MISRA-C:2004 Coding Standard Manual - Version 3.2

Introduction

This document details the Programming Research enforcement (M2CM) of the rules and guidelines of the *MISRA-C:2004* [23] C programming language coding standard.

The MISRA-C2 Compliance Module (M2CM) is a configuration of the Programming Research C static analyser tool: QAC. This enforcement requires that rules be treated as mandatory and the guidelines are recommended practice. Each rule or guideline is listed with the justification provided in the MISRA-C2 document.

Conventions

Throughout this document, a rule is formatted using the following structure.

Rule	This paragraph describes a rule for C. Adherence is mandatory.
Guideline	This paragraph describes a guideline for C. Adherence is recommended.
Enforcement	Immediately beneath the rule statement this standard quotes the QAC message(s) that indicate violation of the rule, e.g. (QAC 5004).
Justification	This paragraph explains the rationale behind the rule or guideline.
Example	Where applicable an example, normally a code fragment, is given which highlights the rule.

Base Standard

This document is the Programming Research, MISRA-C:2004 Compliance Module (M2CM), enforcement of MISRA-C:2004, described in the "MISRA-C:2004 - Guidelines for the use of the C Language in critical systems" [23] document, published in October 2004 by The Motor Industry Software Reliability Association (MISRA). The Base Standard for this document is the ISO C Standard 9899:1990 [2], amended and corrected by ISO/IEC 9899/COR1:1995 [4], ISO/IEC 9899/AMD1:1995 [5], and ISO/IEC 9899/COR2:1996 [6].

In content the ISO/IEC standard [2] and the ANSI standard [7] are identical. Note that the section numbering is different in the two standards, and this document follows the section numbering of the ISO standard.

Glossary of terms

This section contains definitions of a few of the more important terms used in MISRA-C:2004 (MISRA-C2) terminology.

Boolean Expressions

Strictly speaking, there is no Boolean type in C, but there is a conceptual difference between expressions which return a numeric value and expressions which return a Boolean value. An expression is considered to represent a Boolean value either because it appears in a position where a Boolean value is expected or because it uses an operator that gives rise to a Boolean value.

Boolean values are expected in the following contexts:

- the controlling expression of an if statement
- the controlling expression of an iteration statement
- the first operand of the conditional operator ?

Each of these contexts requires an effectively Boolean expression which is either Boolean-by-construct or Boolean-by-enforcement as defined below.

Boolean-by-construct values are produced by the following operators:

- equality operators (== and !=)
- logical operators (!, && and ||)
- relational operators (<, >, <= and >=)

Boolean-by-enforcement values can be introduced by implementing a specific type enforcement mechanism using a tool. A Boolean type could be associated with a specific typedef, and would then be used for any objects that are Boolean. This could bring many benefits, especially if the checking tool can support it, and in particular it could help avoid confusion between logical operations and integer operations.

Small Integer Type

The term small integer type is used to describe those integer types that are subject to integral promotion. The affected types are char, short, bit-field and enum.

Implicit Conversion

There are three particular categories of implicit type conversion that need to be distinguished.

a) Integral Promotion Conversions: Integral promotion describes a process whereby arithmetic operations are always conducted on integer operands of type int or long (signed or unsigned). Operands of any other integer type, (char, short, bit-field and enum) are always converted to type int or unsigned int before an arithmetic operation. These types are referred to as small integer types.

The rules of integral promotion decree that in most arithmetic operations, an operand of a small integer type be converted to an int if an int is able to represent all values of the original type; otherwise the value is converted to unsigned int.

Notice that integral promotion :

- is only applied to small integer types
- is applied to operands of unary, binary and ternary operators
- is not applied to the operands of the logical operators &&, ||, !
- is applied to the control expression of a switch statement.

Integral promotion is frequently confused with "balancing" of operands (described below). In fact, integral promotion takes place in unary operations and it takes place in binary operations where both operands are of the same type.

Because of integral promotion, the result of adding two objects of type unsigned short is always a value of type signed int or unsigned int; in fact, the addition is performed in that type. It is therefore possible for such an operation to derive a result whose value exceeds the size that could be accommodated in the original type of the operands. For example, if the size of an int is 32 bits, it is possible to multiply two objects of type short (16 bits) and derive a 32-bit result with no danger of overflow. On the other hand if the size of an int is only 16 bits, the product of two 16-bit objects will only yield a 16-bit result and appropriate restrictions must be placed on the size of the operands.

Integral promotion also applies to unary operators. For example, the result of applying a bitwise negation operator (~) to an unsigned char operand is typically a negative value of type signed int.

Integral promotion is a fundamental inconsistency in the C language whereby the small integer types behave differently from long and int types. The use of typedefs is a practice that is encouraged in MISRA-C. However, because the behaviour of the various integer types is not consistent, it can be unsafe to ignore the underlying base types (see decription on following pages) unless some restrictions are placed on the way in which expressions are constructed. It is the intention of the following rules that the effects of integral promotion should be neutralised in order to avoid these anomalies.

b) Assigning Conversions: Assigning conversions occur when:

- The type of an assignment expression is converted to the type of the assignment object.
- The type of an initialiser expression is converted to the type of the initialised object.
- The type of a function call argument is converted to the type of the formal parameter as declared in the function prototype.
- The type of the expression used in a return statement is converted to the type of the function as declared in the function prototype.
- The type of the constant expression in a switch case label is converted to the promoted type of the controlling expression. This conversion is performed only for the purposes of comparison.

In each case, the value of an arithmetic expression is unconditionally converted, where necessary, to another type.

c) Balancing Conversdions: Balancing conversions are described in the ISO C standard under the term "Usual Arithmetic Conversions". This is a set of rules which provides a mechanism to yield a common type when two operands of a binary operator are balanced to a common type or the second and third arguments of the conditional operator (? :) are balanced to a common type. Balancing conversions always involve two operands of different type; one and sometimes both operands will be subject to an implicit conversion.

The balancing rules are complicated by the process of integral promotion (described above) under which any operand of a small integer type is first promoted to type int or unsigned int. Integral promotion happens as part of the usual arithmetic conversions even when two operands are of identical type.

The operators explicitly associated with balancing conversions are:

- Multiplicative *, /, %
- Additive +, -
- Bitwise &, ^, |
- The conditional operator (... ? ... : ...)
- Relational operators >, >=, <, <=
- Equality operators ==, !=

Most of these operators yield a result that is the type resulting from the balancing process. Relational and equality operators are the exception in that they yield a Boolean value result of type int.

Notice that the operands of the bitwise shift operators (<< and >>) are not balanced. The type of the result is the promoted type of the first operand; the second operand may be of any signed or unsigned integer type.

Dangerous Type Conversions

There are a number of potential dangers associated with type conversions that it is necessary to avoid:

a. **Loss of value.** Conversion to a type where the magnitude of the value cannot be represented.
b. **Loss of sign.** Conversion from a signed type to an unsigned type resulting in loss of sign.
c. **Loss of precision.** Conversion from a floating type to an integer type with consequent loss of precision.

The only type conversions that can be guaranteed safe for all data values and all possible conforming implementations are:

1. Conversion of an integral value to a wider type of the same signedness.
2. Conversion of a floating type to a wider floating type.

Of course, in practice, if assumptions are about typical type sizes, it is possible to classify other type conversions as safe. In general, MISRA-C:2004 adopts the principle that it is wise to identify potentially dangerous type conversions by making the conversion explicit.

