MISRA C:2004 Permits

Deviation permits for MISRA Compliance

Edition 1, April 2016

First published April 2016 by HORIBA MIRA Limited
Watling Street
Nuneaton
Warwickshire
CV10 0TU
UK

www.misra.org.uk

ISBN 978-1-906400-14-9 PDF

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

MISRA C:2004 Permits

Deviation permits for MISRA Compliance

Edition 1, April 2016

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
April 2016	Initial release

Contents

1 Introduction

This document presents a number of *deviation permits* for use with the MISRA C:2004 Guidelines [1]. It should be used in conjunction with MISRA Compliance:2016 [2], 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:2004 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*.

2 Deviation permits

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

Permit / body / guidelines / guideline.identifier.version
Use-case

Reason Justification

where:

- "Permit" forms a unique identifier for the *deviation permit*, using the following recommended naming scheme:
  - "body" identifies the trade body or organisation responsible for drafting the *deviation permit* ("MISRA" within this document);
  - "guidelines" identifies The Guidelines for which the *deviation permit* has been developed (e.g. C:2004);
  - "guideline" identifies the applicable *guideline* within The Guidelines;
  - "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:2016 [2] used to justify why *deviation* is acceptable for the use-case.

Each *deviation permit* also includes background information giving further details of the use-case and a set of requirements that shall be followed when the *deviation permit* is used to support a *deviation*.

Rule 1.1	All code shall conform to ISO/IEC 9899:1990 C "Programming languages – C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996

Permit / MISRA / C:2004 / 1.1.A.1

The C99 types, signed long long and unsigned long long are used to implement operations on large integer values.

Reason Code quality (Time behaviour)

Background

Long long types were introduced as a new feature in ISO/IEC 9899:1999 [7][8][9][10]. Many compilers which implement ISO/IEC 9899:1990 [3][4][5][6] also provide support, as a language extension, for selected features found in ISO/IEC 9899:1999 [7][8][9][10].

If an application requires operations on large integer values which cannot be represented in *long* types, it may be advantageous to use *long long* types when they are supported by the compiler and tools.

Requirements

If *long long* types are to be used, it shall be confirmed that:

1. The type interpretation and type conversion semantic behaviour defined in ISO/IEC 9899:1999 [7][8][9][10] is supported by the compiler(s) and static analysis tool(s);
2. The static analysis tool(s) accurately enforce the MISRA C:2004 *guidelines* which address the issue of underlying type (Rules 10.1 to 10.6).

Permit / MISRA / C:2004 / 1.1.B.1

The C99 type _Bool is used to implement an effectively Boolean type.

Reason Code quality (Modifiability)

Background

ISO/IEC 9899:1990 [3][4][5][6] does not include support for a Boolean type. Appendix E of MISRA C:2004 [1] defines the concept of an *effectively Boolean* type and this concept is referred to in a number of the individual *guidelines*.

Effectively Boolean expressions appear in two forms:

1. An expression is *Boolean-by-construct* if it is the result of an equality operator (== or `!=`), a logical operator (`!`, `&&` or `||`) or a relational operator (<, >, <=, >=);
2. An expression is *Boolean-by-enforcement* when Boolean characteristics are imparted by tool support.

Type `_Bool` was introduced as a new feature in ISO/IEC 9899:1999 [7][8][9][10] and is the most natural way to enforce support for an *effectively Boolean* type. Many C90 compilers provide support for `_Bool` as an extension to the C90 language.

Requirements

1. Any expression which is assigned to an object of type `_Bool` shall be of *effectively Boolean* type or a *signed* or *unsigned integer constant expression* of value `0` or `1`;
2. The value of an expression shall not be cast to type `_Bool` unless the expression is *effectively Boolean* or of a *signed* or *unsigned* type with a value of `0` or `1`.

Permit / MISRA / C:2004 / 1.1.C.1

The C99 function specifier inline is used.

Reason Code quality (Time behaviour)

Background

The keyword *inline* is described in ISO/IEC 9899:1999 [7][8][9][10] as a *function specifier*. This language feature can be useful because it can eliminate the performance overhead associated with ordinary function calls. Many C90 compilers provide support for *inline* through the use of a language extension.

