MISRA C:2012 Permits

Deviation permits for MISRA Compliance

First Edition, March 2021

First published March 2021 by HORIBA MIRA Limited
Watling Street
Nuneaton
Warwickshire
CV10 0TU
UK

www.misra.org.uk

ISBN 978-1-906400-27-9

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

MISRA C:2012 Permits

Deviation permits for MISRA Compliance

First Edition, March 2021

MISRA Mission Statement

We provide world-leading, best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

MISRA is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety- and security-related 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.

Revision history

The number of *deviation permits* within this document is expected to grow and it is possible that existing *deviation permits* may be revised. The following table is a record of these changes.

Date	Change
March 2021	Initial release — Equivalents to relevant 2004 permits — Permits for use with automatically generated code

Contents

1 Introduction 1

1.1 Scope 1

2 Introduction to the permits 2

3 Directive permits 3

4 Rule permits 4

Permit / MISRA / C:2012 / R-2.1.A.1 4
Permit / MISRA / C:2012 / R-2.1.B.1 5
Permit / MISRA / C:2012 / R-5.3.A.1 6
Permit / MISRA / C:2012 / R-6.1.A.1 7
Permit / MISRA / C:2012 / R-7.1.A.1 8
Permit / MISRA / C:2012 / R-7.2.A.1 8
Permit / MISRA / C:2012 / R-7.3.A.1 9
Permit / MISRA / C:2012 / R-8.4.A.1 9
Permit / MISRA / C:2012 / R-8.5.A.1 9
Permit / MISRA / C:2012 / R-8.14.A.1 10
Permit / MISRA / C:2012 / R-9.2.A.1 10
Permit / MISRA / C:2012 / R-9.3.A.1 11
Permit / MISRA / C:2012 / R-9.5.A.1 11
Permit / MISRA / C:2012 / R-10.1.A.1 11
Permit / MISRA / C:2012 / R-10.3.A.1 12
Permit / MISRA / C:2012 / R-10.4.A.1 12
Permit / MISRA / C:2012 / R-10.6.A.1 13
Permit / MISRA / C:2012 / R-10.7.A.1 13
Permit / MISRA / C:2012 / R-10.8.A.1 14
Permit / MISRA / C:2012 / R-11.9.A.1 14
Permit / MISRA / C:2012 / R-14.2.A.1 15
Permit / MISRA / C:2012 / R-14.3.A.1 15
Permit / MISRA / C:2012 / R-14.3.B.1 16
Permit / MISRA / C:2012 / R-14.3.C.1 17
Permit / MISRA / C:2012 / R-15.7.A.1 18
Permit / MISRA / C:2012 / R-16.1.A.1 19
Permit / MISRA / C:2012 / R-16.3.A.1 19
Permit / MISRA / C:2012 / R-16.4.A.1 20
Permit / MISRA / C:2012 / R-16.5.A.1 20
Permit / MISRA / C:2012 / R-16.6.A.1 21
Permit / MISRA / C:2012 / R-16.7.A.1 21
Permit / MISRA / C:2012 / R-18.1.A.1 21

5 References 23

Appendix A Deviation permit summary 24

1 Introduction

This document presents a number of *deviation permits* for use with the MISRA C:2012 Guidelines [1] [2]. It should be used in conjunction with MISRA Compliance [3], a companion document which describes the purpose of *deviation permits* and which sets out the principles by which the concept of MISRA Compliance is governed.

The *deviation permits* published in this document cover commonly encountered use cases where it has been found that *deviation* from the requirements of MISRA C:2012 is a rational and necessary response to a particular *guideline violation*.

Note: Publication of common use cases as *deviation permits* by MISRA does not imply that they are acceptable within a particular project and their use in support of a *deviation* must be subjected to the same balances and measures as for any other *deviation*.

1.1 Scope

The following versions of the C Standard are referenced within this document:

- ISO/IEC 9899:1990 [4], amended and corrected by [5], [6] and [7] — referred to as “C90”
- ISO/IEC 9899:1999 [8], corrected by [9], [10] and [11] — referred to as “C99”
- ISO/IEC 9899:2011 [12], corrected by [13] — referred to as “C11”
- ISO/IEC 9899:2018 [14] — referred to as “C18”

Unless stated otherwise, permits are applicable to all versions of the language.

2 Introduction to the permits

The *deviation permits* presented within this document have the following structure:

Related guideline

Permit / MISRA / C:2012 / guideline.identifier.version

Use case

Reason Justification

Background

...

Requirements

...

where:

- "Related guideline" gives the guideline number and headline text of the guideline which is to be *deviated*;
- "Permit" forms a unique identifier for the *deviation permit*:
  - "MISRA" identifies that the permit has been drafted by MISRA;
  - "C:2012" identifies The Guidelines for which the *deviation permit* has been developed;
  - "guideline" identifies the applicable *guideline* within The Guidelines (D-X.Y for Directives, R-X.Y for Rules);
  - "identifier" identifies a particular use case associated with the *guideline*;
  - "version" allows specific versions of a *deviation permit* to be uniquely identified.
- "Use case" defines the conditions under which *violation* of the related *guideline* may be supported by the *deviation permit*;
- "Justification" gives the reason identified within MISRA Compliance [3] used to justify why *deviation* is acceptable for the use case;
- "Background" contains information giving further details of the use case;
- "Requirements" defines a set of requirements that shall be followed when the *deviation permit* is used to support a *deviation*.