There are some other dangers in the area of type conversion that also need to be recognised. These are issues that arise from areas of misunderstanding and difficulty in the C language rather than because data values are not preserved.

a) Type widening in integral promotion: The type in which integral expressions are evaluated depends on the type of the operands after any integral promotion. It is always possible to multiply two 8-bit values and access a 16-bit result if the magnitude requires it. It is sometimes, but not always, possible to multiply two 16-bit values and retrieve a 32-bit result. This is a dangerous inconsistency in the C language and in order to avoid confusion it is safer never to rely on the widening type afforded by integral promotion. Consider the following example:

```
uint16_t u16a = 40000;          /* unsigned short / unsigned int ? */
uint16_t u16b = 30000;          /* unsigned short / unsigned int ? */
uint32_t u32x;                  /* unsigned int / unsigned long  ? */

u32x = u16a + u16b;             /* u32x = 70000 or 4464 ?          */
```

The expected result is presumably 70000, but the value assigned to u32x will in practice depend on the implemented size of an int. If the implemented size of an int is 32 bits, the addition will occur in 32-bit signed arithmetic and the correct value will be stored. If the implemented size of an int is only 16 bits, the addition will take place in 16-bit unsigned arithmetic, wraparound will occur and will yield the value 4464 (70000 % 65536). Wraparound in unsigned arithmetic is well defined and may even be intended; but there is potential for confusion.

b) Evaluation type confusion: A similar problem arises from a common misconception among programmers that the type in which a calculation is conducted is influenced in some way by the type to which the result is assigned or converted. For example, in the following code the two 16-bit objects are added together in 16-bit arithmetic (unless promoted to 32-bit int by integral promotion), and the result is converted to type uint32_t on assignment.

```
u32x = u16a + u16b;
```

It is not unusual for programmers to be deceived into thinking that the addition is performed in 32-bit arithmetic - because of the type of u32x.

Confusion of this nature is not confined to integer arithmetic or to implicit conversions. The following examples demonstrate some statements in which the result is well defined but the calculation may not be performed in the type that the programmer assumes.

```
u32a = ( uint32_t )( u16a * u16b );
f64a = u16a / u16b;
f32a = ( float32_t )( u16a / u16b );
f62a = f32a + f32b;
f62a = ( float64_t )( f32a + f32b );
```

c) Change of signedness in arithmetic operations: Integral promotion will often result in two unsigned operands yielding a result of type (signed) int. For example, the addition of two 16-bit unsigned operands will yield a signed 32-bit result if int is 32 bits but an unsigned 16-bit result if int is 16 bits.

d) Change of signedness in bitwise operations: Integral promotion can have some particularly unfortunate repercussions when bitwise operators are applied to small unsigned types. For example a bitwise complement operation on an operand of type unsigned char will generally yield a result of type (signed) int with a negative value. The operand is promoted to type int before the operation and the extra high order bits are set by the complement process. The number of extra bits, if any, is dependent on the size of an int and it is hazardous if the complement operation is followed by a right shift.

Underlying Type

The type of an expression refers to the type of the value obtained when the expression is evaluated. When two items of type long are added, the expression has type long. Most arithmetic operators derive a result whose type is dependent on the type of the operands. On the other hand, there are some operators that yield a Boolean result of type int regardless of the type of the operands. So, for example, when two items of type long are compared with a relational operator the expression has type int.

The term "underlying type" is defined as describing the type that would be obtained from evaluating an expression if it were not for the effects of integral promotion.

When two operands of type int are added the result is of type int and the expression may be said to have type int.

When two operands of type unsigned char are added the result is also (usually) of type int (because of integral promotion), but the underlying type of the expression is defined to be unsigned char.

The term "underlying type" is not known in the C standard or in other texts on the C language but is useful in describing some of the following rules. It describes a hypothetical departure from the C language in which integral promotion does not exist and the usual arithmetic conversions are applied consistently to all integer types. The concept is introduced because the effects of integral promotion are subtle and sometimes dangerous. Integral promotion is an unavoidable feature of the C language, but the intention of these rules is that the effect of integral promotion should be neutralised by taking no advantage of the widening that occurs with small integer operands.

Of course, the C standard does not explicitly define how small integer types would be balanced to a common type in the absence of integral promotion although it does establish the value-preserving principles.

When adding operands of type int, the programmer is obliged to ensure that the result of the operation will not exceed a value that can be represented in type int. If he fails to do so, overflow will occur and the results are undefined. It is the intention of the approach described here that the same principle should apply when small integer operands are added; the programmer should ensure that the result of adding two unsigned chars is representable in an unsigned char, even though integral promotion could give rise to evaluation in a larger type. In other words the limitations of the underlying type of an expression should be observed rather than the actual type.

Underlying Type of an Integer Constant Expression:

One unfortunate aspect of the C language is that it is not possible to define an integer constant with a char or short type. For example the value "5" can be expressed as a literal constant of type int, unsigned int, long or unsigned long by the addition of a suitable suffix; but no suffix is available to create a representation of the value in the various char or short types. This presents a difficulty when attempting to maintain type consistency in expressions. If it is desired to assign a value to an object of type unsigned char, then either an implicit type conversion from an integer type must be tolerated or a cast must be introduced. Many would argue that to use a cast in such circumstances serves only to reduce readability.

The same problem exists when constants are required in initialisers, function arguments or arithmetic expressions. However the problem is largely a philosophical one associated with our aspiration to observe principles of strong typing.

One way of addressing this problem is to imagine that an integer constant, an enumeration constant, a character constant or an integer constant expression has a type appropriate to its magnitude. This objective can be achieved by extending the concept of underlying type to integer constants and imagining that, where possible, the literal constant has been derived by integral promotion from an imaginary constant with a smaller underlying type.

The underlying type of an integer constant expression is therefore defined as follows:

1. If the actual type of the expression is (signed) int, the underlying type is defined to be the smallest signed integer type which is capable of representing its value.
2. If the actual type of the expression is unsigned int, the underlying type is defined to be the smallest unsigned integer type that is capable of representing its value.
3. In all other circumstances, the underlying type of the expression is defined to be the same as its actual type.
4. In a conventional architecture, the underlying type of an integer constant expression will be determined according to its magnitude and signedness as follows:

Unsigned Values

0U	to	255U	8 bit unsigned
256U	to	65535U	16 bit unsigned
65536U	to	4294967295U	32 bit unsigned

Signed Values

-2147483648	to	-32769	32 bit signed
-32768	to	-129	16 bit signed
-128	to	127	8 bit signed
128	to	32767	16 bit signed
32768	to	2147483647	32 bit signed

Notice that underlying type is an artificial concept. It does not in any way influence the type of evaluation that is actually performed. The

concept has been developed simply as a way of defining a safe framework in which to construct arithmetic expressions.

Complex Expressions

The type conversion rules described in the following paragraphs refer in some places to the notion of a "complex expression". The term "complex expression" is defined to mean any expression that is not:

- a constant expression
- an lvalue (i.e. an object)
- the return value of a function

The conversions that may be applied to complex expressions are restricted in order to avoid some of the dangers outlined above. Specifically it is required that a sequence of arithmetic operations in an expression should be conducted in the same type.

The following expressions are complex:

```
s8a + s8b
~u16a
u16a >> 2
foo( 2 ) + u8a
*ppc + 1
++u8a
```

The following expressions are not complex, even though some contain complex sub-expressions:

```
pc[ u8a ]
foo( u8a + u8b )
**ppuc
*( ppc + 1 )
pcbuf[ s16a * 2 ]
```

Evaluation Order

The following notes give some guidance on how dependence on order of evaluation may occur.

Increment Or Decrement Operators: As an example of what can go wrong, consider

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

This will give different results depending on whether b[i] is evaluated before i++ or vice versa. The problem could be avoided by putting the increment operation in a separate statement. The example would then become:

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

Function Arguments: The order of evaluation of function arguments is unspecified.

```
x = func( i++, i );
```

This will give different results depending on which of the function's two parameters is evaluated first.

Function Pointers: If a function is called via a function pointer there shall be no dependence on the order in which function-designator and function arguments are evaluated.

```
p->task_start_fn( p++ );
```

Function Calls: Functions may have additional effects when they are called (e.g. modifying some global data). Dependence on order of evaluation could be avoided by invoking the function prior to the expression that uses it, making use of a temporary variable for the value.