The terminology used to describe the C99 behaviour of *inline* can be confusing. The following paragraph from section 6.7.4 of ISO/IEC 9899:1999 [7][8][9][10] describes some key elements of the specification:

> Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an inline function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the inline function specifier without extern, then the definition in that translation unit is an inline definition. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.

It is important to understand the terms *inline function*, *internal linkage*, *external linkage*, *external definition* and *inline definition*:

- A function declared with the *inline* function specifier is an *inline function*;
- An *inline function*, like any other function, has either *internal linkage* or *external linkage*;
- The term *external definition* refers (subject to the exception described below) to the definition at file scope of any object or a function with either *internal* or *external linkage*;
- An *inline function* with *external linkage* is only an *external definition* if there is a declaration which includes the *extern* storage class specifier or omits the *inline* function specifier. Otherwise it is an *inline* definition.

According to paragraph 6.9 of ISO/IEC 9899:1999 [7][8][9][10]:

> ... If an identifier declared with external linkage is used in an expression (other than as part of the operand of a sizeof operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier ...

So, if an *inline definition* exists in one translation unit, there has to be an *external definition* of the same function in another translation unit. This introduces a danger because two (possibly different) definitions will be accessible and it is unspecified which will be used.

Requirements

1. All declarations of *inline functions* shall include the *static* storage class specifier in order to ensure that they have *internal linkage*;
2. If an *inline function* is used in more than one translation unit, it shall be defined exactly once within a header file.

Notes

1. The implementation of *inline* as an extension in some C90 compilers does not conform to the C99 behaviour of *inline*. For example, it may use a compiler specific definition which predates the standardization process. It should also be noted that the specification of *inline* in C99 differs from that in the C++ language standard;
2. The definition of an *inline* function within a header file will also violate Rule 8.5, requiring a deviation supported by Permit / MISRA / C:2004 / 8.5.A.1. It is acceptable to use a single deviation record to cover the Rule 1.1 and Rule 8.5 violations.

Permit / MISRA / C:2004 / 1.1.D.1

An ISO/IEC 9899:1990 environmental limit is exceeded.

Reason Code quality (Modifiability)

Background

The Environmental Limits described in ISO/IEC 9899:1990 [3][4][5][6] are of two types, Translation Limits and Numerical Limits. Both impose minimum requirements on what is described as a *conforming implementation* of the C language. For example, a conforming compiler is required to support:

- Logical source lines of at least 509 characters;
- Switch statements which contain up to 257 case labels;
- Enumerations consisting of up to 127 enumeration constants;
- Type *int* which is at least 16 bits wide;
- Type *long* which is at least 32 bits wide.

These are all minimum requirements and many compilers will exceed these limits. Source code which exceeds the minimum requirements will not be optimally portable, but it may be preferable to incur a minor loss of portability rather than to introduce code which is more complex, less maintainable and less readable.

Requirements

1. It shall be demonstrated that conformance with the environmental limit is not possible without compromising code quality, maintainability or readability;
2. It shall be confirmed that the implementation supports the required extension to the environmental limit.

Notes

1. Rule 5.1 is a *guideline* which addresses one specific example of the ISO/IEC 9899:1990 [3][4][5][6] environmental limits – the requirement that identifiers in overlapping scope and the same namespace should be distinct within 31 characters. A violation of this limit shall be deemed to be a violation of Rule 5.1 rather than Rule 1.1.

Permit / MISRA / C:2004 / 1.1.E.1

A language extension is used to access a hardware feature.

Reason Access to hardware

Background

In embedded software development it is often necessary to use language extensions to control and address hardware features which are outside the scope of the C language. For example:

- *near* and *far* pointer type qualifiers to support different memory addressing models;
- @ syntax;
- The *interrupt* keyword;
- Inline assembler code;
- Access to special function registers;
- Paged / banked / segmented memory directives;
- Non-standard types and syntax to support bitwise addressing modes.

Requirements

1. A language extension shall only be used when it is required to interact with the hardware environment;
2. The language extension shall be fully specified in the compiler documentation;
3. A justification shall be supplied for each language extension that is used;
4. Any risks associated with the use of the language extension shall be investigated and any precautionary measures which need to be adopted shall be documented;
5. The use of language extensions shall be encapsulated and isolated as far as reasonably possible;
6. If the compiler is changed or updated at any time, all language extensions shall be reviewed in order to verify that behaviour has not changed.