3 Directive permits

This edition does not include permits for any of the Directives.

4 Rule permits

Rule 2.1 A project shall not contain unreachable code

Permit / MISRA / C:2012 / R-2.1.A.1

A section of code is unreachable in a particular build configuration

Reason Code quality (Reusability)

Background

Unreachable code is sometimes introduced when software is designed to be built under different configurations for a product line. Code which is unreachable in a particular configuration will be reachable under different build conditions.

When developing source code which is to be configured in different ways, it may be preferable to tolerate code which is unreachable in a particular build configuration in order to make the system easier to understand and maintain.

Unreachable code is a typical consequence of an invariant operation which will constitute a violation of Rule 14.3.

Requirements

1. All instances of *unreachable code* shall be identified;
2. When *unreachable code* exists to support alternative build configurations, it shall be confirmed that all code shall be reachable in at least one of the alternative build configurations. This does not imply that every alternative build configuration is currently implemented but simply that the unreachable statements exist intentionally.

Note: The presence of unreachable code as a result of invariance will introduce a violation of Rule 14.3, requiring a deviation supported by Permit / MISRA / C:2012 / R-14.3.A.1. It is acceptable to use a single deviation record to cover both the Rule 2.1 and Rule 14.3 violations.

Example

An external sensor or mechanical switch is only used in the Japanese version of a product. The input signal is fixed in the European version, resulting in *unreachable code*.

```
void f ( void )
{
  #if ( MARKET == EUR )
    input = ON;                /* Input is not connected for EUR */
  #elif ( MARKET == JPN )
    input = g_current_data;    /* Sensor is connected for JPN    */
  #else
    #error "Unsupported market"
  #endif

  if ( input == ON )
  {
    /* Reachable for EUR and JPN */
  }
  else
  {
    /* Only reachable for JPN */
  }
}
```

Rule 2.1 A project shall not contain unreachable code

Permit / MISRA / C:2012 / R-2.1.B.1

A section of code is unreachable as a result of defensive coding measures

Reason Code quality (Fault tolerance)

Background

The intention behind defensive coding practices is to engineer robustness into a software system by anticipating the unexpected. The unexpected behaviour could arise from a number of causes such as a hardware failure, a coding error leading to data corruption, or an error introduced during maintenance.

A common feature of defensive code is the introduction of logical operations which are *invariant* but another common natural consequence is the presence of code which is *unreachable*. Unreachable code is a violation of Rule 2.1 and invariant logical operations are a violation of Rule 14.3.

Invariant expressions and unreachable code are concepts based on the assumption that a program behaves in accordance with the language specification. In critical systems it may be necessary to suspend such assumptions and introduce code which will provide a measure of protection in case some form of memory corruption or hardware failure should occur.

The task of identifying invariant logical expressions and unreachable code is an *undecidable* problem. Some examples may be easy to identify but even the most sophisticated of static analysis tools cannot guarantee to identify every instance. However, compilers are becoming increasingly sophisticated and will sometimes be capable of identifying operations which are redundant and statements which are unreachable; they are then at liberty to eliminate such code. This is a concern as defensive code is typically introduced to respond to situations where data corruption has occurred. If data corruption has occurred, any assumptions inherent in the compiler analysis will be invalidated.

If defensive code is designed to respond to the unexpected, it is vitally important that a compiler should not be able to remove the code through optimization. One technique sometimes used to circumvent the compiler's urge to optimize is to apply volatile type qualification to a critical variable. The effect of volatile qualification is to instruct the compiler that the value of the variable can never be assumed because it can be modified in ways that are not under control of the code. If the value of this variable is then tested in a logical operation, it will not be reasonable for the compiler to identify the operation as invariant and remove the operation through optimization.

Defensive code is sometimes introduced to provide a measure of protection against errors which could be introduced in future code modifications.

Rule 15.7 requires that an *if ... else if* sequence should always be followed by an *else* clause. It may sometimes happen that in conforming to Rule 15.7, the controlling expression of the final *else if* clause is invariant (always *true*), in which case, the final *else* clause will be *unreachable*. Of course, in such a situation it may be appropriate to dispense with the final *else* clause altogether and replace the *else if* with *else* thereby eliminating the invariant operation. However, the *unreachable else* clause may still serve a useful purpose in providing protection against future coding mistakes by generating suitable diagnostics or performing actions to mitigate any adverse consequences.

A similar situation, resulting in a Rule 2.1 violation, sometimes arises when introducing an unreachable *default* clause in a *switch* statement. Rule 16.4 requires a *default* clause to be located in every *switch* statement, even when it can be demonstrated that the clause is *unreachable*.

Requirements

1. Code shall not be *unreachable* for defensive coding reasons if this can be avoided by appropriate application of the *volatile* qualifier;
2. All instances of *unreachable code* shall be identified and documented in the code by means of an explanatory comment;
3. If *unreachable code* is included for defensive purposes, the binary produced by the compiler and linker shall be investigated to ensure that the code is present in the final executable.

Note: Studies [15] have shown that some compilers fail to honour *volatile* qualification correctly in some situations.

