The Motor Industry Software Reliability Association

MISRA AC AGC

Guidelines for the application of MISRA-C:2004 in the context of automatic code generation

November 2007
Version 1.0

First published November 2007
by MIRA Limited
Watling Street
Nuneaton
Warwickshire CV10 0TU
UK
www.misra.org.uk

ISBN-978-1-906400-02-6

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

The Motor Industry Software Reliability Association

MISRA AC AGC

Guidelines for the application of MISRA-C:2004 in the context of automatic code generation

November 2007

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

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

Disclaimer

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

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

Executive summary

The MISRA AC family of documents deals with the application of language subsets for automatic code generation purposes. This document, MISRA AC AGC, contains guidance for the application of the MISRA C rules to C code that has been generated automatically from a higher-level model. The guidance it provides is generic and is therefore applicable to all modelling languages and code generators.

Acknowledgements

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

Gunter Blache	ETAS GmbH
Mirko Conrad	DaimlerChrysler AG
Simon Fürst	BMW AG
Christoph Jung	BMW AG
Michael Jungmann	BMW AG
Dietmar Kant	Audi Electronics Venture GmbH
Sebastian Krüger	Audi Electronics Venture GmbH
Marc Lalo	PolySpace Technologies
Pierre R. Mai	PMSF IT Consulting
Martin Meyer	dSPACE GmbH
Frédèric Mognol	Esterel Technologies
Steve Montgomery	Ricardo UK Limited
Stefan-Alexander Schneider	BMW AG
Thomas Wengler	BMW AG

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

Naveen Alwandi
Richard Anderson
Jens Berger
Matt Boesch
Paul Burden
Andrew Burnard
Simone Dozio
Jean-Marc Dressler
Tom Erkkinen
Nicolas Francois
Bob Frankel
Valeria Franzitta
Mike Hennell
Chris Hills
Jacob John
Anders Kallerdahl
Igor Klimchynski
BC Manjunath
Matteo Marioni
Gavin McCall
Paolo Panaroni
Aravind Pillarisetti
Bill Potter
Tobias Schweickhardt
Thomas Wenzel
Liz Whiting

Contents

1. Background — automatic code generators and MISRA C 1

2. Scope and applicability 3
   - 2.1 Applicability 3
   - 2.2 Prerequisite knowledge 3

3. Use of the MISRA AC AGC Guidelines 4
   - 3.1 Primary Categories 4
   - 3.2 Secondary Categories 5

4. Compliance 7

5. Target optimizations and automatic code generators 8

6. Rules 10
   - 6.1 Environment 10
   - 6.2 Language Extensions 11
   - 6.3 Documentation 12
   - 6.4 Character sets 14
   - 6.5 Identifiers 14
   - 6.6 Types 16
   - 6.7 Constants 17
   - 6.8 Declarations and definitions 17
   - 6.9 Initialisation 19
   - 6.10 Arithmetic type conversions 20
   - 6.11 Pointer type conversions 22
   - 6.12 Expressions 23
   - 6.13 Control statement expressions 26
   - 6.14 Control flow 28
   - 6.15 Switch statements 31
   - 6.16 Functions 32
   - 6.17 Pointers and arrays 35
   - 6.18 Structures and unions 36
   - 6.19 Preprocessing directives 37
   - 6.20 Standard libraries 41
   - 6.21 Run-time failures 43

7. References 44

1. Background — automatic code generators and MISRA C

1. Background — automatic code generators and MISRA C

The use of automatic code generators in production automotive control applications has increased dramatically in the last few years. The use of modelling packages and automatic code generation has brought a number of benefits to the development process but it has also introduced a new set of pitfalls.

The application of the MISRA C Guidelines [1] in the context of automatic code generators has often been a point of contention and confusion. It could be argued that the MISRA C Guidelines were developed for human programmers and are therefore not applicable to automatically generated code. This point of view is supported by the fact that the developer is insulated from the pitfalls of the C language by the modelling package and the automatic code generator. The counter-argument to this point of view is that the MISRA C Guidelines embody good programming practice and adopting them for automatically generated code gives an additional level of protection from pitfalls in both C **and** in the modelling language.

The MISRA C Guidelines themselves provide little additional guidance on their application to automatically generated code. They do suggest that:

- the degree to which code generated by a tool is compliant with the MISRA C Guidelines could be one criterion by which automatic code generation tools might be assessed, and
- automatically generated code should be treated in the same manner as manually produced code for verification and validation purposes, i.e. that it **should** be MISRA C compliant.

The suggestion that automatically generated code should be MISRA C compliant leads to issues for both the model developer and the code generator vendor. The model developer has only limited control over the C code generated and generally feels unable and understandably unwilling to handle all MISRA C conformance issues directly. The code generator vendors are also unable to deal with all issues relating to MISRA C conformance as they have only limited control over the C code that can be generated for a given input model while preserving the intent of the model developer.

The rule prohibiting floating point equality and inequality tests (MISRA C Rule 13.3) is a good case in point. Suppose that a model developer has used modelling constructs that compare floating-point numbers for equality or inequality. The code generator must generate the corresponding floating-point tests in the C source if it is to preserve the semantics of the model faithfully, clearly something that would reasonably be expected of the code generator. However, there are other instances where the code generator might introduce comparisons that are not directly suggested by the model. An example would be performing a comparison to guard against division by zero, something which the model developer has little or no control over. In this case, an exact equality or inequality test is precisely what is required since the code generator has no knowledge of the application and therefore cannot perform a range-based test instead. This example illustrates one

1. Background — automatic code generators and MISRA C (continued)

of the key difficulties with requiring 100% compliance of automatically generated code with the requirements of MISRA C.

This difficulty has in practice resulted in the use or enforcement of *ad-hoc* subsets of MISRA C. These subsets have often been designed around the capabilities of the code generator being used rather than any deeper analysis of the applicability of certain rules or the need for additional rules at the model level.

The MISRA AC AGC document is intended to help users and implementers of automatic code generators in implementing the MISRA C guidelines. It provides a framework for understanding the concerns that each of the individual rules is trying to address. Using this framework, the reader can decide whether individual rules are applicable in a given project setting or for a given use of a code generator and whether additional rules are required at the model level to address the concerns underlying any given MISRA C rule.