For example

```
x = f( a ) + g( a );
```

could be written as

```
x = f( a );
x += g( a );
```

As an example of what can go wrong, consider an expression to get two values off a stack, subtract the second from the first, and push the result back on the stack:

```
push( pop() - pop() );
```

This will give different results depending on which of the pop() function calls is evaluated first (because pop() has side-effects).

Nested Assignment Statements: Assignments nested within expressions cause additional side effects. The best way to avoid any chance of this leading to a dependence on order of evaluation is to not embed assignments within expressions.

For example, the following is not recommended:

```
x = y = y = z / 3 ;
x = y = y++;
```

Accessing A Volatile: The volatile type qualifier is provided in C to denote objects whose value can change independently of the execution of the program (for example an input register). If an object of volatile qualified type is accessed this may change its value. C compilers will not optimise out reads of a volatile. In addition, as far as a C program is concerned, a read of a volatile has a side-effect (changing the value of the volatile). It will usually be necessary to access volatile data as part of an expression, which then means there may be dependence on order of evaluation. Where possible though it is recommended that volatiles only be accessed in simple assignment statements, such as the following:

```
volatile uint16_t v;
/* . . . */
x = v;
```

The rule addresses the order of evaluation problem with side-effects. Note that there may also be an issue with the number of times a sub-expression is evaluated, which is not covered by this rule. This can be a problem with function invocations where the function is implemented as a macro. For example, consider the following function-like macro and its invocation:

```
#define MAX( a, b ) ( ( ( a ) > ( b ) ) ? ( a ) : ( b ) )
/* . . . */
z = MAX( i++, j );
```

The definition evaluates the first parameter twice if a > b but only once if a <= b. The macro invocation may thus increment i either once or twice, depending on the values of i and j.

It should be noted that magnitude-dependent effects, such as those due to floating-point rounding, are also not addressed by this rule. Although the order in which side-effects occur is undefined, the result of an operation is otherwise well-defined and is controlled by the structure of the expression. In the following example, f1 and f2 are floating-point variables; F3, F4 and F5 denote expressions with floating-point types.

```
f1 = F3 + ( F4 + F5 );
f2 = ( F3 + F4 ) + F5;
```

The addition operations are, or at least appear to be, performed in the order determined by the position of the parentheses, i.e. first F4 is added to F5 then secondly F3 is added to give the value of f1. Provided that F3, F4 and F5 contain no side-effects, their values are independent of the order in which they are evaluated. However, the values assigned to f1 and f2 are not guaranteed to be the same because floating-point rounding following the addition operations will depend on the values being added.

Coding Rules

The following sections comprise the rules of the MISRA-C2 coding standard. These rules are supported by the M2CM Compliance Module provided by Programming Research for the QAC static analyser.

---

1 Environment

Required 1.1 **All code shall conform to ISO/IEC 9899:1990 C programming language, ISO 9899, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2: 1996192**

These guidelines are based on ISO/IEC 9899:1990 [2] amended and corrected by ISO/IEC 9899/COR1:1995 [4], ISO/IEC 9899/AMD1:1995 [5], and ISO/IEC 9899/COR2: 1996 [6]. No claim is made as to their suitability with respect to the ISO 9899:1999 [8] version of the standard. Any reference in this document to 'Standard C' refers to the older ISO/IEC 9899:1990 [2] standard.

It is recognised that it will be necessary to raise deviations (as described in section 4.3.2) to permit certain language extensions, for example to support hardware specific features.

Deviations are required if the environmental limits as specified in ISO/IEC 9899:1990 5.2.4 [2] are exceeded, other than as allowed by Rule 5.1.

PRL Commentary

Enforcement of this rule is implemented as follows:

- *Messages which identify language constraint errors.*
- *Messages which identify constructs which are supported by the ISO:C99 standard but not by the ISO:C90 standard*
- *Messages which identify constructs which are extensions to the ISO C language.*
- *Messages which identify constructs where the ISO:C90 environmental limits are exceeded.*

4 messages are specifically excluded from this enforcement:

Messages 776 and 778 are disabled in accordance with the concessions described in Rule 1.4.

Messages 815 and 816 are disabled because the conformance limits imposed by the ISO standard to attain total portability for include file names are excessively restrictive. See Rule 19.2.

See also Rule 2.1, Rule 5.1

Reference MISRA Guidelines Table 3;IEC 61508 Part 7: Table C.1;

Implemented by QAC messages:

0180 [C99] Use of ll for conversion specifier.
0232 [C] Value of hex escape sequence is not representable in type 'unsigned char'.
0233 [C] Value of octal escape sequence is not representable in type 'unsigned char'.
0240 [E] This file contains the control-M character at the end of a line.
0241 [E] This file contains the control-Z character - was this transferred from a PC?
0244 [C] Value of character constant is not representable in type 'int'.
0246 [E] Binary integer constants are a language extension.
0261 [C] Comment still open at end of included file.
0320 [C99] Declaration within 'for' statement.
0321 [C] Declaration within 'for' statement defines an identifier '%s' which is not an object.
0322 [C] Illegal storage class specifier used in 'for' statement declaration.
0338 [C] Octal or hex escape sequence value is too large for 'unsigned char' or 'wchar_t' type.
0410 [L] Nesting of parentheses exceeds 32 - program does not conform strictly to ISO:C90.
0422 [C] Function call contains fewer arguments than prototype specifies.
0423 [C] Function call contains more arguments than prototype specifies.
0426 [C] Called function has incomplete return type.
0427 [C] Object identifier used as if it were a function or a function pointer identifier.
0429 [C] Function argument is not of arithmetic type.
0430 [C] Function argument is not of compatible 'struct'/'union' type.
0431 [C] Function argument points to a more heavily qualified type.
0432 [C] Function argument is not of compatible pointer type.
0435 [C] The 'struct'/'union' member '%s' does not exist.
0436 [C] Left operand of '.' must be a 'struct' or 'union' object.
0437 [C] Left operand of '->' must be a pointer to a 'struct' or 'union' object.
0446 [C] Operand of ++/-- must have scalar (arithmetic or pointer) type.
0447 [C] Operand of ++/-- must be a modifiable object.
0448 [C] Operand of ++/-- must not be a pointer to an object of unknown size.
0449 [C] Operand of ++/-- must not be a pointer to a function.
0450 [C] An expression of array type cannot be cast.
0451 [C] Subscripting requires a pointer (or array lvalue).
0452 [C] Cannot subscript a pointer to an object of unknown size.
0453 [C] An array subscript must have integral type.
0454 [C] The address-of operator '&' cannot be applied to an object declared with 'register'.
0456 [C] This expression does not have an address - '&' may only be applied to an lvalue or a function designator.
0457 [C] The address-of operator '&' cannot be applied to a bit-field.
0458 [C] Indirection operator '*' requires operand of pointer type.
0466 [C] Unary '+' requires arithmetic operand.
0467 [C] Operand of '!' must have scalar (arithmetic or pointer) type.
0468 [C] Unary '-' requires arithmetic operand.
0469 [C] Bitwise not '~' requires integral operand.
0476 [C] 'sizeof' cannot be applied to a bit-field.
0477 [C] 'sizeof' cannot be applied to a function.
0478 [C] 'sizeof' cannot be applied to an object of unknown size.
0481 [C] Only scalar expressions may be cast to other types.
0482 [C] Expressions may only be cast to 'void' or scalar types.
0483 [C] A pointer to an object of unknown size cannot be the operand of an addition operator.
0484 [C] A pointer to an object of unknown size cannot be the operand of a subtraction operator.
0485 [C] Only integral expressions may be added to pointers.
0486 [C] Only integral expressions and compatible pointers may be subtracted from pointers.