Example

The value of the critical variable `state` in the following code should never be other than `S1` or `S2`. The *default* switch clause provides assurance that a plausible value will be reinstated if its value should ever be corrupted. *Volatile* qualification is applied in order to prevent the compiler treating the *default* clause as *unreachable*, with the risk that it might remove the code altogether.

```
bool_t f ( uint8_t i )
{
  static volatile enum { S1, S2 } state = S1;

  switch ( state )
  {
    case S1: if ( i != 0 ) { state = S2; } break;
    case S2: if ( i != 0 ) { state = S1; } break;
    default:               { state = S1; } break;
  }

  return ( state == S2 );
}
```

Rule 5.3 An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Permit / MISRA / C:2012 / R-5.3.A.1

In automatically generated code, an identifier declared in an inner scope hides an identifier declared in an outer scope

Reason Adopted code integration

Background

Rule 5.3 addresses the concern that developers might be confused by identical identifiers declared in different scopes. An automatic code generator is capable of tracking the identifiers that it uses and is not confused about any identifier reuse.

When manually generated code is injected into automatically generated code, or vice-versa, it is possible for an identifier to be hidden without the code generator being aware.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. This *deviation permit* shall only be applied to automatically generated code where both the identifier that is being hidden and the one that is hiding another identifier are declared in the same automatically generated code.

Rule 6.1 Bit-fields shall only be declared with an appropriate type

Permit / MISRA / C:2012 / R-6.1.A.1

In C90 code, bit-fields are defined with an integer type other than unsigned int or signed int

Reason Code quality (Resource utilization)

Background

C90 requires that the type of a bit-field should be either *int*, *signed int* or *unsigned int*. The behaviour is *undefined* if a bit-field is defined with any other type. There are 3 distinct issues to consider:

1. Under Rule 6.1, it is not permitted to define a bit-field of type *int*. This prohibition exists because a bit-field of type *int* can be either *signed* or *unsigned* depending on the implementation. Defining bit-fields explicitly as either *signed* or *unsigned* removes the potential for *implementation defined behaviour*.
2. Many C90 compilers support the use of other integer types for bit-fields, for example, *signed char* or *unsigned char*. The use of these non-conforming types may beneficially affect the alignment and access characteristics of *struct* members, yielding significant advantages in terms of performance and memory usage. It should be noted that the C99 language standard permits the use of any integer type supported by the implementation.
3. Some compilers implement non-standard integer types (e.g. type *bit*) in support of bitwise addressing modes.

Requirements

1. When supported by a C90 implementation, the following basic types (including any explicitly signed compatible version or an equivalent *typedef*) may be used to define a bit-field:
   - *signed char* and *unsigned char*
   - *signed short* and *unsigned short*
   - *signed int* and *unsigned int*
   - *signed long* and *unsigned long*
2. When supported by the implementation, type _Bool may be used to define a bit-field;
3. When supported by the implementation, any other integer type not supported by the C90 language standard may be used to define a bit-field;
4. A type other than *unsigned int* or *signed int* shall only be used to define a bit-field when it can be demonstrated that a performance improvement is both available and necessary.

Note: The use of a type covered by requirement (2) or (3) will also constitute a violation of advisory Rule 1.2.

Rule 7.1 Octal constants shall not be used

Permit / MISRA / C:2012 / R-7.1.A.1

In automatically generated code, octal constants are used

Reason Adopted code integration

Background

Developers writing constants that have a leading zero might expect them to be interpreted as decimal constants. An automatic code generator is unlikely to generate an octal constant unintentionally.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 7.2 A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type

Permit / MISRA / C:2012 / R-7.2.A.1

In automatically generated code, integer constants that are represented in an unsigned type have no “u” or “U” suffix.

Reason Adopted code integration

Background

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).

An automatic code generator is aware of these factors and does not need to make the signedness of constants explicit.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 7.3 The lowercase character "l" shall not be used in a literal suffix

Permit / MISRA / C:2012 / R-7.3.A.1

In automatically generated code, the lowercase character "l" is used in a literal suffix

Reason Adopted code integration

Background

In automatically generated code that is not intended to be read, reviewed or modified by human programmers, there no possibility that the developer confuses "`1`" (digit 1) and "`l`" (letter "el").

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 8.4 A compatible declaration shall be visible when an object or function with external linkage is defined

Permit / MISRA / C:2012 / R-8.4.A.1

In automatically generated code, there is no compatible declaration visible when an object or function with external linkage is defined

Reason Adopted code integration

Background

If a declaration for an object or function is not visible when that object or function is defined, a compiler cannot check that the declaration and definition are compatible. However, this is not the only strategy for guaranteeing compatibility. For example, an automatic code generator may hold the declaration of each object or function in a data dictionary and use that declaration in each translation that needs it.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. It shall be demonstrated, for example by documentation or tool qualification, that automatically generated declarations and the related definition are compatible.

Rule 8.5 An external object or function shall be declared once in one and only one file

Permit / MISRA / C:2012 / R-8.5.A.1

In automatically generated code, external objects or functions are declared more than once in one or more files

Reason Adopted code integration

Background

If there is more than one declaration for an object or function in one or more files, there may be incompatibilities between the declarations and the definition. An automatic code generator may hold

the declaration of each object or function in a data dictionary and ensure that all declarations of an object or function in the generated code are identical and compatible with the definition of the object or function.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. It shall be demonstrated, for example by documentation or tool qualification, that automatically generated declarations and the related definition are compatible.