2. Scope and applicability

2.1 Applicability

The MISRA AC AGC Guidelines are designed for application to all production code in an automotive embedded system that has been generated from a model by the use of an automatic code generator. They are **not** applicable to any code that has **not** generated by an automatic code generator. Any code that has been generated manually should comply with the requirements of MISRA C. Such code might be an entire module of manually generated code or it might be manually generated C code embedded within the model that has been injected *verbatim* into the automatically generated code

While the MISRA AC AGC Guidelines make a number of recommendations to be put in place at the model level, these do not constitute a full set of recommendations that are to be considered when drawing up suitable modelling guidelines for a graphical modelling language. Many other issues will have to be taken into account when selecting modelling languages and drawing up suitable modelling guidelines. As a general guideline it is highly desirable to use modelling languages that have well-defined semantics in order that the behaviour of the model, and therefore the generated C code, can be predicted and reasoned about. A full treatment of modelling package guidelines is outside the scope of this document but is addressed by other documents in the MISRA AC family as described in MISRA AC INT [2].

2.2 Prerequisite knowledge

This document is not intended to be an introduction or training aid to the subjects it covers. It is assumed that readers of this document are familiar with the ISO/IEC 9899:1990 C programming language standard [3], including associated technical corrigenda and amendments, and have access to these primary reference documents. It is also assumed that readers are familiar with, and have access to, the MISRA C Guidelines. Finally, it is assumed that users of these guidelines have received appropriate training and are competent C language programmers.

3. Use of the MISRA AC AGC Guidelines

The MISRA AC AGC Guidelines do not introduce any new rules. Instead, they give guidance for the application of the MISRA C Guidelines in the context of automatic code generation. This is done in two ways:

- The MISRA C rules of are classified into primary categories, which give guidance on the role those rules play in the context of automatic code generation.
- The rules are further assigned secondary categories when additional considerations apply to those rules in the context of automatic code generation that go beyond the considerations in the context of manual coding.

The classification into primary and secondary categories is backed at all times by a rationale, which discusses the issues arising from automatic code generation. Thus all recommendations follow the following format:

> **Rule X.Y (Original MISRA-C:2004 Rule number)**
>
> **Text of original MISRA-C:2004 rule.**
>
> **Categories**
>
> List of primary and optional secondary categories into which the rule has been classified by the MISRA AC AGC Guidelines.
>
> **Rationale**
>
> Rationale for the classification and issues to keep in mind in the context of automatic code generation. This text also details any documentation or model level requirements implied by any secondary classifications.

3.1 Primary Categories

The rules of MISRA C are classified into the following distinct primary categories:

- **OBL** — Obligatory
  Adherence to rules which are classified OBL is required either:
  - to ensure that the generated C code is (as far as possible) strictly conforming to ISO/IEC 9899:1990, or
  - because the rule covers error-prone language features, whether related to automatic code generation or compiler implementation, or
  - because adherence to the rule is required to support critical automotive development processes.

- **REC** — Recomended
  Adherence to rules which are classified REC is recommended. The rule embodies good coding practice and adherence to the rule does not place an unreasonably high burden on the implementers of automatic code generator. It might be that the rule covers potentially error-prone or

highly unreadable language constructs, or because adherence to the rule is beneficial to automotive development processes.

- **READ** — Readability
  Adherence to rules which are classified READ is recommended, in order to increase code readability. If the code generated by an automatic code generator is not intended to be read, reviewed or modified by human programmers, adherence to rules categorized as READ is optional.

- **NA** — Not Applicable
  Adherence to rules which are classified NA is positively **not** recommended because adherence to the rule in the context of automatic code generation is either
  - not beneficial, or
  - does not make sense in this context, or
  - might potentially introduce additional issues if applied unwisely in this context.

The set of rules in MISRA C Guidelines are partitioned into these four non-overlapping categories, i.e. each rule is assigned to precisely one of these primary categories.

3.2 Secondary Categories

In addition to the primary categories, individual rules may also be tagged with one or more of the following secondary categories:

- **MOD** — Modelling level
  Rules which are classified as MOD concern design decisions which lie within the responsibility of the application designer and should thus be handled at the level of the modelling language. These rules are not intended to work around potential defects in automatic code generators. Whether such rules also apply at the C code level, and especially whether a code generator can or should enforce such rules at the code generation stage, depends on the specific rules concerned and is discussed in the rationale entry for each rule. In all cases specific recommendations should be put into place at the modelling stage to handle the issues raised by the given rule. It should be noted that that absence of a MOD classification for a given rule does not absolutely mean that no modelling guideline can be derived from it. It is possible that any rule, could give rise to a corresponding modelling guideline if the constructs considered in a rule are under the control of the user at the model level for a particular modelling language or/or automatic code generator. Given the wide variation in modelling languages and code generators it is not possible for the MISRA AC AGC to deal with all possibilities. Therefore, a MOD classification should be taken as an indication for the need for guidelines at the model level but its absence should not be taken as an indication that no guideline is required at the model level.

- **DOC** — Documentation
  Rules which are classified as DOC concern important information on the generated code that the user of the code will have to take into account when processing the generated code further. For example, the generated code may place requirements on any compiler used to compile it. In

order to ensure that the user has all the necessary information to handle the generated code, such as selecting a suitable compiler to compile the code, the relevant information shall be documented by the implementer of the code generator. For example, this might include any special treatment required by the generated code or any special-purpose coding strategies employed. The rationale of any rule that is classified as DOC will give more details on the information that is to be documented by the implementer of the code generator.

4. Compliance

Compliance with the MISRA AC AGC Guidelines can be claimed in two ways. Firstly, compliance can be claimed for the C code modules that are produced by an automatic generator in much the same manner as compliance with MISRA C might be claimed for manually generated code. Secondly, compliance can be claimed for an automatic code generation tool itself. This is a statement that all code generated by the code generator will comply at the level indicated provided that the code generator has been configured as specified by the code generator vendor in its compliance statement.

Note, for the avoidance of doubt, MISRA does not provide any compliance or conformance scheme for tools or code.

Compliance for both code and code generators can be claimed for the following levels:

- **OBL** — Obligatory
  Compliance with level OBL requires that all rules classified as OBL are adhered to unless reasonable justification is provided for non-adherence to individual rules.
- **REC** — Recommended
  Compliance with level REC requires that all rules classified as OBL or REC are adhered to unless reasonable justification is provided for non-adherence to individual rules.
- **READ** — Readability
  Compliance with level READ requires that all rules classified as OBL, REC or READ are adhered to, unless reasonable justification is provided for non-adherence to individual rules.

At all levels of compliance strict adherence to rules classified as **NA** is not recommended.

Statements of compliance shall include a compliance matrix and follow all other conditions laid out in Section 4.4 “Claiming compliance” of the MISRA C Guidelines.