Notes

1. Encapsulating and isolating the use of language extensions as far as reasonably possible will aid portability.

Example

```
extern char far *ptr;

asm( "NOP" );

extern struct STM aft @0xF800u;
```

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

Permit / MISRA / C:2004 / 5.1.A.1

Identifiers defined with external linkage in different translation units are not distinct within 31 characters.

Reason Adopted code integration

Background

Rule 5.1 addresses one specific instance of the ISO/IEC 9899:1990 [3][4][5][6] environmental limits which are the subject of Permit / MISRA / C:2004 / 1.1.D.1. According to The Standard, behaviour is undefined "*... if identifiers that are intended to denote the same entity differ in a character beyond the minimal significant characters*".

Rule 5.1 and Rule 1.1 are actually slightly more relaxed than the requirements of The Standard, because of requirements imposed on the compiler / linker implementation by Rule 1.4 ("The compiler / linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers"). In practice, this can be summarised as follows:

1. An identifier must be distinct from any other identifier in overlapping scope and in the same namespace within the first 31 characters (corresponding uppercase and lowercase characters being considered distinct);
2. An identifier with external linkage must be distinct from all other identifiers with external linkage in the entire system within the first 31 characters (corresponding uppercase and lowercase characters being considered distinct);

In practice, many implementations exceed these requirements and are able to distinguish identifiers which are not distinct within 31 characters. It is therefore possible to violate Rule 5.1 at the expense of portability, providing the relevant identifiers are distinct within the number of characters supported by the implementation.

Requirements

1. This *deviation permit* shall be applied only to external identifiers, i.e. to the identifiers of objects and functions which are defined with external linkage;
2. Two external identifiers which are not distinct within 31 characters shall only be permitted to coexist within the same system if the identifiers are defined within different translation units and at least one of the translation units consists of *adopted code*;
3. It shall be established that any identifiers which are not distinct within 31 characters are distinct within the number of characters supported by the implementation.

Rule 6.4 Bit fields shall only be defined to be of type *unsigned int* or *signed int*

Permit / MISRA / C:2004 / 6.4.A.1

Bit-fields are defined with an integer type other than unsigned int or signed int.

Reason Code quality (Resource utilization)

Background

ISO/IEC 9899:1990 [3][4][5][6] 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 6.4 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 the 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.

Notes

1. The use of a type covered by requirement (2) or (3) will also constitute a violation of Rule 1.1, requiring a deviation. Permit / MISRA / C:2004 / 1.1.A.1, Permit / MISRA / C:2004 / 1.1.B.1 and Permit / MISRA / C:2004 / 1.1.E.1 cover the use of the *long long*, `_Bool` and bitwise types. It is acceptable to use a single deviation record to cover the Rule 1.1 and Rule 6.4 violations.

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

Permit / MISRA / C:2004 / 8.5.A.1

An inline function is defined within a header file.

Reason Code quality (Modularity)

Background

The intention of Rule 8.5 is to avoid the possibility of unintended code duplication. This is best achieved in the case of an ordinary function (i.e. not an *inline function*) by defining it with *external linkage* in a source file and declaring it in a *header file* which is included wherever the function is referenced.

However, the object code of an *inline function* is intentionally duplicated in every location where the function is called. If an *inline function* is required in more than one translation unit, the definition **shall** be located in a *header file* to avoid duplication of the source code.

Rule 8.5 did not consider the use of *inline functions* as they are not part of ISO/IEC 9899:1990 [3][4][5][6].

Requirements

1. A function shall not be defined within a *header file* unless it is an *inline function*.

Notes

1. The use of *inline* will also violate Rule 1.1, requiring a deviation supported by Permit / MISRA / C:2004 / 1.1.C.1. It is acceptable to use a single deviation record to cover the Rule 1.1 and Rule 8.5 violations.

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

Permit / MISRA / C:2004 / 12.7.A.1

A left-shift operation (<< or <<=) is applied to an operand whose underlying type is signed in order to perform multiplication by a value which is a power of two.