Rule 8.14 The restrict type qualifier shall not be used

Permit / MISRA / C:2012 / R-8.14.A.1

In automatically generated code, the restrict type qualifier is used

Reason Adopted code integration

Background

Rule 8.14 states that 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. If *restrict* is used incorrectly, there is a significant risk that a compiler will generate code that does not behave as expected.

If an automatic code generator can determine that two pointers do not alias, it may use the *restrict* qualifier to improve the efficiency of code generated by a compiler.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. Any function that is defined with the *restrict* qualifier must not be called from manually written code unless it is demonstrated that the actual parameters respect the *restrict* specifications.

Rule 9.2 The initializer for an aggregate or union shall be enclosed in braces

Permit / MISRA / C:2012 / R-9.2.A.1

In automatically generated code, initializers for an aggregate or union are not enclosed in braces

Reason Adopted code integration

Background

This rule is for the benefit of human developers and reviewers. Using braces to indicate initialization of sub-objects 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 which is not relevant for automatically generated code.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 9.3 Arrays shall not be partially initialized

Permit / MISRA / C:2012 / R-9.3.A.1

In automatically generated code, an array is partially initialized

Reason Adopted code integration

Background

A partially initialized array might be indicative of a programming error in manually generated code. An automatic code generator is aware of C's initialization rules and does not need to initialize every object explicitly when default initialization has the same effect.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 9.5 Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Permit / MISRA / C:2012 / R-9.5.A.1

In automatically generated code, designated initializers are used to initialize an array object but the size of the array is not specified explicitly

Reason Adopted code integration

Background

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 for a human developer which initializer has the highest index, especially when the initializer contains a large number of elements. An automatic code generator can keep track of all the initialized elements and the size of the array.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 10.1 Operands shall not be of an inappropriate essential type

Permit / MISRA / C:2012 / R-10.1.A.1

In automatically generated code, operands are of an inappropriate essential type

Reason Adopted code integration

Background

Rule 10.1 imposes some restrictions on the usage of certain essential types for certain operators. It intends to avoid problems associated with integer promotions and the usual arithmetic conversions. An automatic code generator may adopt a different approach to avoiding such problems which is equally valid. Further, the automatic code generator is aware of both the implicit type conversions defined within The Standard and the sizes of the types on the target machine.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. This *deviation permit* shall only be applied in the following cases of inappropriate essential types:
   2.1 The operands of one of the `!`, `&&` or `||` operators is of *essentially character*, *essentially enum*, *essentially signed* or *essentially unsigned* type;
   2.2 The left operand of the >> or << operators is of *essentially signed type* and the project is aware of the implementation's behaviour.

Rule 10.3 The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Permit / MISRA / C:2012 / R-10.3.A.1

In automatically generated code, the value of an expression is assigned to an object with a narrower essential type or of a different essential type category

Reason Adopted code integration

Background

The C language permits assignments between different arithmetic types to be performed automatically. However, the use of implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. An automatic code generator is able to avoid unintended results as it is aware of both the implicit type conversion defined within The Standard and the implemented sizes of the types on the target machine.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. This *deviation permit* shall only be applied if the code generator uses a different, documented, strategy to ensure the expected result.
3. This *deviation* permit shall only be applied if the compiler's implementation has been verified to be consistent with the code generator's configuration (implicit or explicit).

Rule 10.4 Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Permit / MISRA / C:2012 / R-10.4.A.1

In automatically generated code, operands of an operator in which the usual arithmetic conversions are performed have different essential type category

Reason Adopted code integration

Background

The C language permits assignments between different arithmetic types to be performed automatically. However, the use of implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. An automatic code generator is able to avoid unintended

results as it is aware of both the implicit type conversion defined within The Standard and the implemented sizes of the types on the target machine.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. This *deviation permit* shall only be applied if the code generator uses a different, documented, strategy to ensure the expected result.
3. This *deviation* permit shall only be applied if the compiler's implementation has been verified to be consistent with the code generator's configuration (implicit or explicit).

Rule 10.6 The value of a composite expression shall not be assigned to an object with wider essential type

Permit / MISRA / C:2012 / R-10.6.A.1

In automatically generated code, the value of a composite expression is assigned to an object with wider essential type

Reason Adopted code integration

Background

There may be 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*. There is also 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.

An automatic code generator is able to avoid unintended results as it is aware of both the implicit type conversion defined within The Standard and the implemented sizes of the types on the target machine.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

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

Permit / MISRA / C:2012 / R-10.7.A.1

In automatically generated code, a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed and the other operand has wider essential type

Reason Adopted code integration

Background

There may be 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*. There is also 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.

An automatic code generator is able to avoid unintended results as it is aware of both the implicit type conversion defined within The Standard and the implemented sizes of the types on the target machine.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 10.8 The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Permit / MISRA / C:2012 / R-10.8.A.1

In automatically generated code, the value of a composite expression is cast to a different essential type category or a wider essential type

Reason Adopted code integration

Background

There may be 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*. There is also 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.

An automatic code generator is able to avoid unintended results as it is aware of both the implicit type conversion defined within The Standard and the implemented sizes of the types on the target machine.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. This *deviation permit* shall only be applied if the compiler's implementation has been verified to be consistent with the code generator's configuration (implicit or explicit).