In addition to these requirements the compliance statement shall include the following information:

- For each rule with a **DOC** classification, a statement identifying the location of the required documentation
- For each rule with a **MOD** classification, either
  - a statement identifying the location of the equivalent modelling guideline, or
  - if no such guideline is necessary because the intent of the guideline is inherently fulfilled in the modelling language of the code generator, a statement to this effect

5. Target optimizations and automatic code generators

One of the advantages that the use of automatic code generators can offer is the generation of target-specific code from a portable model. In this way, both portability and efficiency considerations can be realized at the same time, something that can be much more difficult to achieve in manually coded projects. It is recognized that target-specific code will generally deviate from one or more of the MISRA C rules, especially those provisions concerned with language extensions and implementation-defined behaviour.

MISRA C provides a mechanism for deviating from individual rules as described in Section 4.3.2 “Deviation procedure” of the MISRA Guidelines. In the context of automatic code generation, it is the automatic code generator implementer who has the responsibility for identifying which MISRA C rules are deviated from when generating code that is specific to a certain target. In addition, the implementer of the automatic code generator shall document prominently:

- The ways in which the generated code makes use of features not defined, or not fully defined, in ISO/IEC 9899:1990 as amended and corrected by ISO/IEC 9899/COR1:1995 [4], ISO/IEC 9899/AMD1:1995 [5], and ISO/IEC 9899/COR2:1996 [6].
- Which compiler toolchain, including specific version numbers, the target-specific extensions were designed to operate with and against which the code generation techniques were tested and validated

Additionally, all target-specific code generated shall contain a comment in each unit of source code generated clearly indicating that the code is target-specific. This comment shall include information regarding the target and compiler toolchain for which the code has been specifically generated.

It is the responsibility of the user of the automatic code generator to ensure that in each project the compiler toolchain employed to compile the generated code does indeed match the documented requirements.

While the generation of target-specific code can be useful in terms of code optimization, it is not without difficulties. Not only is the generated code non-portable but there is an increased risk of erroneous translation as the language extensions are likely to be less well understood and less well tested than the standard language features.

Therefore, an automatic code generator that has the facility to produce target-specific code shall also offer a mode that produces, to the greatest extent possible, fully conforming ISO C code as required by MISRA C Rule 1.1. Users of automatic code generation tools should always consider the use of this standard mode of the automatic code generator and only select target-specific generation if the situation warrants it.

5. Target optimizations and automatic code generators (continued)

Suitable measures to validate the proper working of the entire toolchain including the automatic code generator, compiler and linker should be always be taken, especially when features outside of ISO/IEC 9899:1990 are relied upon.

6. Rules

6.1 Environment

Rule 1.1

All code shall conform to ISO 9899:1990 "Programming languages – C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996

Categories

OBL, MOD

Rationale

The MOD classification applies to this rule only if the degree of conformity can be influenced by the user at the modelling level. For example ISO/IEC 9899:1990 specifies a number of translation limits, such as a minimum level of C compound statement nesting, that a conforming implementation must support. In order to ensure that the generated code does not exceed any such translation limits, it might be necessary to apply rules at the modelling level to restrict the complexity of the model.

It is recognized, that the goal of portability can be realized in the context of code generation by other means than strict adherence to the ISO standard, e.g. by generating specific code for specific target compilers.

Rule 1.2

No reliance shall be placed on undefined or unspecified behaviour

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 1.3

Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform

Categories

OBL, MOD, DOC

Rationale

The use of multiple programming languages is normally not under the control of the automatic code generator. In cases where multiple modelling languages and/or multiple automatic code generators are used within a project, interfacing between code generated by different automatic code generators remains a concern that is to be addressed. Therefore the nature of the C level interfaces used by an automatic code generator should be documented so that integration with other code can be readily and safely achieved.

Rule 1.4

The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers

Categories

OBL, DOC, MOD

Rationale

If length of generated identifiers is under control of the automatic code generator, then the automatic code generator implementer shall document the limits required of the compiler/linker so that the user can confirm that those limits are supported by the compiler/linker employed. Further, it is recommended that the automatic code generator offer a mode that restricts the length of identifiers to 31 characters. Where applicable this mode should also offer the ability to generate a table mapping model names to the shortened identifiers of the generated code.

If the length of generated identifiers lies under user control, for example if modelling level identifiers are used directly in the generated code, then this rule is applicable at the model level.

Rule 1.5

Floating-point implementations should comply with a defined floating-point standard

Categories

REC, DOC, MOD

Rationale

If the automatically generated code relies on a particular floating-point standard for correct operation then the automatic code generator implementer shall document this reliance prominently. If the reliance of the generated code on a given floating-point standard is under the control of the user then it is the user's responsibility to document the standard upon which the code relies.

In either case it falls to the user to establish that the compiler and processor employed follow the floating-point standard upon which the code relies. In all cases the consideration of IEEE 754 [7] and its successors as an appropriate standard is recommended.

6.2 Language Extensions

Rule 2.1

Assembly language shall be encapsulated and isolated

Categories

OBL, DOC, MOD

Rationale

The automatic code generator implementer shall document any use of assembly language constructs if their use is under the control of the automatic code generator. The MOD classification applies if assembly language constructs can be injected at the model level and a corresponding rule shall therefore be instituted at the model level. In cases where assembly language constructs cannot be encapsulated as a macro, for example targets for which assembly language code is introduced with `#pragma` directives, non-adherence to this rule is acceptable for automatically generated code.

Rule 2.2

Source code shall only use `/*` … `*/` style comments

Categories

OBL

Rationale

The same considerations apply as for manually created code. The ISO/IEC 9899:1990 C standard applies and there shall be no dependence of generated code on C99.

Rule 2.3

The character sequence `/*` shall not be used within a comment

Categories

OBL

Rationale

The same considerations apply as for manually created code. The automatic code generator implementer should ensure that comments entered at the model level are not able to violate this rule.

Rule 2.4

Sections of code should not be "commented out"

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.3 Documentation

Rule 3.1

All usage of implementation-defined behaviour shall be documented

Categories

OBL, DOC

Rationale

The automatic code generator implementer shall document the nature of any reliance on implementation-defined behaviour.

Rule 3.2

The character set and the corresponding encoding shall be documented

Categories

OBL, DOC, MOD

Rationale

The automatic code generator implementer shall document any reliance on the character set if this is under the control of the automatic code generator. Where the character set is under the control of the user then the MOD classification applies.

Rule 3.3

The implementation of integer division in the chosen compiler should be determined, documented and taken into account

Categories

OBL, DOC, MOD

Rationale

The automatic code generator implementer shall document any reliance on the specific implementation of integer division. If the user has control over the behaviour of the integer division operation then the MOD classification applies.

Rule 3.4

All uses of the *#pragma* directive shall be documented and explained

Categories

OBL, DOC, MOD

Rationale

The automatic code generator implementer shall document any reliance on pragmas. If the choice of pragmas used is under the control of the user then responsibility for the documentation falls to the user and the MOD classification appliers.