Reason Code quality (Time behaviour)

Background

Rule 12.7 applies to the complete range of bitwise operators. This *deviation permit* only applies to an operand with *signed underlying type* when it is the left-hand operand of the left-shift operator (<<) or the corresponding compound assignment operator (<<=).

When performing a multiplication by a power of two, it is sometimes possible to gain a performance advantage by using a bitwise left-shift operation instead of a multiplication operation. Whether such an advantage is realized in practice will depend on the compiler; many modern compilers will optimize the operation and use a left-shift instruction instead of a multiplication.

If the value of an expression is non-negative, a left-shift operation will generate the same result as a multiplication by the corresponding power of two, providing the result can be represented. However, if the value of the expression is negative, the behaviour is less certain. ISO/IEC 9899:1999 [7][8][9][10] describes the behaviour as undefined, whilst ISO/IEC 9899:1990 [3][4][5][6] is silent.

In practice, on many computers with a conventional architecture, performing a left-shift operation on an operand with *signed underlying type* and a negative value will generate the same result as a multiplication operation. However, the operations will not be equivalent if the representation of *signed* values is not implemented in two's complement or is complicated by the presence of padding bits.

Requirements

1. A bitwise left-shift operation shall not be performed on an operand of *signed underlying type* unless it is necessary to optimize the performance of a multiplication operation. An analysis of the comparative performance shall be performed and shift operations shall only be used to satisfy an imperative performance requirement.
2. If left-shift operations are to be performed on operands of negative value, an investigation shall be conducted to verify the behaviour of the implementation. It shall be confirmed that the results of all left-shift operations on any negative value are identical to the behaviour of a multiplication by the corresponding power of two, providing the results are representable.
3. A bitwise left-shift operation shall not be performed on an operand of *signed underlying type* unless the right-hand operand is a *constant expression*.

Permit / MISRA / C:2004 / 12.7.B.1

A right-shift operation (>> or >>=) is applied to an operand whose underlying type is signed in order to perform division by a value which is a power of two.

Reason Code quality (Time behaviour)

Background

Rule 12.7 applies to the complete range of bitwise operators. This *deviation permit* only applies to an operand with *signed underlying type* when it is the left-hand operand operand of the right-shift operator (>>) or the corresponding compound assignment operator (>>=).

When performing a division by a power of two, it is sometimes possible to gain a performance advantage by using a bitwise right-shift operation instead of a division operation. Whether such an advantage is realized in practice will depend on the compiler; many modern compilers will optimize the operation and use a right-shift instruction instead of a division.

If the value of an expression is non-negative, a right-shift operation will generate the same result as a division by the corresponding power of two. However, if the value of the expression is negative, the behaviour becomes *implementation defined*.

For the divide (`/`) operator:

- ISO/IEC 9899:1990 [3][4][5][6] states that if a division operation is performed on operands where just one operand is negative and the result is not exact, the result may either be rounded towards zero or away from zero. So, for example, the result of the expression `-1 / 2` may be either `0` or `-1`;
- ISO/IEC 9899:1999 [7][8][9][10] states that the result of an inexact division operation on a negative value is always rounded towards zero. So the result of `-1 / 2` is always `0`.

For the right shift (>>) operator:

- The result of a right shift operation on an operand of negative value is *implementation defined* in both ISO/IEC 9899:1990 [3][4][5][6] and ISO/IEC 9899:1999 [7][8][9][10];
- When performing a right shift operation, compilers may perform either a logical shift or an arithmetic shift operation. In an arithmetic shift, the sign bit is always preserved but in a logical shift, the sign bit is replaced with zero bits. The difference between a logical shift and arithmetic shift is only significant when the left hand operand is negative;
- If a logical shift is applied to a negative value, the result will probably be meaningless;
- If an arithmetic shift is applied to a negative value, the operation is equivalent to a division operation by the corresponding power of 2 but if the division is inexact, the result is rounded away from zero.

As a consequence, according to ISO/IEC 9899:1999 [7][8][9][10], the results of performing a right shift operation instead of the corresponding division operation on an operand with *signed underlying type* and negative value will only be identical if:

1. The right-shift operation is an arithmetic shift;
2. The division operation is exact and there is no remainder.

Requirements

1. A bitwise right-shift operation shall not be performed on an expression of *signed underlying type* unless it is necessary to optimize the performance of a division operation. An analysis of the comparative performance shall be performed and shift operations shall only be used to satisfy an imperative performance requirement.