Rule 11.9 The macro NULL shall be the only permitted form of integer null pointer constant

Permit / MISRA / C:2012 / R-11.9.A.1

In automatically generated code, an integer null pointer constant other than the macro NULL or (void *) 0 is used

Reason Adopted code integration

Background

Using `NULL` rather than `0` makes it clear that a *null pointer constant* was intended. An automatic code generator is aware of the differences between `0` and a *null pointer constant*.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 14.2 A for loop shall be well-formed

Permit / MISRA / C:2012 / R-14.2.A.1

In automatically generated code, a for-loop is not well-formed

Reason Adopted code integration

Background

This rule is for the benefit of human developers and reviewers. Using a restricted form of loop makes code easier to review and to analyse, which is not relevant if the code is not intended to be reviewed or analysed.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Note: this requirement cannot be satisfied if manually written code is injected into the automatically generated code as code review will then be required.

Rule 14.3 Controlling expressions shall not be invariant

Permit / MISRA / C:2012 / R-14.3.A.1

The result of a logical operation is invariant in a particular build configuration

Reason Code quality (Reusability)

Background

Invariant operations are often introduced when software is designed to be built under different configurations for a product line. For example, a logical operation which is “always true” in a particular configuration may evaluate to “false” in a different build.

When developing source code which is to be configured in different ways, it may be preferable to tolerate some logical operations which are invariant in a particular build configuration in order to make the code easier to understand and maintain.

Requirements

1. The reason for the invariant operation shall be justified and documented.

Note: The presence of the invariance will introduce unreachable code in violation of Rule 2.1, requiring a deviation supported by Permit / MISRA / C:2012 / R-2.1.A.1. It is acceptable to use a single deviation record to cover both the Rule 2.1 and Rule 14.3 violations.

Rule 14.3 Controlling expressions shall not be invariant

Permit / MISRA / C:2012 / R-14.3.B.1

An invariant logical operation is present as a result of defensive coding measures

Reason Code quality (Fault tolerance)

Background

The intention behind defensive coding practices is to engineer robustness into a software system by anticipating the unexpected. The unexpected behaviour could arise from a number of causes such as a hardware failure, a coding error leading to data corruption, or an error introduced during maintenance.

A common feature of defensive code is the introduction of logical operations which are *invariant* but another common natural consequence is the presence of code which is *unreachable*. Invariant logical operations are a violation of Rule 14.3 and unreachable code is a violation of Rule 2.1.

Invariant expressions and unreachable code are concepts based on the assumption that computer code behaves in accordance with the language specification. In critical systems it may be necessary to suspend such assumptions and introduce code which will provide a measure of protection in case some form of memory corruption or hardware failure should occur.

The task of identifying invariant logical expressions and unreachable code is an *undecidable* problem. Some examples may be easy to identify but even the most sophisticated of static analysis tools cannot guarantee to identify every instance. However, compilers are becoming increasingly sophisticated and will sometimes be capable of identifying operations which are redundant and statements which are unreachable; they are then at liberty to eliminate such code. This introduces a danger because defensive code is typically introduced to respond to situations where data corruption has occurred. If data corruption has occurred, any assumptions inherent in the compiler analysis will be invalidated.

If defensive code is designed to respond to the unexpected, it is vitally important that a compiler should not be able to remove the code through optimization. One technique sometimes used to circunvent the compiler's urge to optimize, is to apply volatile type qualification to a critical variable. The effect of volatile qualification is to instruct the compiler that the value of the variable can never be assumed because it can be modified in ways that are not under control of the code. If the value of this variable is then tested in a logical operation, it will not be reasonable for the compiler to identify the operation as invariant and remove the operation through optimization.

Defensive code is sometimes introduced to provide a measure of protection against errors which could be introduced in future code modifications.

Rule 15.7 requires that an *if ... else if* sequence should always be followed by an *else* clause. It may sometimes happen that in conforming to Rule 15.7, the controlling expression of the final *else if* clause is invariant (always *true*), in which case, the final *else* clause will be *unreachable*. Of course, in such a situation it may be appropriate to dispense with the final *else* clause altogether and replace the *else if* with *else* thereby eliminating the invariant operation. However, the *unreachable else* clause may still serve a useful purpose in providing protection against future coding mistakes by generating suitable diagnostics or performing actions to mitigate any adverse consequences.

A similar situation, resulting in a Rule 2.1 violation, sometimes arises when introducing an *unreachable* `default` clause in a *switch* statement. Rule 16.4 requires a *default* clause to be located in every *switch* statement, even when it can be demonstrated that the clause is *unreachable*.

Requirements

1. An operation shall not be *invariant* for defensive coding reasons if invariance can be avoided by appropriate application of the *volatile* qualifier;
2. The reason for the invariant operation shall be documented;
3. The binary code shall be examined to ensure that the defensive code has not been removed through optimization.

Note: Studies [15] have shown that some compilers fail to honour *volatile* qualification correctly in some situations.