Rule 3.5

If it is being relied upon, the implementation defined behaviour and packing of bitfields shall be documented

Categories

OBL, DOC

Rationale

The same considerations apply as for manually created code.

Rule 3.6

All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation

Categories

OBL, DOC, MOD

Rationale

The same considerations apply as for manually created code. The MOD classification applies to user-supplied libraries. It is the responsibility of the automatic code generator implementer to document the compliance and validation status of any libraries supplied with the automatic code generator.

6.4 Character sets

Rule 4.1

Only those escape sequences that are defined in the ISO C standard shall be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 4.2

Trigraphs shall not be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.5 Identifiers

Rule 5.1

Identifiers (internal and external) shall not rely on the significance of more than 31 characters

Categories

OBL, DOC, MOD

Rationale

This is a requirement on the automatic code generator, unless the length of identifiers is under user-control, in which case the MOD classification applies. In either case the length of generated identifiers shall be documented in automatic code generator documentation.

Rule 5.2

Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier

Categories

REC

Rationale

This rule is required for manually created code because it can reduce scope for confusion when identifiers in inner scopes hide those in outer scopes. For automatically generated code, there is less scope for confusion because the automatic code generator should keep track of scopes. However, this is an area in which compilers have historically been shown to have bugs. Therefore, it might seem reasonable to assume that similar bugs might exist in automatic code generators. For this reason, this rule is given the REC classification.

Rule 5.3

A *typedef* name shall be a unique identifier

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 5.4

A tag name shall be a unique identifier

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 5.5

No object or function identifier with static storage duration should be reused

Categories

READ, MOD

Rationale

The same considerations apply as for manually created code. Insofar as choice of identifier names lies under the control of the user, the MOD classification applies and equivalent model level rules should be put in place.

Rule 5.6

No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names

Categories

READ, MOD

Rationale

The same considerations apply as for manually created code. Insofar as choice of identifier names lies under the control of the user, the MOD classification applies and equivalent model level rules should be put in place.

Rule 5.7

No identifier name should be reused

Categories

READ, MOD

Rationale

The same considerations apply as for manually created code. Insofar as choice of identifier names lies under the control of the user, the MOD classification applies and equivalent model level rules should be put in place.

6.6 Types

Rule 6.1

The plain *char* type shall be used only for the storage and use of character values

Categories

OBL, MOD

Rationale

The same considerations apply as for manually created code. Insofar as choice of types lies under the control of the user, the MOD classification applies and equivalent model level rules should be put in place.

Rule 6.2

***signed* and *unsigned char* type shall be used only for the storage and use of numeric values**

Categories

OBL, MOD

Rationale

The same considerations apply as for manually created code. Insofar as choice of identifier names lies under the control of the user, the MOD classification applies and equivalent model level rules should be put in place.

Rule 6.3

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

Categories

OBL, MOD

Rationale

The definition of the *typedefs* used by the automatic code generator is usually under the control of the user in order to ease porting or adaptation to embedded targets. Hence appropriate rules should be introduced at the model level.

Rule 6.4

Bit fields shall only be defined to be of type *unsigned int* or *signed int*

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 6.5

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.7 Constants

Rule 7.1

Octal constants (other than zero) and octal escape sequences shall not be used

Categories

REC

Rationale

The same considerations apply as for manually created code.

6.8 Declarations and definitions

Rule 8.1

Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call

Categories

OBL

Rationale

The same considerations apply as for manually created code. This is especially relevant for integration with manually created code but it does also provide the compiler with additional opportunities to check automatically generated code as well.

Rule 8.2

Whenever an object or function is declared or defined, its type shall be explicitly stated

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.3

For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.4

If objects or functions are declared more than once their types shall be compatible

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.5

There shall be no definitions of objects or functions in a header file

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.6

Functions shall be declared at file scope

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.7

Objects shall be defined at block scope if they are only accessed from within a single function

Categories

OBL

Rationale

An automatic code generator should be able to keep track of scopes and can therefore handle a more variables with wide scope than would be considered manageable for manually created code. However, the overriding considerations of readability and analysability suggest that this rule should be classified as OBL.

Rule 8.8

An external object or function shall be declared in one and only one file

Categories

OBL

Rationale

This is especially important in order to prevent problems during integration with manually created code and other standard libraries.

Rule 8.9

An identifier with external linkage shall have exactly one external definition

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.10

All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required

Categories

OBL

Rationale

This is especially important in order to prevent unnecessary problems during integration with manually created code and other standard libraries.

Rule 8.11

The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 8.12

When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.9 Initialisation

Rule 9.1

All automatic variables shall have been assigned a value before being used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 9.2

Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures

Categories

READ

Rationale

Using braces to show structure during the initialization of arrays and structure improves readability. However, the automatic code generator can be relied on to ensure that the number of initializers matches the number of elements in the object so there is no need to elevate this rule to a REC classification.

Rule 9.3

In an enumerator list, the "=" construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.10 Arithmetic type conversions

Rule 10.1

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

Categories

REC, DOC

Rationale

This rule and the concept of an underlying type in MISRA C is intended to prevent accidental or erroneous casts that can lead to undefined behaviour or loss of data. However, the concept of an underlying type as employed in this rule is not the only safe casting strategy. Therefore this rule itself is not categorized as OBL. Nonetheless, the automatic code generator implementer shall either adhere to this rule or shall employ a different safe strategy for casting and that strategy shall be documented.

Rule 10.2

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

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 10.3

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

Categories

REC, DOC

Rationale

This rule and the concept of an underlying type in MISRA C is intended to prevent accidental or erroneous casts that can lead to undefined behaviour or loss of data. However, the concept of an underlying type as employed in this rule is not the only safe casting strategy. Therefore this rule itself is not categorized as OBL. Nonetheless, the automatic code generator implementer shall either adhere to this rule or shall employ a different safe strategy for casting and that strategy shall be documented.

Rule 10.4

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

Categories

REC

Rationale

The same considerations apply as for manually created code.

Rule 10.5

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

Categories

REC, DOC

Rationale

This rule and the concept of an underlying type in MISRA C is intended to prevent accidental or erroneous casts that can lead to undefined behaviour or loss of data. However, the concept of an underlying type as employed in this rule is not the only safe casting strategy. Therefore this rule itself is not categorized as OBL. Nonetheless, the automatic code generator implementer shall either adhere to this rule or shall employ a different safe strategy for casting and that strategy shall be documented.

Rule 10.6

A "U" suffix shall be applied to all constants of unsigned type

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.11 Pointer type conversions

Rule 11.1

Conversions shall not be performed between a pointer to a function and any type other than an integral type

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 11.2

Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 11.3

A cast should not be performed between a pointer type and an integral type

Categories

OBL

Rationale

Casting a pointer to an integer results in implementation-defined behaviour. Where the value will not fit in the integer type, there is a good chance that the defined behaviour is in fact undefined.