0487 [C] If two pointers are subtracted, they must be pointers that address compatible types.
0493 [C] Type of left operand is not compatible with this operator.
0494 [C] Type of right operand is not compatible with this operator.
0495 [C] Left operand of '%', '<<', '>>', '&', '^' or '|' must have integral type.
0496 [C] Right operand of '%', '<<', '>>', '&', '^' or '|' must have integral type.
0513 [C] Relational operator used to compare pointers to incompatible types.
0514 [C] Relational operator used to compare a pointer with an incompatible operand.
0515 [C] Equality operator used to compare a pointer with an incompatible operand.
0536 [C] First operand of '&&', '||' or '?' must have scalar (arithmetic or pointer) type.
0537 [C] Second operand of '&&' or '||' must have scalar (arithmetic or pointer) type.
0540 [C] 2nd and 3rd operands of conditional operator '?' must have compatible types.
0541 [C] Argument no. %s does not have object type.
0542 [C] Controlling expression must have scalar (arithmetic or pointer) type.
0546 [C] 'enum %s' has unknown content. Use of an enum tag with undefined content is not permitted.
0547 [C] This declaration of tag '%s' conflicts with a previous declaration.
0550 [C] Left operand of '+=' or '-=' is a pointer to an object of unknown size.
0551 [E] Cast may not operate on the left operand of the assignment operator.
0554 [C] 'static %s()' has been declared and called but no definition has been given.
0555 [C] Invalid assignment to object of void type or array type.
0556 [C] Left operand of assignment must be a modifiable object.
0557 [C] Right operand of assignment is not of arithmetic type.
0558 [C] Right operand of '+=' or '-=' must have integral type when left operand is a pointer.
0559 [C] Right operand of '<<=', '>>=', '&=', '|=', '^=' or '%=' must have integral type.
0560 [C] Left operand of '<<=', '>>=', '&=', '|=', '^=' or '%=' must have integral type.
0561 [C] Right operand of assignment is not of compatible 'struct'/'union' type.
0562 [C] Right operand of assignment points to a more heavily qualified type.
0563 [C] Right operand of assignment is not of compatible pointer type.
0564 [C] Left operand of assignment must be an lvalue (it must designate an object).
0565 [C] Left operand of '+=' or '-=' must be of arithmetic or pointer to object type.
0580 [C] Constant is too large to be representable.
0588 [C] Width of bit-field must be an integral constant expression.
0589 [C] Enumeration constant must be an integral constant expression.
0590 [C] Array bound must be an integral constant expression.
0591 [C] A 'case' label must be an integral constant expression.
0601 [E] Function 'main()' is not of type 'int (void)' or 'int (int, char *[])'.
0604 [C99] Declaration appears after statements in a compound statement.
0605 [C] A declaration must declare a tag or an identifier.
0609 [L] More than 12 pointer, array or function declarators modifying a declaration - program does not conform strictly to ISO:C90.
0611 [L] Nesting of 'struct' or 'union' types exceeds 15 - program does not conform strictly to ISO:C90.
0612 [L] Size of object '%s' exceeds 32767 bytes - program does not conform strictly to ISO:C90.
0614 [L] More than 127 block scope identifiers defined within a block - program does not conform strictly to ISO:C90.
0616 [C] Illegal combination of type specifiers or storage class specifiers.
0617 [C99] 'const' qualifier has been duplicated.
0618 [C99] 'volatile' qualifier has been duplicated.
0619 [C] The identifier '%s' has already been defined in the current scope within the ordinary identifier namespace.
0620 [C] Cannot initialize '%s' because it has unknown size.
0621 [C] The struct/union '%s' cannot be initialized because it has unknown size.
0622 [C] The identifier '%s' has been declared both with and without linkage in the same scope.
0627 [C] '%s' has different type to previous declaration in the same scope.
0628 [C] '%s' has different type to previous declaration at wider scope.
0629 [C] More than one definition of '%s' (with internal linkage).
0631 [C] More than one declaration of '%s' (with no linkage).
0633 [E] Empty structures and unions are a language extension.
0635 [E] Bit-fields in this struct/union have been declared with types other than int, signed int or unsigned int.
0638 [C] Duplicate member name '%s' in 'struct' or 'union'.
0639 [L] Number of members in 'struct' or 'union' exceeds 127 - program does not conform strictly to ISO:C90.
0640 [C] '%s' in 'struct' or 'union' type may not have 'void' type.
0641 [C] '%s' in 'struct' or 'union' type may not have function type.

0642 [C] '%s' in 'struct' or 'union' type may not be an array of unknown size.
0643 [C] '%s' in 'struct' or 'union' type may not be a 'struct' or 'union' with unknown content.
0644 [C] Width of bit-field must be no bigger than the width of an 'int'.
0645 [C] A zero width bit-field cannot be given a name.
0646 [C] Enumeration constants must have values representable as 'int's.
0647 [L] Number of enumeration constants exceeds 127 - program does not conform strictly to ISO:C90.
0649 [C] K&R style declaration of parameters is not legal after a function header that includes a parameter list.
0650 [C] Illegal storage class specifier on named function parameter.
0651 [C] Missing type specifiers in function declaration.
0653 [C] Duplicate definition of 'struct', 'union' or 'enum' tag '%s'.
0655 [C] Illegal storage class specifier on unnamed function parameter.
0656 [C] Function return type cannot be function or array type, or an incomplete struct/union (for function definition).
0657 [C] Unnamed parameter specified in function definition.
0659 [C] The identifier '%s' was not given in the parameter list.
0660 [E] Defining an unnamed member in a struct or union. This is a language extension.
0662 [E] Accessing a member of an unnamed struct or union member in this way is a language extension.
0664 [C] Parameter specified with type 'void'.
0665 [C] Two parameters have been declared with the same name '%s'.
0671 [C] Initializer for object of arithmetic type is not of arithmetic type.
0673 [C] Initializer points to a more heavily qualified type.
0674 [C] Initializer for pointer is of incompatible type.
0675 [C] Initializer is not of compatible 'struct'/'union' type.
0677 [C] Array size is negative, or unrepresentable.
0682 [C] Initializer for object of a character type is a string literal.
0683 [C] Initializer for object of a character type is a wide string literal.
0684 [C] Too many initializers.
0685 [C] Initializer for any object with static storage duration must be a constant expression.
0690 [C] String literal contains too many characters to initialize object.
0698 [C] String literal used to initialize an object of incompatible type.
0699 [C] String literal used to initialize a pointer of incompatible type.
0708 [C] No definition found for the label '%s' in this function.
0709 [C] Initialization of locally declared 'extern %s' is illegal.
0715 [L] Nesting of control structures (statements) exceeds 15 - program does not conform strictly to ISO:C90.
0736 [C] 'case' label does not have unique value within this 'switch' statement.
0737 [C] More than one 'default' label found in 'switch' statement.
0738 [C] Controlling expression in a 'switch' statement must have integral type.
0739 [L] Number of 'case' labels exceeds 257 - program does not conform strictly to ISO:C90.
0746 [C] 'return exp;' found in '%s()' whose return type is 'void'.
0747 [C] 'return exp;' found in '%s()' whose return type is qualified 'void'.
0755 [C] 'return' expression is not of arithmetic type.
0756 [C] 'return' expression is not of compatible 'struct'/'union' type.
0757 [C] 'return' expression points to a more heavily qualified type.
0758 [C] 'return' expression is not of compatible pointer type.
0766 [C] 'continue' statement found outside an iteration statement.
0767 [C] 'break' statement found outside a 'switch' or iteration statement.
0768 [C] 'case' or 'default' found outside a 'switch' statement.
0774 [C] 'auto' may not be specified on global declaration of '%s'.
0775 [C] 'register' may not be specified on global declaration of '%s'.
0801 [C] The '##' operator may not be the first token in a macro replacement list.
0802 [C] The '##' operator may not be the last token in a macro replacement list.
0803 [C] The '#' operator may only appear before a macro parameter.
0804 [C] Macro parameter '%s' is not unique.
0810 [L] '#include "%s"' causes nesting to exceed 8 levels - program does not conform strictly to ISO:C90.
0811 [C] The glue operator '##' may only appear in a '#define' preprocessing directive.
0812 [C] Header name token '<text>' found outside '#include' preprocessing directive.
0821 [C] '#include %s' does not identify a header or source file that can be processed.
0828 [L] More than 8 levels of nested conditional inclusion - program does not conform strictly to ISO:C90.
0830 [E] Unrecognized text encountered after a preprocessing directive.
0831 [E] Use of '\\' in this '#include' line is a PC extension - this usage is non-portable.