2. A bitwise right-shift operation may not be performed on an integer expression of negative value if any of the discarded bits are non-zero. This is equivalent to requiring that the corresponding division operation is exact and has no remainder.

3. If right-shift operations are to be performed on expressions of negative value, an investigation shall be conducted to verify the behaviour of the implementation. It shall be confirmed that when operating on any expression of negative value whose absolute value is a power of two, the result of a right-shift operation is identical to the result of a division by the corresponding power of two.

4. A bitwise right-shift operation shall not be performed on an expression of *signed underlying type* unless the right-hand operand is a *constant expression*.

Permit / MISRA / C:2004 / 12.7.C.1

A bitwise complement (~) or bitwise exclusive OR (^ or ^=) operation is applied to an operand whose underlying type is signed in order to implement a “data mirroring” operation.

Reason Code quality (Fault tolerance)

Background

Rule 12.7 applies to the complete range of bitwise operators. This *deviation permit* only applies to an operand with *signed underlying type* when it is an operand of the bitwise complement (~) or bitwise exclusive OR (^ or ^=) operator.

The numeric value resulting from bitwise operations on data with *signed underlying type* is generally meaningless. However, the ~ and ^ operators can be used to perform “data mirroring” by creating a complementary bit pattern of the value. Subsequent comparison of the stored complementary value with a newly computed complementary value can then be used to allow data corruption to be detected.

Requirements

1. A bitwise exclusive OR (^) or a bitwise complement (~) operation shall only be applied to an operand of *signed underlying type* if the operation is performed for the purpose of data mirroring as a way of improving reliability.

Example

The following example is intended to identify a fault condition which can only arise as a result of data corruption from some external source (e.g. a cosmic ray event). A compiler implementation is not required to take external modification of objects into consideration and it is permitted to assume that the value assigned to an object will not change unexpectedly.

The *volatile* qualification applied to `var1_s16` and `var1chk_s16` requires the compiler to assume that the values assigned to them **may** change outside of its control. If this were not done, the compiler would be permitted to treat the controlling expression of the *if* statement (used to detect

the fault) as invariant, resulting in the error handling statements being *unreachable code*. Rule 13.7 and Rule 14.1 would be violated as a consequence.

```
volatile s16_t var1_s16;
volatile s16_t var1chk_s16;

var1chk_s16 = ~var1_s16;

/* Other processing */

if ( var1_s16 != ( s16_t )~var1chk_s16 )
{
  /* Error ... data corruption detected */
}
```

Rule 13.7 Boolean operations whose results are invariant shall not be permitted.

Permit / MISRA / C:2004 / 13.7.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.

Notes

1. The presence of the invariance will introduce *unreachable code* in violation of Rule 14.1, requiring a deviation supported by Permit / MISRA / C:2004 / 14.1.A.1. It is acceptable to use a single deviation record to cover the Rule 13.7 and Rule 14.1 violations.

Permit / MISRA / C:2004 / 13.7.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 13.7 and unreachable code is a violation of Rule 14.1.

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 optimize the code away. One technique sometimes used to foil 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 optimize the operation away.

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

Rule 14.9 requires that an *if ... else if* sequence should always be followed by an *else* clause. It may sometimes happen that in conforming to Rule 14.9, 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 14.1 violation, sometimes arises when introducing an *unreachable* `default` clause in a *switch* statement. Rule 15.0 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 optimized away by the compiler.

Notes

1. It should be noted that studies [11] have shown that some compilers fail to honour *volatile* qualification correctly in some situations.

Permit / MISRA / C:2004 / 13.7.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 13.7 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 13.7. 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 );
}
```

file2.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 14.1 There shall be no *unreachable code*.

Permit / MISRA / C:2004 / 14.1.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 13.7.

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.

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 */
  }
}
```

Permit / MISRA / C:2004 / 14.1.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*. Invariant logical operations are a violation of Rule 13.7 and unreachable code is a violation of Rule 14.1.

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 optimize the code away. One technique sometimes used to foil 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 optimize the operation away.