Rule 11.4

A cast should not be performed between a pointer to object type and a different pointer to object type

Categories

OBL

Rationale

Casting between pointers to different object types can lead to undefined behaviour as the alignment requirements for the objects aren't readily available to the automatic code generator when generating generic code, i.e. code that is not target-specific.

Rule 11.5

A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.12 Expressions

Rule 12.1

Limited dependence should be placed on C's operator precedence rules in expressions

Categories

REC

Rationale

Fully parenthesized expressions are no harder for the automatic code generator to generate and can improve readability.

Rule 12.2

The value of an expression shall be the same under any order of evaluation that the standard permits

Categories

OBL, MOD

Rationale

The same considerations apply as for manually created code. In some cases the MOD classification applies because it is possible, if not always easy, for the user to create expressions with embedded side-effects. For example, if might be possible in a modelling language to create a block for which all the operands refer to volatile objects. In such a case, the value of the generated expression would depend on the order of evaluation of those objects.

Rule 12.3

The *sizeof* operator shall not be used on expressions that contain side effects

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 12.4

The right hand operand of a logical `&&` or `||` operator shall not contain side effects

Categories

OBL, MOD

Rationale

The use of side-effects in short-circuiting logical operators is highly unreadable and error-prone. It is known to be an area in which compiler have contained errors. The MOD classification applies only if this resides under the control of the user.

Rule 12.5

The operands of a logical `&&` or `||` shall be primary-expressions

Categories

REC

Rationale

This rule is a special case of rule 12.1 and the same rationale applies, namely that fully parenthesized expressions are no harder for the automatic code generator to generate and can improve readability.

Rule 12.6

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

Categories

REC

Rationale

The same considerations apply as for manually created code. Usage of expressions that are effectively Boolean should also be allowed for the first operand of the `?` operator if usage of that operator is allowed.

Rule 12.7

Bitwise operators shall not be applied to operands whose underlying type is signed

Categories

OBL, MOD

Rationale

The same considerations apply as for manually created code. If the generation of shift operations lies under user control, then this rule is applicable at the model level.

Rule 12.8

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

Categories

OBL, MOD

Rationale

The same considerations apply as for manually created code. If control of the shift operator's right hand operand lies under user control then this rule is applicable at the model level.

Rule 12.9

The unary minus operator shall not be applied to an expression whose underlying type is unsigned

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 12.10

The comma operator shall not be used

Categories

REC

Rationale

Use of the comma operator can lead to highly unreadable code and compilers have been known to contain bugs in its implementation. It is not unreasonable to expect automatic code generator implementers to use alternative strategies to achieve the effect of the comma operator. Given the power of contemporary optimizing compilers, it is unlikely that any reasonable alternative strategy will give rise to significantly less efficient code.

Rule 12.11

Evaluation of constant unsigned integer expressions should not lead to wrap-around

Categories

OBL, MOD

Rationale

This is OBL since compilers have been known to contain bugs in this area. There is less need for automatic code generators to use constant expressions that manually coders as the code generator can evaluate many constant expressions itself. The MOD classification applies only if generation of constant unsigned integer expressions resides under user-control.

Rule 12.12

The underlying bit representations of floating-point values shall not be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 12.13

The increment (++) and decrement (- -) operators should not be mixed with other operators in an expression

Categories

READ

Rationale

Provided that the ISO C rules regarding sequence points and side-effect visibility are observed, this is only a readability issue. Adherence to Rule 12.2 will ensure that this is the case since it requires that the expression's value be the same under all permitted orders of evaluation.

6.13 Control statement expressions

Rule 13.1

Assignment operators shall not be used in expressions that yield a Boolean value

Categories

REC

Rationale

Compliance with this rule greatly improves both readability and robustness of code. It is reasonable to expect automatic code generator implementers to use alternative strategies to achieve the desired effect.

Rule 13.2

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

Categories

REC

Rationale

Compliance with this rule greatly improves both readability and robustness of code. It is reasonable to expect automatic code generator implementers to use alternative strategies to achieve the desired effect.

Rule 13.3

Floating-point expressions shall not be tested for equality or inequality

Categories

NA, MOD

Rationale

The specification of equality tests remains under the control of the user and it is therefore the responsibility of the user to employ suitable predicates and application-specific values of epsilon. The automatic code generator is not capable of either supplying suitable values or overriding any values supplied by the user. If it were to insert code to avoid any equality or inequality tests that were suggested directly by the model then the generated code would comply with this rule. However, the correctness of the generated code would be questionable, given the automatic code generator's inability to know how the test should be constructed.

For technical tests inserted by the automatic code generator, e.g. to guard against the generation of exceptions on division by zero, exact equality or inequality tests are often the only proper choice in the absence of a suitable user-supplied value for epsilon. Checking of automatically generated code would therefore often reveal violations of this rule despite the offending constructs being both intentional and desirable.

Therefore, rather than relying on checking the C code, it is preferable to put rules into place at the model level and to check those rules at that level. The automatic code generator should **not** attempt to satisfy this rule when translating modelling constructs and it **should** insert appropriate equality or inequality tests when necessary to protect against exceptions.

Rule 13.4

The controlling expression of a *for* statement shall not contain any objects of floating type

Categories

REC, MOD

Rationale

Correct usage of floating point numbers is possible by automatic code generators, e.g. for mathematical iteration algorithms. However for readability and ease of checking purposes adherence to this rule is recommended. It should be noted however that the problems alluded to in the rule have to be dealt with, regardless of the looping construct chosen, i.e. *do* and *while* loops should be checked for improper use of floating point numbers as well. Corresponding rules should be put into place at the model level if the controlling expression of the *for* statement is under the control of the user.

Rule 13.5

The three expressions of a *for* statement shall be concerned only with loop control

Categories

REC, MOD

Rationale

The same considerations apply as for manually created code.

Rule 13.6

Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop

Categories

REC, MOD

Rationale

The same considerations apply as for manually created code.

Rule 13.7

Boolean operations whose results are invariant shall not be permitted

Categories

OBL, MOD

Rationale

The same considerations apply as for manually created code. However exceptions can be made for the generation of non-terminating loops, e.g. for the main control loop if generated by the automatic code generator. It should also be noted that expressions containing volatile variables are not invariant, even if they often look like they might be. This is especially true in the presence of run-time calibration of parameters in running systems as is typically performed in standard automotive development processes. Since the code generator usually cannot decide for itself whether certain expressions are invariant, equivalent model level rules should be put into place.