0834	[C] Function-like macro '%s()' is being redefined as an object-like macro.
0835	[C] Macro '%s' is being redefined with different parameter names.
0844	[C] Macro '%s' is being redefined with a different replacement list.
0845	[C] Object-like macro '%s' is being redefined as a function-like macro.
0850	[C99] Macro argument is empty.
0851	[C] More arguments in macro call than specified in definition.
0852	[C] Unable to find the ')' that marks the end of the macro call.
0856	[C] Fewer arguments in macro call than specified in definition.
0857	[L] Number of macro definitions exceeds 1024 - program does not conform strictly to ISO:C90.
0858	[L] Number of macro parameters exceeds 31 - program does not conform strictly to ISO:C90.
0859	[L] Number of arguments in macro call exceeds 31 - program does not conform strictly to ISO:C90.
0866	[C] The string literal in a '#line' directive cannot be a 'wide string literal'.
0873	[C] Preprocessing token cannot be converted to an actual token.
0875	[L] String literal exceeds 509 characters - program does not conform strictly to ISO:C90.
0877	[C] '#if' and '#elif' expressions may contain only integral constants.
0899	[E] Unrecognized preprocessing directive has been ignored - assumed to be a language extension.
0930	[C99] Trailing comma at the end of an enumerator-list.
0940	[C] Illegal usage of a variably modified type.
0941	[C] A variable length array may not be initialized.
0943	[C] Jump to label '%s' is a jump into the scope of an identifier with variably modified type.
0944	[C] The label '%s' is inside the scope of an identifier with variably modified type.
0945	[C99] WARNING. Operand of sizeof is an expression of variable length array type.
1001	[E] '#include %s' is a VMS extension.
1002	[E] '%s' is not a legal identifier in ISO C.
1003	[E] '#%s' is a language extension for in-line assembler. All statements located between #asm and #endasm will be ignored.
1006	[E] This in-line assembler construct is a language extension. The code has been ignored.
1008	[E] '#%s' is not a legal ISO C preprocessing directive.
1011	[C99] Use of '//' comment.
1012	[E] Use of a C++ reference type ('type &') will be treated as a language extension.
1014	[E] Non-standard type specifier - this will be treated as a language extension.
1015	[E] '%s' is not a legal keyword in ISO C - this will be treated as a language extension.
1018	[C99] Use of LL suffix.
1019	[E] '@ address' is not supported in ISO C - this will be treated as a language extension.
1020	[E] '__typeof__' is not supported in ISO C, and is treated as a language extension.
1021	[E] A statement expression is not supported in ISO C, and is treated as a language extension.
1022	[E] '__alignof__' is not supported in ISO C, and is treated as a language extension.
1023	[C] Using '__alignof__' on function types is illegal.
1024	[C] Using '__alignof__' on incomplete types is illegal.
1025	[C] Using '__alignof__' on bit-fields is illegal.
1026	[E] The indicated @word construct has been ignored.
1027	[C99] Use of type 'long long'.
1028	[E] Use of the sizeof operator in a preprocessing directive is a language extension.
1029	[E] Whitespace encountered between backslash and new-line has been ignored.
1030	[C99] Macro defined with variable argument list.
1031	[C99] Initializer for 'struct', 'union' or array type is not a constant expression.
1033	[C] The identifier __VA_ARGS__ may only be used in the replacement list of a variadic macro.
1034	[E] Macro defined with named variable argument list. This is a language extension.
1035	[E] No macro arguments supplied for variable argument list. This is a language extension.
1036	[E] Comma before ## ignored in expansion of variadic macro. This is a language extension.
1037	[E] Arrays of length zero are a language extension.
1038	[E] The sequence '', ##__VA_ARGS__'' is a language extension.
1041	[E] Empty aggregate initializers are a language extension.
1042	[E] Using I64 or UI64 as an integer constant suffix. This is a language extension.
1043	[E] Defining an anonymous union object. This is a language extension.
1044	[E] Defining an anonymous struct object. This is a language extension.
1045	[E] Use of the #include_next preprocessing directive is a language extension.
1046	[E] Function is being declared with default argument syntax. This is a language extension.
1047	[C] Function is being declared with default argument syntax after a previous call to the function. This is not allowed.

1048 [C] Default argument values are missing for some parameters in this function declaration. This is not allowed.
1051 [C99] A variable length array has been declared.
1052 [C99] A variable length array of unspecified size has been declared.
1053 [C99] Designators have been used in this initialization list.
1054 [C99] A compound literal has been used.
1055 [C99] The keyword 'inline' has been used.
1056 [C99] The keyword '_Bool' has been used.
3236 [C] 'inline' may not be applied to function 'main'.
3237 [C] inline function '%1s' has external linkage and is defining an object, '%2s', with static storage duration.
3238 [C] inline function '%1s' has external linkage and is referring to an object, '%2s', with internal linkage.
3244 [C] 'inline' may only be used in the declaration of a function identifier.
3664 [E] Using a dot operator to access an individual bit is a language extension.

Required 1.2 **No reliance shall be placed on undefined or unspecified behaviour.**

This rule requires that any reliance on undefined and unspecified behaviour, which is not specifically addressed by other rules, shall be avoided. Where a specific behaviour is explicitly covered in another rule, only that specific rule needs to be deviated if required. See ISO/IEC 9899:1990 Appendix G [2] for a complete list of these issues.

Implemented by QAC messages:

0160 [U] Using unsupported conversion specifier number %s.
0161 [U] Unknown length modifier used with 'i' or 'd' conversion specifier, number %s.
0162 [U] Unknown length modifier used with 'o' conversion specifier, number %s.
0163 [U] Unknown length modifier used with 'u' conversion specifier, number %s.
0164 [U] Unknown length modifier used with 'x' conversion specifier, number %s.
0165 [U] Unknown length modifier used with 'X' conversion specifier, number %s.
0166 [U] Unknown length modifier used with 'f' conversion specifier, number %s.
0167 [U] Unknown length modifier used with 'e' conversion specifier, number %s.
0168 [U] Unknown length modifier used with 'E' conversion specifier, number %s.
0169 [U] Unknown length modifier used with 'g' conversion specifier, number %s.
0170 [U] Unknown length modifier used with 'G' conversion specifier, number %s.
0171 [U] Unknown length modifier used with 'c' conversion specifier, number %s.
0172 [U] Unknown length modifier used with '%%' conversion specifier, number %s.
0173 [U] Unknown length modifier used with 's' conversion specifier, number %s.
0174 [U] Unknown length modifier used with 'n' conversion specifier, number %s.
0175 [U] Unknown length modifier used with 'p' conversion specifier, number %s.
0176 [U] Incomplete conversion specifier, number %s.
0177 [U] Field width of format conversion specifier exceeds 509 characters.
0178 [U] Precision of format conversion specifier exceeds 509 characters.
0179 [U] Argument type does not match conversion specifier number %s.
0184 [U] Insufficient arguments to satisfy conversion specifier, number %s.
0185 [U] Call contains more arguments than conversion specifiers.
0186 [U] A call to this function must include at least one argument.
0190 [U] Using unsupported conversion specifier number %s.
0191 [U] Unknown length modifier used with 'd/i/n' conversion specifier, number %s.
0192 [U] Unknown length modifier used with 'o' conversion specifier, number %s.
0193 [U] Unknown length modifier used with 'u' conversion specifier, number %s.
0194 [U] Unknown length modifier used with 'x/X' conversion specifier, number %s.
0195 [U] Unknown length modifier used with 'e/E/f/F/g/G' conversion specifier, number %s.
0196 [U] Unknown length modifier used with 's' conversion specifier, number %s.
0197 [U] Unknown length modifier used with 'p' conversion specifier, number %s.
0198 [U] Unknown length modifier used with '%%' conversion specifier, number %s.
0199 [U] Unknown length modifier used with '[' conversion specifier, number %s.
0200 [U] Unknown length modifier used with 'c' conversion specifier, number %s.
0201 [U] Incomplete conversion specifier, number %s.
0203 [U] Value of character prior to '-' in '[]' is greater than following character.
0204 [U] Field width of format conversion specifier exceeds 509 characters.
0206 [U] Argument type does not match conversion specifier number %s.
0207 [U] 'scanf' expects address of objects being stored into.
0208 [U] Same character occurs in scanset more than once.