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

Rule 14.9 requires that an *if ... else if* sequence should always be followed by an *else* clause. It may sometimes happen that in conforming to Rule 14.9, 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 14.1 violation, sometimes arises when introducing an *unreachable* `default` clause in a *switch* statement. Rule 15.0 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.

Notes

1. It should be noted that studies [11] 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 sensible 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 17.1 Pointer arithmetic shall only be applied to pointers that address an array or array element.

Permit / MISRA / C:2004 / 17.1.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

Rule 17.1 states that pointer arithmetic shall only be applied to pointers that address an array or array element; however the *undefined behaviour* which this guideline seeks to avoid refers only to pointers which do not behave "like a pointer to an element of an array object".

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.

The phrase "arithmetic operations" as used in Rule 17.1 describes all the operations used to calculate an address. It therefore refers to arithmetic operators such as +, – , ++ and ––, but also to the array indexing operator `[ ]`. Note however that any such use of these operators will also violate Rule 17.4 because of the following requirements:

1. Arithmetic operations (+, –, ++, ––) shall not be applied to a pointer;
2. Array indexing, `[ ]`, shall not be applied to a pointer except when the pointer is a function parameter which has been declared with array syntax.

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.

Notes

1. Operations of this type will usually introduce Rule 17.4 violations, requiring a deviation supported by Permit / MISRA / C:2004 / 17.4.A.1. It is acceptable to use a single deviation record to cover the Rule 17.1 and Rule 17.4 violations.

Example

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

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

void check1 ( void )
{
  u8_t  *mem      = ( u8_t * ) ROM_STAADR; /* ROM start address */
  u8_t   checksum = 0u;                    /* Computed checksum */
  u16_t  i;
```

```
  for( i = 0u; i < CODE_SIZE; i++ )
  {
    checksum += *mem;            /* Update checksum             */
    mem++;                       /* Rule 17.1 & 17.4 violations */
  }
}
```

Clearing internal RAM following reset of the micro-controller:

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

void clear ( void )
{
  u8_t  *mem      = ( u8_t * ) RAM_STAADR;         /* RAM start address */
  u32_t  ram_size = RAM_ENDADR - RAM_STAADR + 1;   /* RAM length        */
  u16_t  i;

  for ( i = 0u; i < ram_size; i++ )                /* Clear all RAM     */
  {
    mem[ i ] = 0u;                        /* Rule 17.1 & 17.4 violations */
  }
}
```

It is possible to achieve the same result without violating Rule 17.1 or Rule 17.4, but at the expense of code which may be considered less readable:

```
typedef u8_t ( *pCode_t )[ CODE_SIZE ];

#define codeMemory ( *pCode );

void check2 ( void )
{
  /* Declare a pointer to a CODE_SIZE array of u8_t */
  pCode_t pCode = ( pCode_t ) ROM_STAADR;

  u8_t  checksum = 0u;
  u16_t i;

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

Rule 17.4 Array indexing shall be the only allowed form of pointer arithmetic.

Permit / MISRA / C:2004 / 17.4.A.1

Array indexing is applied to a pointer defined to address a specific memory location in the hardware memory space.

Reason Access to hardware, Code quality (Modifiability)

Background

There are particular situations in which it is necessary to access areas of memory which are not strictly of array type but which nevertheless are intended to be treated like array objects, for example, when an area of memory is accessed at a specific hardware address. It is convenient in these situations to address the memory space using a pointer to the first element of the "array" and address other elements of the array using some form of pointer arithmetic.

Violations arise when accessing memory in this way as Rule 17.4 imposes the following requirements:

1. Arithmetic operations (+, –, ++, ––) shall not be applied to a pointer;
2. Array indexing, `[ ]`, shall not be applied to a pointer except when the pointer is a function parameter which has been declared with array syntax.

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.

Notes

1. Operations of this type will introduce Rule 17.1 violations, requiring a deviation supported by Permit / MISRA / C:2004 / 17.1.A.1. It is acceptable to use a single deviation record to cover the Rule 17.1 and Rule 17.4 violations.

Example

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

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

void check1 ( void )
{
  u8_t  *mem      = ( u8_t * ) ROM_STAADR; /* ROM start address */
  u8_t   checksum = 0u;                    /* Computed checksum */
  u16_t  i;

  for( i = 0u; i < CODE_SIZE; i++ )
  {
    checksum += *mem;            /* Update checksum             */
    mem++;                       /* Rule 17.1 & 17.4 violations */
  }
}
```