6.14 Control flow

Rule 14.1

There shall be no unreachable code

Categories

OBL, MOD

Rationale

The presence of unreachable code is normally indicative of automatic code generator errors. It can also lead to excessive memory consumption where the unreachable code is not removed by the compiler, a process which itself is error-prone. Where unreachability can be determined at the model level, the MOD classification applies. Determination of unreachable code should however take into account the effect of volatile variables, which can (and will) change values at run-time even if this is not apparent from the source code (see also the rationale for rule 13.7).

Rule 14.2

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

Categories

OBL, MOD

Rationale

Violation of this rule is usually indicative of automatic code generator errors. It can also lead to excessive memory consumption where the unreachable code is not removed by the compiler, a process which itself is error-prone. Where generation of such code is determined at the model level the MOD classification applies.

Rule 14.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character

Categories

OBL

Rationale

Violation of this rule is usually indicative of automatic code generator errors. It is reasonable to expect automatic code generator implementers to use alternative strategies to achieve the desired effect.

Rule 14.4

The *goto* statement shall not be used

Categories

REC

Rationale

Avoidance of the *goto* statement is generally recommended. It can however have a place in automatically generated code for state-machines, where *goto* or *switch* statements are idiomatic ways of implementing state transitions. Even this limited usage would have a negative impact on readability and analysability and for this reason this rule is classified as REC.

Rule 14.5

The *continue* statement shall not be used

Categories

REC

Rationale

The classification as REC is warranted by the impact on readability and analysability.

Rule 14.6

For any iteration statement there shall be at most one *break* statement used for loop termination

Categories

REC, MOD

Rationale

The classification as REC is warranted by the impact on readability and analysability. Where the generation of iteration exits is determined at the model level, this rule applies at the model level.

Rule 14.7

A function shall have a single point of exit at the end of the function

Categories

OBL, MOD

Rationale

Exceptions might be granted for idiomatically generated code for state-machines or similar regularized control structures (e.g. lookup-tables), especially if those are highly time-critical. Those exceptions would have to be argued against the IEC 61508 HR recommendation, where applicable. Where the generation of function exits is determined at the model level, this rule applies at the model level.

Rule 14.8

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

Categories

REC

Rationale

While this is mostly a readability issue, the robustness of code in the presence of erroneous or ill-advised macros is also a concern. Since it is reasonable to expect automatic code generator implementers to use alternative strategies to achieve the desired effect this rule is classified as REC.

Rule 14.9

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

Categories

REC

Rationale

While this is mostly a readability issue, the robustness of code in the presence of erroneous or ill-advised macros is also a concern. Since it is reasonable to expect automatic code generator implementers to use alternative strategies to achieve the desired effect this rule is classified as REC.

Rule 14.10

All *if … else if* constructs shall be terminated with an *else* clause

Categories

READ, MOD

Rationale

This is a readability issue for the generated code. However for all instances where the presence of useful *else* clauses resides under the control of the user, corresponding model level rules should be introduced.

6.15 Switch statements

Rule 15.1

A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement

Categories

OBL

Rationale

The generation of unstructured switch statements greatly detracts from readability. More importantly using unstructured switch statements is placing reliance on questionable behaviour in a poorly understood area of the C language with the result that it is likely to expose errors or undefined behaviour in compilers.

Rule 15.2

An unconditional *break* statement shall terminate every non-empty switch clause

Categories

REC

Rationale

It is recommended that automatic code generators comply with this MISRA C rule to the greatest extent possible. However, it is recognized that an automatic code generator can implement certain modelling language constructs, such as state machines, most efficiently if this rule is violated. Therefore, if an automatic code generator generates code that deliberately omits a *break* statement in order to have a *case* clause drop through to the next *case* clause, the automatic code generator must insert a comment into the generated C that explains why the *break* statement is absent.

Rule 15.3

The final clause of a switch statement shall be the default clause

Categories

NA, MOD

Rationale

This MISRA C rule requires that the final clause of a *switch* statement must be a *default* clause. The rule is made for the purpose of encouraging defensive programming. Since defensive programming applies equally to modelling languages, a corresponding model level rule should be put in place to cover the situation where generation of a *default* clause resides under the control of the automatic code generator user.

In the presence of models which do not specify default behaviour, the automatic code generator is unable to determine what to put into the *default* clause. The automatic code generator could generate an empty *default* clause to satisfy the MISRA C rule but this does not provide any increase in defensive programming. It is much better for the automatic code generator **not** to generate the *default* clause. In this situation, a MISRA C compliance check will reveal the lack of *default* clause and this can be traced back to the model level.

If an automatic code generator determines that it can optimize away a *default* clause that is suggested by the model, for example if every valid value for the *switch* expression is covered by a *case* clause, then it may only do so if it inserts a comment into the generated C that explains why the *default* clause is absent. This comment can be reviewed and accepted as part of the MISRA C compliance argument.

Rule 15.4

A *switch* expression shall not represent a value that is effectively Boolean

Categories

REC

Rationale

Adherence to this rule can prevent readability issues, and it is thus recommended. It is not too difficult for an automatic code generator implementers to use alternative strategies to achieve the desired effect.

Rule 15.5

Every *switch* statement shall have at least one *case* clause

Categories

REC, MOD

Rationale

Adherence to this rule can prevent readability issues, and it is thus recommended. It is not too difficult for automatic code generator implementers to use alternative strategies to achieve the desired effect.

If the number of case clauses within a *switch* statement lies under user control, then this rule is applicable at the model level.

6.16 Functions

Rule 16.1

Functions shall not be defined with a variable number of arguments

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 16.2

Functions shall not call themselves, either directly or indirectly

Categories

OBL, MOD

Rationale

Since the use of recursion might be prompted by equivalent constructs in the model, corresponding rules at the model level should be put into place.

Rule 16.3

Identifiers shall be given for all of the parameters in a function prototype declaration

Categories

OBL

Rationale

This is especially important for integration of automatically generated code with other code. It can also improve readability and the ability to debug code. It does not require significant effort for automatic code generator implementers to satisfy this rule.

Rule 16.4

The identifiers used in the declaration and definition of a function shall be identical

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 16.5

Functions with no parameters shall be declared with parameter type void

Categories

REC

Rationale

The same considerations apply as for manually created code.

Rule 16.6

The number of arguments passed to a function shall match the number of parameters

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 16.7

A pointer parameter in a function prototype should be declared as pointer to *const* if the pointer is not used to modify the addressed object

Categories

REC, MOD

Rationale

This is highly recommended and it does not require significant effort for automatic code generator implementers to satisfy this rule where the information is available. The benefits in terms of reliability, avoidance of integration issues and readability are significant. In cases where the necessary information is to be specified at the model level, e.g. for external interfaces, a corresponding rule should be instituted at the model level.