0235	[U] Unknown escape sequence.
0275	[U] Floating value is out of range for conversion to destination type.
0301	[u] Cast between a pointer to object and a floating type.
0302	[u] Cast between a pointer to function and a floating type.
0304	[U] The address of an array declared 'register' may not be computed.
0307	[u] Cast between a pointer to object and a pointer to function.
0309	[U] Integral type is not large enough to hold a pointer value.
0337	[U] String literal has undefined value. This may be a result of using '#' on \\.
0400	[U] '%s' is modified more than once between sequence points - evaluation order unspecified.
0401	[U] '%s' may be modified more than once between sequence points - evaluation order unspecified.
0402	[U] '%s' is modified and accessed between sequence points - evaluation order unspecified.
0403	[U] '%s' may be modified and accessed between sequence points - evaluation order unspecified.
0475	[u] Operand of 'sizeof' is an expression designating a bit-field.
0543	[U] 'void' expressions have no value and may not be used in expressions.
0544	[U] The value of an incomplete 'union' may not be used.
0545	[U] The value of an incomplete 'struct' may not be used.
0602	[U] The identifier '%s' is reserved for use by the library.
0623	[U] '%s' has incomplete type and no linkage - this is undefined.
0625	[U] '%s' has been declared with both internal and external linkage - the behaviour is undefined.
0626	[U] '%s' has different type to previous declaration (which is no longer in scope).
0630	[U] More than one definition of '%s' (with external linkage).
0632	[U] Tentative definition of '%s' with internal linkage cannot have unknown size.
0636	[U] There are no named members in this 'struct' or 'union'.
0654	[U] Using 'const' or 'volatile' in a function return type is undefined.
0658	[U] Parameter cannot have 'void' type.
0661	[U] '%s()' may not have a storage class specifier of 'static' when declared at block scope.
0667	[U] '%s' is declared as a typedef and may not be redeclared as an object at an inner scope without an explicit type specifier.
0668	[U] '%s' is declared as a typedef and may not be redeclared as a member of a 'struct' or 'union' without an explicit type specifier.
0672	[U] The initializer for a 'struct', 'union' or array is not enclosed in braces.
0676	[u] Array element is of function type. Arrays cannot be constructed from function types.
0678	[u] Array element is array of unknown size. Arrays cannot be constructed from incomplete types.
0680	[u] Array element is 'void' or an incomplete 'struct' or 'union'. Arrays cannot be constructed from incomplete types.
0706	[U] Label '%s' is not unique within this function.
0745	[U] 'return;' found in '%s()', which has been defined with a non-'void' return type.
0777	[U] External identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.
0779	[U] Identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.
0809	[U] The '#include' preprocessing directive has not been followed by <h-char-sequence> or "s-char-sequence".
0813	[U] Using any of the characters ' " or /* in '#include <%s>' gives undefined behaviour.
0814	[U] Using the characters ' or /* in '#include "%s"' gives undefined behaviour.
0836	[U] Definition of macro named 'defined'.
0837	[U] Use of '#undef' to remove the operator 'defined'.
0848	[U] Attempting to #undef '%s', which is a predefined macro name.
0853	[U] Macro arguments contain a sequence of tokens that has the form of a preprocessing directive.
0854	[U] Attempting to #define '%s', which is a predefined macro name.
0864	[U] '#line' directive specifies line number which is not in the range 1 to 32767.
0865	[U] '#line' directive is badly formed.
0867	[U] '#line' has not been followed by a line number.
0872	[U] Result of '##' operator is not a legal preprocessing token.
0874	[U] Character string literal and wide character string literal are adjacent.
0885	[U] The token 'defined' is generated in the expansion of this macro.
0887	[U] Use of 'defined' must match either 'defined(identifier)' or 'defined identifier'.
0888	[U] 'defined' requires an identifier as an argument.
0914	[U] Source file does not end with a newline character.
0915	[U] Source file ends with a backslash character followed by a newline.
0942	[U] A * can only be used to specify array size within function prototype scope.
1509	'%1s' has external linkage and has multiple definitions.
1510	'%1s' has external linkage and has incompatible declarations.

2800	Constant: Overflow in signed arithmetic operation.
2810	Constant: Dereference of NULL pointer.
2820	Constant: Arithmetic operation on NULL pointer.
2830	Constant: Division by zero.
2840	Constant: Dereference of an invalid pointer value.
3113	[U] 'return' statement includes no expression but function '%s()' is implicitly of type 'int'.
3114	[U] Function '%s()' is implicitly of type 'int' but ends without returning a value.
3239	[U] inline function '%1s' has external linkage, but is not defined within this translation unit.
3311	[u] An earlier jump to this statement will bypass the initialization of local variables.
3312	[u] This goto statement will jump into a previous block and bypass the initialization of local variables.
3319	[U] Function called with number of arguments which differs from number of parameters in definition.
3437	[u] The assert macro has been suppressed to call a function of that name.
3438	[U] #undef'ing the assert macro to call a function of that name causes undefined behaviour.

Required 1.3 **Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.**

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

Reference Unspecified 11;

Required 1.4 **The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.**

The ISO standard requires external identifiers to be distinct in the first 6 characters. However compliance with this severe and unhelpful restriction is considered an unnecessary limitation since most compilers/linkers allow at least 31 character significance (as for internal identifiers).

The compiler/linker must be checked to establish this behaviour. If the compiler/linker is not capable of meeting this limit, then use the limit of the compiler.

Reference Undefined 7;Implementation 5-6;

Advisory 1.5 **Floating-point implementations should comply with a defined floating-point standard.**

Floating-point arithmetic has a range of problems associated with it. Some (but not all) of the problems can be overcome by using an implementation that conforms to a recognised standard. An example of an appropriate standard is ANSI/IEEE Std 754 [21].

The definition of the floating-point types, in accordance with Guideline 6.3, provides an opportunity for noting the floating-point standard in use, for example:

```
/* IEEE 754 single-precision floating-point */
typedef float float32_t;
```

See also Guideline 6.3

2 Language Extensions

Required 2.1 **Assembly language shall be encapsulated and isolated.**

Where assembly language instructions are required it is recommended that they be encapsulated and isolated in either (a) assembler functions, (b) C functions or (c) macros. For reasons of efficiency it is sometimes necessary to embed simple assembly language instructions in-line, for example to enable and disable interrupts. If it is necessary to do this for any reason, then it is recommended that it be achieved by using macros.

Note that the use of in-line assembly language is an extension to standard C, and therefore also requires a deviation against Rule 1.1.

```
#define NOP asm( "    NOP" )
```

See also Rule 1.1

Reference Unspecified 11;

Implemented by QAC messages:

3006 This function contains a mixture of in-line assembler statements and C statements.

Required 2.2 **Source code shall only use C-style comments.**

This excludes the use of // C99 style comments and C++ style comments, since these are not permitted in C90. Many compilers support the // style of comments as an extension to c90. The use of // in preprocessor directives (e.g. #define) can vary. Also the mixing of /* ... */ and // is not consistent. This is more than a style issue, since different (pre C99) compilers may behave differently.

Implemented by QAC messages:

1011 [C99] Use of '//' comment.