Rule 14.3 Controlling expressions shall not be invariant

Permit / MISRA / C:2012 / R-14.3.C.1

An invariant logical operation is introduced at system level as a result of interfacing to adopted code

Reason Adopted code integration

Background

Demonstrating that a particular logical operation is invariant can be a simple task or it can be very difficult. In the following code, it is obvious that the > operation is invariant (its result is always *false*).

```
if ( ( n < 10 ) && ( n > 20 ) )
{
  /* ... */
}
```

However, in the following example it is impossible to determine whether the > operation is invariant without examination of the wider context to determine the possible value domain of the variable `n`. Compliance with Rule 14.3 cannot be guaranteed when viewing the function `foo` in isolation.

```
extern void foo ( uint32_t n )
{
  if ( n > 30 )
  {
    /* ... */
  }
}
```

The task of identifying an invariant logical expression is described as an *undecidable* problem. Some examples may be easy to identify but even the most sophisticated of static analysis tools cannot guarantee to identify every instance. In the above example, there is no evidence to suggest that the operation will be invariant but it is not possible to prove this without access to the entire system code base. The identification of *invariant* operations generally requires system-wide code analysis.

For this reason, a code library which exhibits no violation of a particular rule when viewed in isolation, can still introduce non-compliance with that rule when adopted in a project. An operation which is not intrinsically invariant may become invariant according to how it is used in a particular system.

In the following example, the value of a variable is subject to the same range check in two different functions within different code modules. If the two range checks always occur in the same sequence, the second operation will be invariant and therefore technically a violation of Rule 14.3. Of course, if both range checks occurred within the same code module, the duplication would be evidence of poor design; but such duplication and the redundancy which results can easily occur when integrating modules which have not been designed within the same project.

```
// header.h
#define HMAX 30U
extern void f2 ( uint32_t n );

// file1.c
#include "header.h"

extern void f1 ( uint32_t n )
{
  if ( n > HMAX )
  {
    n = HMAX;
  }

  f2 ( n );
}

// adopted_code.c
#include "header.h"

extern void f2 ( uint32_t n )
{
  if ( n > HMAX )   /* Invariant? */
  {
    n = HMAX;
  }
}
```

Redundancy may also arise in the form of duplication when two libraries which have been developed separately contain some elements of functionality which overlap.

Requirements

1. The invariant operation associated with this *deviation permit* is introduced as a consequence of integrating adopted code. The violation is not associated with an individual code module, but occurs as a result of using code modules which have not been designed within the same development project.

Rule 15.7 All if ... else if constructs shall be terminated with an else statement

Permit / MISRA / C:2012 / R-15.7.A.1

In automatically generated code, an if ... else if construct is not terminated with an else statement

Reason Adopted code integration

Background

Terminating a sequence of *if ... else if* constructs with an *else* statement is defensive programming. An automatic code generator can determine that an *else* statement is not necessary.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Rule 16.1 All switch statements shall be well-formed

Permit / MISRA / C:2012 / R-16.1.A.1

In automatically generated code, a switch statement is not well-formed

Reason Adopted code integration

Background

The syntax for the *switch* statement in C is not particularly rigorous and can allow complex, unstructured behaviour which might be confusing to the developer. An automatic code generator is aware of the *switch* statement syntax.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. This deviation permit may only be applied when the violation corresponds to a use case covered by one or more of:

   Permit / MISRA / C:2012 / R-16.3.A.1
   Permit / MISRA / C:2012 / R-16.4.A.1
   Permit / MISRA / C:2012 / R-16.5.A.1
   Permit / MISRA / C:2012 / R-16.6.A.1
   Permit / MISRA / C:2012 / R-16.7.A.1

Rule 16.3 An unconditional break statement shall terminate every switch-clause

Permit / MISRA / C:2012 / R-16.3.A.1

In automatically generated code, a switch-clause is not terminated by an unconditional break statement

Reason Adopted code integration

Background

If a developer fails to end a *switch-clause* with a *break* statement, then control flow "falls" into the following *switch-clause* or, if there is no such clause, off the end and into the statement following the *switch* statement. Whilst falling into a following *switch-clause* is sometimes intentional, it is often an error. An automatic code generator is capable of handling *switch-clauses* that fall through into subsequent clauses.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Note: Requirement 1 cannot be satisfied if manually written code is injected into the automatically generated code as code review will then be required.

Note: A tool may also report a violation of Rule 16.1 – Permit / MISRA / C:2012 / R-16.1.A.1 exists to cover this situation.

Rule 16.4 Every switch statement shall have a default label

Permit / MISRA / C:2012 / R-16.4.A.1

In automatically generated code, a switch statement has no default label.

Reason Adopted code integration

Background

A *default* label is needed for defensive programming. Any statements following the *default* label are intended to take some appropriate action.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.
2. An automatic code generator may only omit a *default* clause that is suggested by the model (when every value for the controlling expression is covered by a *case* clause, for example) if it inserts a comment into the generated code to explain why the *default* clause is absent. This comment can be reviewed as part of the MISRA C compliance argument.

Note: A tool may also report a violation of Rule 16.1 – Permit / MISRA / C:2012 / R-16.1.A.1 exists to cover this situation.

Rule 16.5 A default label shall appear as either the first or the last switch label of a switch statement

Permit / MISRA / C:2012 / R-16.5.A.1

In automatically generated code, the default label is neither the first nor the last switch label of a switch statement.

Reason Adopted code integration

Background

It is not necessary to locate the *default* label in automatically generated coded if it is not read by humans.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Note: A tool may also report a violation of Rule 16.1 – Permit / MISRA / C:2012 / R-16.1.A.1 exists to cover this situation.