Rule 16.8

All exit paths from a function with non-*void* return type shall have an explicit return statement with an expression

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 16.9

A function identifier shall only be used with either a preceding `&`, or with a parenthesised parameter list, which may be empty

Categories

OBL

Rationale

Compliance with this rule enhances readability and it does not require significant effort for automatic code generator implementers to satisfy it.

Rule 16.10

If a function returns error information, then that error information shall be tested

Categories

OBL, MOD

Rationale

The overall error-handling strategy, especially with regard to third-party code, employed by a project must at all times reside under the control of the user. Only the user can ascertain that the error-handling strategy is effective, coherent and aligned with the overall safety considerations. Allowing the automatic code generator to decide how to check for errors and handle them is dangerous since the code generator can have no knowledge of the application. Therefore adequate rules concerning error-handling should be put into place at the model level, preferably in the form of project-specific rules that are in accordance with the overall safety plan. The automatic code generator shall only be responsible for the checking and handling of errors in code completely under its control and for faithful implementation of any checking present at the model level.

6.17 Pointers and arrays

Rule 17.1

Pointer arithmetic shall only be applied to pointers that address an array or array element

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 17.2

Pointer subtraction shall only be applied to pointers that address elements of the same array

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 17.3

>, >=, <, <= shall not be applied to pointer types except where they point to the same array

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 17.4

Array indexing shall be the only allowed form of pointer arithmetic

Categories

REC

Rationale

Adherence to this rule is highly recommended. However pointer increment might be faster for certain compilers and this might be an important consideration in especially performance-critical code such as table-lookup.

Rule 17.5

The declaration of objects should contain no more than 2 levels of pointer indirection

Categories

READ

Rationale

As explained in the supporting text in the MISRA C Guidelines this is a readability and analysability issue.

Rule 17.6

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

Categories

OBL

Rationale

There is no issue with assigning the address of an object after it has ceased to exist. A problem only occurs if that pointer is subsequently dereferenced at which instant the behaviour is undefined. Even if this event can be prevented, such constructs are highly unreadable and error-prone and should be avoided in all cases, especially since there is usually little reason to employ them.

6.18 Structures and unions

Rule 18.1

All structure and union types shall be complete at the end of a translation unit

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 18.2

An object shall not be assigned to an overlapping object

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 18.3

An area of memory shall not be reused for unrelated purposes

Categories

REC, MOD

Rationale

An automatic code generator is likely to be far more effective in ensuring that no overlap in usage occurs when memory is reused. However a compiler is already very capable at performing optimization of storage for local variables using techniques such as live range analysis. The automatic code generator should not attempt to repeat any such optimizations as it is a complex process and risks introducing errors. It also has a significant impact on readability and the ability to perform static analysis. This rule also applies at the model level since users are very error-prone in performing and maintaining a proper lifetime analysis.

Rule 18.4

Unions shall not be used

Categories

OBL

Rationale

Safe usage of unions is made difficult for automatic code generators because of the number of instances of implementation-defined behaviour. Unions should therefore generally be avoided.

It is recognized that efficiency considerations apply and that deviation from this rule may be acceptable in certain circumstances, especially when generating target-specific code. In such circumstances, the implementation-defined behaviour is known to the automatic code generator and can be taken into account. The considerations of Section 5 "Target optimizations and automatic code generators" should be taken into account in these circumstances.

6.19 Preprocessing directives

Rule 19.1

***#include* statements in a file should only be preceded by other preprocessor directives or comments**

Categories

REC

Rationale

Compliance with this rule enhances readability and robustness especially when the included headers are not themselves automatically generated or controlled by the automatic code generator. This rule does not require significant effort for automatic code generator implementers to satisfy it.

Rule 19.2

Non-standard characters should not occur in header file names in *#include* directives

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.3

The *#include* directive shall be followed by either a <filename> or "filename" sequence

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.4

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

Categories

REC

Rationale

This rule should generally be adhered because it improves readability even though an automatic code generator is more likely to be able to ensure the correct usage of macros defined in other ways. An automatic code generator can usually employ other strategies for performing textual substitutions in the code it generates.

Rule 19.5

Macros shall not be #*define*'d or #*undef*'d within a block

Categories

READ

Rationale

This is in essence a readability issue.

Rule 19.6

#*undef* shall not be used

Categories

READ

Rationale

This is in essence a readability issue.

Rule 19.7

A function should be used in preference to a function-like macro

Categories

REC

Rationale

The use of function-like macros is sometimes acceptable on the grounds of efficiency, especially for an automatic code generator as it is likely to be able to ensure correct usage of macros. However an automatic code generator can also use an alternative strategy of directly inlining functions thereby avoiding the issues under consideration, but at the expense of reduced readability.

Rule 19.8

A function-like macro shall not be invoked without all of its arguments

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.9

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.10

In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

Categories

OBL

Rationale

While an automatic code generator is likely to be able to guarantee proper expansion of macros in other ways adherence to this rule results in a significant increase in robustness and readability.

Rule 19.11

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.12

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.13

The # and ## operators should not be used

Categories

REC

Rationale

This rule is not OBL because the MISRA C rule is advisory and compiler implementation variability is sufficiently small to permit safe usage of # and ##. However, automatic code generators should be conservative in their usage of # and ##, especially as they can achieve the effects of # and ## by themselves much more safely.

Rule 19.14

The *defined* preprocessor operator shall only be used in one of the two standard forms

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.15

Precautions shall be taken in order to prevent the contents of a header file being included twice

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.16

Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 19.17

All *#else*, *#elif* and *#endif* preprocessor directives shall reside in the same file as the *#if* or *#ifdef* directive to which they are related

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.20 Standard libraries

Rule 20.1

Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined

Categories

OBL, MOD

Rationale

Only the free-standing environment as defined by ISO/IEC 9899:1990 may be assumed. Ideally, no dependence should be placed on any compiler-provided libraries. Where the user can control identifier usage at the model level, corresponding rules at the model level should be put into place.

Rule 20.2

The names of standard library macros, objects and functions shall not be reused

Categories

OBL, MOD

Rationale

Where the user can control identifier usage at the model level, corresponding rules at the model level should be put into place.

Rule 20.3

The validity of values passed to library functions shall be checked

Categories

OBL, MOD

Rationale

Where the values passed to library functions reside under the control of the user, model level rules should be put into place. In all other cases this is a conformance issue for the automatic code generator.

Rule 20.4

Dynamic heap memory allocation shall not be used

Categories

OBL, MOD

Rationale

Where the decision to use dynamic heap memory allocation resides under control of the user, corresponding model-level rules should be put into place.