Required 2.3 **The character sequence /* shall not be used within a comment.**

C does not support the nesting of comments even though some compilers support this as a language extension. A comment begins with /* and continues until the first */ is encountered. Any /* occurring inside a comment is a violation of this rule. Consider the following code fragment:

```
/* some comment, end comment marker accidentally omitted

<<New Page>>
Perform_Critical_Safety_Function( X );
/* this comment is not compliant */
```

In reviewing the page containing the call to the function, the assumption is that it is executed code.

Because of the accidental omission of the end comment marker, the call to the safety critical function will not be executed.

Implemented by QAC messages:

3108 Nested comments are not recognized in the ISO standard.

Advisory 2.4 **Sections of code should not be 'commented out'.**

Where it is required for sections of source code not to be compiled then this should be achieved by use of conditional compilation (e.g. #if or #ifdef constructs with a comment). Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.

---

3 Documentation

Required 3.1 **All usage of implementation-defined behaviour shall be documented.**

This rule requires that any reliance on implementation-defined behaviour, which is not specifically addressed by other rules, shall be documented, for example by reference to compiler documentation. Where a specific behaviour is explicitly covered in another rule, only that specific rule needs to be deviated if required. See ISO/IEC 9899:1990 Appendix G [2] for a complete list of these issues.

Implemented by QAC messages:

0202 [I] '-' character in '[]' conversion specification is implementation defined.
0284 [I] Multiple character constants have implementation defined values.
0285 [I] Character constant contains character which is not a member of the basic source character set.
0286 [I] String literal contains character which is not a member of the basic source character set.
0287 [I] Header name contains character which is not a member of the basic source character set.
0288 [I] Source file '%s' has comments containing characters which are not members of the basic source character set.
0289 [I] Source file '%s' has preprocessing tokens containing characters which are not members of the basic source character set.
0292 [I] Source file '%s' has comments containing one of the characters '$', '@' or '`'.
0299 [I] Source file '%s' includes #pragma directives containing characters which are not members of the basic source character set.
0303 [I] Cast between a pointer to volatile object and an integral type.
0305 [I] Cast between a pointer to function and an integral type.
0306 [I] Cast between a pointer to object and an integral type.
0308 Non-portable cast involving pointer to an incomplete type.
0581 [I] Floating-point constant may be too small to be representable.
0634 [I] Bit-fields in this struct/union have not been declared explicitly as unsigned or signed.

0878 Using wide character or string literals.
2850 Constant: Implicit conversion to a signed integer type of insufficient size.
2851 Definite: Implicit conversion to a signed integer type of insufficient size.
2852 Apparent: Implicit conversion to a signed integer type of insufficient size.
2853 Suspicious: Implicit conversion to a signed integer type of insufficient size.
2855 Constant: Casting to a signed integer type of insufficient size.
2856 Definite: Casting to a signed integer type of insufficient size.
2857 Apparent: Casting to a signed integer type of insufficient size.
2860 Constant: Implementation-defined value resulting from left shift operation on expression of signed type.
2861 Definite: Implementation-defined value resulting from left shift operation on expression of signed type.
2862 Apparent: Implementation-defined value resulting from left shift operation on expression of signed type.
2890 Constant: Negative value implicitly converted to an unsigned type.
2895 Constant: Negative value cast to an unsigned type.

Required 3.2 **The character set and the corresponding encoding shall be documented.**

For example, ISO 10646-1 [22] defines an international standard for mapping character sets to numeric values. For portability, "character-constants" and "string-literals" should only contain characters that map to a documented subset. The source code is written in one, or more character sets. Optionally, the program can execute in a second or multiple character sets. All the source and execution character sets shall be documented.

Advisory 3.3 **The implementation of integer division in the chosen compiler should be determined, documented and taken into account.**

Potentially an ISO compliant compiler can do one of two things when dividing two signed integers, one of which is positive and one negative. Firstly it may round up, with a negative remainder (e.g. -5/3 = -1 remainder -2), or secondly it may round down with a positive remainder (e.g. -5/3 = -2 remainder +1). It is important to determine which of these is implemented by the compiler and to document it for programmers, especially if it is the second (perhaps less intuitive) implementation.

Reference Implementation 18;

Required 3.4 **All uses of the #pragma directive shall be documented and explained.**

This rule places a requirement on the user of this standard to produce a list of any pragmas they choose to use in an application. The meaning of each pragma shall be documented. There shall be sufficient supporting description to demonstrate that the behaviour of the pragma, and its implications for the application, have been fully understood.

Any use of pragmas should be minimised, localised and encapsulated within dedicated functions wherever possible.

Reference Implementation 40;

Implemented by QAC messages:

3116 Unrecognized #pragma arguments '%s' This #pragma directive has been ignored.

Required 3.5 **If it is being relied upon, the implementation-defined behaviour and packing of bitfields shall be documented.**

This is a particular problem where bit fields are used because of the poorly defined aspects of bit fields described under Rules 6.4 and 6.5. The 'bit field' facility in C is one of the most poorly defined parts of the language. There are two main uses to which bit fields could be put:

1. To access the individual bits, or groups of bits, in larger data types (in conjunction with unions). This use is not permitted (see Rule 18.4).

2. To allow flags or other short-length data to be packed to save storage space.

The packing together of short-length data to economise on storage is the only acceptable use of bit fields envisaged in this standard. Provided the elements of the structure are only ever accessed by their name, the programmer needs to make no assumptions about the way that the bit fields are stored within the structure.

It is recommended that structures be declared specifically to hold the sets of bit fields, and do not include any other data within the same structure. Note that Guideline 6.3 need not be followed in defining bit-fields, since their lengths are specified in the structure.

If the compiler has a switch to force bit fields to follow a particular layout then this could assist in such a justification.

For example the following is acceptable:

```
struct message                /* Struct is for bit-fields only */
{
```

```
    signed   int  little: 4;  /* Note: use of basic types is required */
    unsigned int  x_set:  1;
    unsigned int  y_set:  1;
} message_chunk;
```

If using bit fields, be aware of the potential pitfalls and areas of implementation-defined (i.e. non-portable) behaviour. In particular the programmer should be aware of the following:

* The alignment of the bit fields in the storage unit is implementation-defined, that is whether they are allocated from the high end or low end of the storage unit (usually a byte).

* Whether or not a bit field can overlap a storage unit boundary is also implementation-defined (e.g. if a 6-bit field and a 4-bit field are declared in that order, whether the 4 bit field will start a new byte or whether it will be 2 bits in one byte and 2 bits in the next).

See also Guideline 6.3, Rule 6.4, Rule 6.5, Rule 18.4

Reference Unspecified 10;Implementation 30-31;

Required 3.6

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

This rule refers to any libraries used in the production code, which therefore may include standard libraries supplied with the compiler, other third-party libraries, or libraries designed in-house. This is recommended by IEC 61508 Part 3.

Reference IEC 61508 Part 3;

---

4 Character Sets

Required 4.1

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

Only "simple-escape-sequences" in ISO/IEC 9899:1990 [3-6] Section 6.1.3.4 and \0 are permitted escape sequences.
All "hexadecimal-escape-sequences" are prohibited.
The "octal-escape-sequences" other than \0 are also prohibited under Rule 7.1.

Reference Undefined 11;Implementation 11;

Implemented by QAC messages:

0235	[U] Unknown escape sequence.
3610	Hexadecimal escape sequence used.

Required 4.2

Trigraphs shall not be used.

Trigraphs are denoted by a sequence of 2 question marks followed by a specified third character (e.g. ??- represents a '~' (tilde) character and ??) represents a ']'). They can cause accidental confusion with other uses of two question marks. For example the string

```
"(Date should be in the form ??-??-??)"
```

```
- would not behave as expected, actually being interpreted by the compiler as
"(Date should be in the form ~~]"
```

Implemented by QAC messages:

3601	Trigraphs (??x) are an ISO feature.

---

5 Identifiers

Required 5.1

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

The ISO standard requires internal identifiers to be distinct in the first 31 characters to guarantee code portability. This limitation shall not be exceeded, even if the compiler supports it. This rule shall apply across all name spaces. Macro names are also included and the 31 character limit applies before and after substitution.

