MISRA C:2012 (Third Edition) Guidelines for the use of the C language in critical systems, as revised by: MISRA C:2012 Amendment 1, Additional security guidelines for MISRA C:2012 MISRA C:2012 Amendment 2, Updates for ISO/IEC 9899:2011 Core functionality MISRA C:2012 Technical Corrigendum 1 The document can be downloaded from the “Resources” section of the Forum or directly from