Rule 20.5

The error indicator *errno* shall not be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.6

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.7

The *setjmp* macro and the *longjmp* function shall not be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.8

The signal handling facilities of <*signal.h*> shall not be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.9

The input/output library <*stdio.h*> shall not be used in production code

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.10

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.11

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

Categories

OBL

Rationale

The same considerations apply as for manually created code.

Rule 20.12

The time handling functions of library <*time.h*> shall not be used

Categories

OBL

Rationale

The same considerations apply as for manually created code.

6.21 Run-time failures

Rule 21.1

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

Categories

OBL, MOD, DOC

Rationale

The overall error-handling strategy lies in the domain of the user. Therefore corresponding rules shall be put into place at the model level. As far as code completely under the control of the automatic code generator is concerned, the automatic code generator shall ensure that no runtime errors can occur through an adequate code generation strategy. If that is infeasible, the employed error-handling strategy of the automatic code generator has to be documented, and taken into account by the user in the project. Automatically generated code that interfaces with hand-written code requires special scrutiny, and might have to employ documented error-handling code for robustness.

7. References

[1] *Guidelines for the use of the C language in critical systems*, ISBN 0-9524156-2-3, MIRA, October 2004

[2] *Introduction to the MISRA guidelines for the use of automatic code generation in automotive systems*, ISBN 978-1-906400-00-2, MIRA, November 2007

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

[4] *ISO/IEC 9899:COR1:1995, Technical Corrigendum 1*, International Organization for Standardization, 1995

[5] *ISO/IEC 9899:AMD1:1995, Amendment 1*, International Organization for Standardization, 1995

[6] *ISO/IEC 9899:COR2:1996, Technical Corrigendum 2*, International Organization for Standardization, 1996

[7] *ANSI/IEEE Std 754, IEEE Standard for Binary Floating-Point Arithmetic*, American National Standards Institute, 1985

Appendix A: Summary of Rule Categories

Rule	Primary				Secondary	
	OBL	REC	READ	NA	MOD	DOC
1.1	●				●	
1.2	●					
1.3	●				●	●
1.4	●				●	●
1.5		●			●	●
2.1	●				●	●
2.2	●					
2.3	●					
2.4	●					
3.1	●					●
3.2	●				●	●
3.3	●				●	●
3.4	●				●	●
3.5	●					●
3.6	●				●	●
4.1	●					
4.2	●					
5.1	●				●	●
5.2		●				
5.3	●					
5.4	●					
5.5			●		●	
5.6			●		●	
5.7			●		●	
6.1	●				●	
6.2	●				●	
6.3	●				●	
6.4	●					
6.5	●					
7.1		●				
8.1	●					
8.2	●					
8.3	●					
8.4	●					
8.5	●					
8.6	●					
8.7	●					
8.8	●					
8.9	●					
8.10	●					
8.11	●					

Rule	Primary				Secondary	
	OBL	REC	READ	NA	MOD	DOC
8.12	●					
9.1	●					
9.2			●			
9.3	●					
10.1		●				●
10.2	●					
10.3		●				●
10.4		●				
10.5		●				●
10.6	●					
11.1	●					
11.2	●					
11.3	●					
11.4	●					
11.5	●					
12.1		●				
12.2	●				●	
12.3	●					
12.4	●				●	
12.5		●				
12.6		●				
12.7	●				●	
12.8	●				●	
12.9	●					
12.10		●				
12.11	●				●	
12.12	●					
12.13			●			
13.1		●				
13.2		●				
13.3				●	●	
13.4		●			●	
13.5		●			●	
13.6		●			●	
13.7	●				●	
14.1	●				●	
14.2	●				●	
14.3	●					
14.4		●				
14.5		●				
14.6		●			●	
14.7	●				●	
14.8		●				

Rule	Primary				Secondary	
	OBL	REC	READ	NA	MOD	DOC
14.9		●				
14.10			●		●	
15.1	●					
15.2		●				
15.3				●	●	
15.4		●				
15.5		●			●	
16.1	●					
16.2	●				●	
16.3	●					
16.4	●					
16.5		●				
16.6	●					
16.7		●			●	
16.8	●					
16.9	●					
16.10	●				●	
17.1	●					
17.2	●					
17.3	●					
17.4		●				
17.5			●			
17.6	●					
18.1	●					
18.2	●					
18.3		●			●	
18.4	●					
19.1		●				
19.2	●					
19.3	●					
19.4		●				
19.5			●			
19.6			●			
19.7		●				
19.8	●					
19.9	●					
19.10	●					
19.11	●					
19.12	●					
19.13		●				
19.14	●					
19.15	●					
19.16	●					

Rule	Primary				Secondary	
	OBL	REC	READ	NA	MOD	DOC
19.17	●					
20.1	●				●	
20.2	●				●	
20.3	●				●	
20.4	●				●	
20.5	●					
20.6	●					
20.7	●					
20.8	●					
20.9	●					
20.10	●					
20.11	●					
20.12	●					
21.1	●				●	●

Appendix B

Appendix B: Breakdown of Rules by Primary Category

The number of rules in each of the primary categories is:

OBL	98
REC	32
READ	9
NA	2

These rules belonging to each category are listed in the table below.

Category	Rules				
OBL	1.1	6.2	11.1	16.2	19.12
	1.2	6.3	11.2	16.3	19.14
	1.3	6.4	11.3	16.4	19.15
	1.4	6.5	11.4	16.6	19.16
	2.1	8.1	11.5	16.8	19.17
	2.2	8.2	12.2	16.9	20.1
	2.3	8.3	12.3	16.10	20.2
	2.4	8.4	12.4	17.1	20.3
	3.1	8.5	12.7	17.2	20.4
	3.2	8.6	12.8	17.3	20.5
	3.3	8.7	12.9	17.6	20.6
	3.4	8.8	12.11	18.1	20.7
	3.5	8.9	12.12	18.2	20.8
	3.6	8.10	13.7	18.4	20.9
	4.1	8.11	14.1	19.2	20.10
	4.2	8.12	14.2	19.3	20.11
	5.1	9.1	14.3	19.8	20.12
	5.3	9.3	14.7	19.9	21.1
	5.4	10.2	15.1	19.10	
	6.1	10.6	16.1	19.11	
REC	1.5	12.5	13.6	15.4	19.4
	5.2	12.6	14.4	15.5	19.7
	7.1	12.10	14.5	16.5	19.13
	10.1	13.1	14.6	16.7	
	10.3	13.2	14.8	17.4	
	10.4	13.4	14.9	18.3	
	10.5	13.5	15.2	19.1	
READ	5.5	5.7	12.13	17.5	19.6
	5.6	9.2	14.10	19.5	
NA	13.3	15.3			