Rule 16.6 Every switch statement shall have at least two switch-clauses

Permit / MISRA / C:2012 / R-16.6.A.1

In automatically generated code, a switch statement has only one switch-clause

Reason Adopted code integration

Background

A *switch* statement with a single path is redundant and might be indicative of a programming error in manually generated code but may be necessary or desirable in automatically generated code.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Note: A tool may also report a violation of Rule 16.1 – Permit / MISRA / C:2012 / R-16.1.A.1 exists to cover this situation.

Rule 16.7 A switch-expression shall not have essentially Boolean type

Permit / MISRA / C:2012 / R-16.7.A.1

In automatically generated code, a switch-expression has essentially Boolean type

Reason Adopted code integration

Background

A *switch* statement which has a controlling expression with *essentially Boolean* type might be indicative of a programming error in manually generated code but may be necessary or desirable in automatically generated code.

Requirements

1. This *deviation permit* shall only be applied to automatically generated code that is not intended to be read, reviewed or modified by human programmers.

Note: A tool may also report a violation of Rule 16.1 – Permit / MISRA / C:2012 / R-16.1.A.1 exists to cover this situation.

Rule 18.1 A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Permit / MISRA / C:2012 / R-18.1.A.1

Arithmetic operations are performed on pointers that are used to address a region in memory which is not an array object

Reason Access to hardware, Code quality (Modifiability)

Background

There are particular situations in which it may be necessary to access areas of memory which are not strictly of array type but which nevertheless behave like array objects, for example, memory space which is accessed at a specific hardware address.

Requirements

1. Pointer arithmetic may be performed on pointers which address a block of memory that is being accessed at some absolute hardware address;
2. All instances where such operations occur shall be identified;
3. The validity of all memory addresses computed in this way shall be verified.

Example

Verifying a ROM checksum following reset of the micro-controller:

```
#define CODE_SIZE  0x1000u
#define ROM_STAADR 0xE000u

void check1 ( void )
{
  uint8_t  *mem      = ( uint8_t * ) ROM_STAADR; /* ROM start address */
  uint8_t   checksum = 0u;                       /* Computed checksum */
  uint16_t  i;

  for( i = 0u; i < CODE_SIZE; i++ )
  {
    checksum += *mem;                  /* Update checksum             */
    mem++;                             /* Rule 18.1 violation         */
  }
}
```

Clearing internal RAM following reset of the micro-controller:

```
#define RAM_STAADR 0xF000u
#define RAM_ENDADR 0xFFFFu

void clear ( void )
{
  uint8_t  *mem      = ( uint8_t * ) RAM_STAADR;      /* RAM start address */
  uint32_t  ram_size = RAM_ENDADR - RAM_STAADR + 1;   /* RAM length        */
  uint16_t  i;

  for ( i = 0u; i < ram_size; i++ )                   /* Clear all RAM     */
  {
    mem[ i ] = 0u;                           /* Rule 18.1 violation         */
  }
}
```

It is possible to achieve the same result without violating Rule 18.1, but at the expense of code which may be considered less readable:

```
typedef uint8_t ( *pCode_t )[ CODE_SIZE ];

#define codeMemory ( *pCode );

void check2 ( void )
{
  /* Declare a pointer to a CODE_SIZE array of uint8_t */
  pCode_t pCode = ( pCode_t ) ROM_STAADR;

  uint8_t  checksum = 0u;
  uint16_t i;

  for( i = 0u; i < CODE_SIZE; i++ )
  {
    checksum += codeMemory[ i ];        /* Update checksum */
  }
}
```

5 References

[1] MISRA C:2012 *Guidelines for the use of the C language in critical systems* (3rd Edition), ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), MIRA Limited, Nuneaton, March 2013

[2] MISRA C:2012 *Guidelines for the use of the C language in critical systems* (3rd Edition, 1st Revision), ISBN 978-1-906400-21-7 (paperback), ISBN 978-1-906400-22-4 (PDF), HORIBA MIRA Limited, Nuneaton, February 2019

[3] MISRA Compliance:2020 *Achieving compliance with MISRA Coding Guidelines*, ISBN 978-1-906400-26-2 (PDF), HORIBA MIRA Limited, Nuneaton, February 2020

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

[5] ISO/IEC 9899:1990/COR 1:1995, *Technical Corrigendum 1*, International Organization for Standardization, 1995

[6] ISO/IEC 9899:1990/AMD 1:1995, *Amendment 1*, International Organization for Standardization, 1995

[7] ISO/IEC 9899:1990/COR 2:1996, *Technical Corrigendum 2*, International Organization for Standardization, 1996

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

[9] ISO/IEC 9899:1999/COR 1:2001, *Technical Corrigendum 1*, International Organization for Standardization, 2001

[10] ISO/IEC 9899:1999/COR 2:2004, *Technical Corrigendum 2*, International Organization for Standardization, 2004

[11] ISO/IEC 9899:1999/COR 3:2007, *Technical Corrigendum 3*, International Organization for Standardization, 2007

[12] ISO/IEC 9899:2011, *Programming languages — C*, International Organization for Standardization, 2011

[13] ISO/IEC 9899:2011/COR 1:2012, *Technical Corrigendum 1*, International Organization for Standardization, 2012

[14] ISO/IEC 9899:2018, *Programming languages — C*, International Organization for Standardization, 2018

[15] *Volatiles Are Miscompiled, and What to Do about It*, Eric Eide and John Regehr, University of Utah, School of Computing, 2008 https://collections.lib.utah.edu/details?id=708000 (last accessed 2021-02-25).