Clearing internal RAM following reset of the micro-controller:

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

void clear ( void )
{
  u8_t  *mem      = ( u8_t * ) RAM_STAADR;          /* RAM start address */
  u32_t  ram_size = RAM_ENDADR - RAM_STAADR + 1;    /* RAM length        */
  u16_t  i;

  for ( i = 0u; i < ram_size; i++ )                 /* Clear all RAM     */
  {
    mem[ i ] = 0u;                        /* Rule 17.1 & 17.4 violations */
  }
}
```

It is possible to achieve the same result without violating Rule 17.1 or Rule 17.4, but at the expense of code which may be considered less readable:

```
typedef u8_t ( * pCode_t )[ CODE_SIZE ];

#define codeMemory ( *pCode );

void check2 ( void )
{
  /* Declare a pointer to a CODE_SIZE array of u8_t */
  pCode_t pCode = ( pCode_t ) ROM_STAADR;

  u8_t  checksum = 0u;
  u16_t i;

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

3 References

[1] MISRA-C:2004 *Guidelines for the use of the C language in critical systems*, ISBN 978-0-9524156-2-6, MIRA Limited, Nuneaton, October 2004

[2] MISRA Compliance:2016 *Achieving compliance with MISRA Coding Guidelines*, ISBN 978-1-906400-13-2, MIRA, Nuneaton, April 2016

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

[4] ISO/IEC 9899:1990/COR 1:1995, *Technical Corrigendum 1*, 1995

[5] ISO/IEC 9899:1990/AMD 1:1995, *Amendment 1*, 1995

[6] ISO/IEC 9899:1990/COR 2:1996, *Technical Corrigendum 2*, 1996

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

[8] ISO/IEC 9899:1999/COR 1:2001, *Technical Corrigendum 1*, 2001

[9] ISO/IEC 9899:1999/COR 2:2004, *Technical Corrigendum 2*, 2004

[10] ISO/IEC 9899:1999/COR 3:2007, *Technical Corrigendum 3*, 2007

[11] *Volatiles Are Miscompiled, and What to Do about It*, Eric Eide and John Regehr, University of Utah, School of Computing

Appendix A Deviation permit summary

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

Permit / MISRA / C:2004 / 1.1.A.1 *Code quality (Time behaviour)*

Permit / MISRA / C:2004 / 1.1.B.1 *Code quality (Modifiability)*

Permit / MISRA / C:2004 / 1.1.C.1 *Code quality (Time behaviour)*

Permit / MISRA / C:2004 / 1.1.D.1 *Code quality (Modifiability)*

Permit / MISRA / C:2004 / 1.1.E.1 *Access to hardware*

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

Permit / MISRA / C:2004 / 5.1.A.1 *Adopted code integration*

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

Permit / MISRA / C:2004 / 6.4.A.1 *Code quality (Resource utilization)*

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

Permit / MISRA / C:2004 / 8.5.A.1 *Code quality (Modularity)*

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

Permit / MISRA / C:2004 / 12.7.A.1 *Code quality (Time behaviour)*

Permit / MISRA / C:2004 / 12.7.B.1 *Code quality (Time behaviour)*

Permit / MISRA / C:2004 / 12.7.C.1 *Code quality (Fault tolerance)*

Boolean operations whose results are invariant shall not be permitted.

Permit / MISRA / C:2004 / 13.7.A.1 *Code quality (Reusability)*

Permit / MISRA / C:2004 / 13.7.B.1 *Code quality (Fault tolerance)*

Permit / MISRA / C:2004 / 13.7.C.1 *Adopted code integration*

There shall be no unreachable code.

Permit / MISRA / C:2004 / 14.1.A.1 *Code quality (Reusability)*

Permit / MISRA / C:2004 / 14.1.B.1 *Code quality (Fault tolerance)*

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

Permit / MISRA / C:2004 / 17.1.A.1 *Access to hardware, Code quality (Modifiability)*

Array indexing shall be the only allowed form of pointer arithmetic.

Permit / MISRA / C:2004 / 17.4.A.1 *Access to hardware, Code quality (Modifiability)*