Appendix A Deviation permit summary

Rule permits

Permit / MISRA / C:2012 / R-2.1.A.1 *Code quality (Reusability)*

A section of code is unreachable in a particular build configuration

Permit / MISRA / C:2012 / R-2.1.B.1 *Code quality (Fault tolerance)*

A section of code is unreachable as a result of defensive coding measures

Permit / MISRA / C:2012 / R-5.3.A.1 *Adopted code integration*

In automatically generated code, an identifier declared in an inner scope hides an identifier declared in an outer scope

Permit / MISRA / C:2012 / R-6.1.A.1 *Code quality (Resource utilization)*

In C90 code, bit-fields are defined with an integer type other than unsigned int or signed int

Permit / MISRA / C:2012 / R-7.1.A.1 *Adopted code integration*

In automatically generated code, octal constants are used

Permit / MISRA / C:2012 / R-7.2.A.1 *Adopted code integration*

In automatically generated code, integer constants that are represented in an unsigned type have no “u” or “U” suffix.

Permit / MISRA / C:2012 / R-7.3.A.1 *Adopted code integration*

In automatically generated code, the lowercase character “l” is used in a literal suffix

Permit / MISRA / C:2012 / R-8.4.A.1 *Adopted code integration*

In automatically generated code, there is no compatible declaration visible when an object or function with external linkage is defined

Permit / MISRA / C:2012 / R-8.5.A.1 *Adopted code integration*

In automatically generated code, external objects or functions are declared more than once in one or more files

Permit / MISRA / C:2012 / R-8.14.A.1 *Adopted code integration*

In automatically generated code, the restrict type qualifier is used

Permit / MISRA / C:2012 / R-9.2.A.1 *Adopted code integration*

In automatically generated code, initializers for an aggregate or union are not enclosed in braces

Permit / MISRA / C:2012 / R-9.3.A.1 *Adopted code integration*

In automatically generated code, an array is partially initialized

Permit / MISRA / C:2012 / R-9.5.A.1 *Adopted code integration*

In automatically generated code, designated initializers are used to initialize an array object but the size of the array is not specified explicitly

Permit / MISRA / C:2012 / R-10.1.A.1 *Adopted code integration*

In automatically generated code, operands are of an inappropriate essential type

Permit / MISRA / C:2012 / R-10.3.A.1 *Adopted code integration*

In automatically generated code, the value of an expression is assigned to an object with a narrower essential type or of a different essential type category

Permit / MISRA / C:2012 / R-10.4.A.1 *Adopted code integration*

In automatically generated code, operands of an operator in which the usual arithmetic conversions are performed have different essential type category

Permit / MISRA / C:2012 / R-10.6.A.1 *Adopted code integration*

In automatically generated code, the value of a composite expression is assigned to an object with wider essential type

Permit / MISRA / C:2012 / R-10.7.A.1 *Adopted code integration*

In automatically generated code, a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed and the other operand has wider essential type

Permit / MISRA / C:2012 / R-10.8.A.1 *Adopted code integration*

In automatically generated code, the value of a composite expression is cast to a different essential type category or a wider essential type

Permit / MISRA / C:2012 / R-11.9.A.1 *Adopted code integration*

In automatically generated code, an integer null pointer constant other than the macro NULL or (void

Permit / MISRA / C:2012 / R-14.2.A.1 *Adopted code integration*

In automatically generated code, a for-loop is not well-formed

Permit / MISRA / C:2012 / R-14.3.A.1 *Code quality (Reusability)*

The result of a logical operation is invariant in a particular build configuration

Permit / MISRA / C:2012 / R-14.3.B.1 *Code quality (Fault tolerance)*

An invariant logical operation is present as a result of defensive coding measures

Permit / MISRA / C:2012 / R-14.3.C.1 *Adopted code integration*

An invariant logical operation is introduced at system level as a result of interfacing to adopted code

Permit / MISRA / C:2012 / R-15.7.A.1 *Adopted code integration*

In automatically generated code, an if ... else if construct is not terminated with an else statement

Permit / MISRA / C:2012 / R-16.1.A.1 *Adopted code integration*

In automatically generated code, a switch statement is not well-formed

Permit / MISRA / C:2012 / R-16.3.A.1 *Adopted code integration*

In automatically generated code, a switch-clause is not terminated by an unconditional break statement

Permit / MISRA / C:2012 / R-16.4.A.1 *Adopted code integration*

In automatically generated code, a switch statement has no default label.

Permit / MISRA / C:2012 / R-16.5.A.1 *Adopted code integration*

In automatically generated code, the default label is neither the first nor the last switch label of a switch statement.

Permit / MISRA / C:2012 / R-16.6.A.1 *Adopted code integration*

In automatically generated code, a switch statement has only one switch-clause

Permit / MISRA / C:2012 / R-16.7.A.1 *Adopted code integration*

In automatically generated code, a switch-expression has essentially Boolean type

Permit / MISRA / C:2012 / R-18.1.A.1 *Access to hardware, Code quality (Modifiability)*

Arithmetic operations are performed on pointers that are used to address a region in memory which is not an array object