The ISO standard requires external identifiers to be distinct in the first 6 characters, regardless of case, to guarantee optimal portability. However this limitation is particularly severe and is considered unnecessary. The intent of this rule is to sanction a relaxation of the ISO requirement to a degree commensurate with modern environments and it shall be confirmed that 31 character/case significance is supported by the implementation.

Note that there is a related issue with using identifier names that differ by only one or a few characters, especially if the identifier names are long. The problem is heightened if the differences are in easily mis-read characters like 1 (one) and l (lower case L), 0 and O, 2 and Z, 5 and S, or n and h. It is recommended to ensure that identifier names are always easily visually distinguishable. Specific guidelines on this issue could be placed in the style guidelines.

See also Rule 1.1

Reference Undefined 7;Implementation 5-6;

Implemented by QAC messages:

0777 [U] External identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.

0779 [U] Identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.

Required 5.2 **Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.**

The terms outer and inner scope are defined as follows. Identifiers that have file scope can be considered as having the outermost scope. Identifiers that have block scope have a more inner scope. Successive, nested blocks, introduce more inner scopes. This rule disallows the case where a second inner definition hides an outer definition. If the second definition does not hide the first definition, then this rule is not violated.

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

```
int16_t i;
{
    int16_t i;        /* This is a different variable                    */
         /* This is not compliant                          */
    i = 3;            /* It could be confusing as to which i this refers */
}
```

See also Guideline 5.7

Implemented by QAC messages:

2547 This declaration of tag '%s' hides a more global declaration.

3334 This declaration of '%s' hides a more global declaration.

Required 5.3 **A typedef name shall be a unique identifier.**

No typedef name shall be reused either as a typedef name or for any other purpose.

For example:

```
{
    typedef unsigned char uint8_t;
}
{
    typedef unsigned char uint8_t;   /* Not compliant - redefinition */
}
{
    unsigned char uint8_t;           /* Not compliant - reuse of uint8_t */
}
```

typedef names shall not be reused anywhere within a program. The same typedef shall not be duplicated anywhere in the source code files even if the declarations are identical. Where the type definition is made in a header file, and that header file is included in multiple source files, this rule is not violated.

See also Guideline 5.7

Implemented by QAC messages:

1506 The identifier '%1s' is declared as a typedef and is used elsewhere for a different kind of declaration.

1507 '%1s' is used as a typedef for different types.

1508 The typedef '%1s' is declared in more than one location.

3448 Declaration of typedef '%s' is not in a header file although it is used in a definition or declaration with external linkage.

Required 5.4 **A tag name shall be a unique identifier.**

No tag name shall be reused either to define a different tag or for any other purpose within the program. ISO/IEC 9899:1990 [2] does not define the behaviour when an aggregate declaration uses a tag in different forms of type specifier (struct or union). Either all uses of the tag should be in structure type specifiers, or all uses should be in union type specifiers, For example:

```
struct stag { uint16_t a; uint16_t b; };

struct stag a1 = { 0, 0 };      /* Compliant - compatible with above   */
union stag  a2 = { 0, 0 };      /* Not compliant - not compatible with
                                   previous declarations               */

void foo( void )
{
    struct stag { uint16_t a; }; /* Not compliant - tag stag redefined  */
}
```

The same tag definition shall not be duplicated anywhere in the source code files even if the definitions are identical. Where the tag definition is made in a header file, and that header file is included in multiple source files, this rule is not violated.

See also Guideline 5.7

Implemented by QAC messages:

0547 [C] This declaration of tag '%s' conflicts with a previous declaration.

Advisory 5.5 — No object or function identifier with static storage duration should be reused.

Regardless of scope, no identifier with static storage duration should be re-used across any source files in the system. This includes objects or functions with external linkage and any objects or functions with the static storage class specifier.

While the compiler can understand this and is in no way confused, the possibility exists for the user to incorrectly associate unrelated variables with the same name. One example of this confusion is having an identifier name with internal linkage in one file and the same identifier name with external linkage in another file.

See also Guideline 5.7

Implemented by QAC messages:

1525 Object/function with external linkage has same identifier as another object/function with internal linkage.
1526 Object with no linkage has same identifier as another object/function with external linkage.
1527 Object/function with internal linkage has same identifier as another object/function with internal linkage.
1528 Object with no linkage has same identifier as another object/function with internal linkage.
1529 Object with no linkage has same identifier as another object with static storage duration but no linkage.

Advisory 5.6 — No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.

Name space and scope are different. This rule is not concerned with scope. For example, ISO C allows the same identifier (vector) for both a tag and a typedef at the same scope.

```
typedef struct vector { uint16_t x ; uint16_t y; uint16_t z; } vector;
/* Rule violation ^^                                            ^^ */
```

ISO C defines a number of different name spaces (see ISO/IEC 9899:1990 6.1.2.3 [2]). It is technically possible to use the same name in separate name spaces to represent completely different items. However this practice is deprecated because of the confusion it can cause, and so names should not be reused, even in separate name spaces.

The example below illustrates a violation of this rule in which value is inadvertently used instead of record.value:

```
struct { int16_t key; int16_t value; } record;

int16_t value;                /* Rule violation - 2nd use of value */

record.key = 1;
value = 0;                    /* should have been record.value */
```

By contrast, the example below does not violate the rule because two member names are less likely to be confused:

```
struct device_q { struct device_q *next;  /* ... */ } devices[N_DEVICES];
struct task_q { struct task_q *next;  /* ... */ }
tasks[N_TASKS];

devices[0].next = &devices[1];
tasks[0].next   = &tasks[1];
```

See also Guideline 5.7

Implemented by QAC messages:

0780 Another identifier '%s' is already in scope in a different namespace.

0781 '%s' is being used as a structure/union member as well as being a label, tag or ordinary identifier.

Advisory 5.7 — No identifier name should be reused.

Regardless of scope, no identifier should be re-used across any files in the system. This rule incorporates the provisions of 5.2, 5.3, 5.4, 5.5, and 5.6.

```
struct air_speed
{
    uint16_t speed; /* knots */
} * x;
struct gnd_speed
{
    uint16_t speed; /* mph                                          */
                    /* Not Compliant - speed is in different units */
} * y;
x->speed = y->speed;
```

Where an identifier name is used in a header file, and that header file is included in multiple source files, this rule is not violated. The use of a rigorous naming convention can support the implementation of this rule.

See also Rule 5.2, Rule 5.3, Rule 5.4, Guideline 5.5, Guideline 5.6

---

6 Types

Required 6.1 — The plain char type shall be used only for the storage and use of character values.

There are three distinct char types, (plain) char, signed char and unsigned char. signed char and unsigned char shall be used for numeric data and plain char shall be used for character data. The signedness of the plain char type is implementation-defined and should not be relied upon.

The only permissible operators on plain char types are assignment and equality operators (=, ==, !=).

Reference Implementation 14;

Implemented by QAC messages:

1810 An operand of 'essentially character' type is being added to another operand of 'essentially character' type.

1811 An operand of 'essentially character' type is being subtracted from an operand of 'essentially signed' type.

1812 An operand of 'essentially character' type is being subtracted from an operand of 'essentially unsigned' type.

1813 An operand of 'essentially character' type is being balanced with an operand of 'essentially floating' type in this arithmetic operation.

4401 An expression of 'essentially Boolean' type (%1s) is being converted to character type, '%2s' on assignment.

4421 An expression of 'essentially enum' type (%1s) is being converted to character type, '%2s' on assignment.

4431 An expression of 'essentially signed' type (%1s) is being converted to character type, '%2s' on assignment.

4441 An expression of 'essentially unsigned' type (%1s) is being converted to character type, '%2s' on assignment.

4451 An expression of 'essentially floating' type (%1s) is being converted to character type, '%2s' on assignment.

4510 An expression of 'essentially character' type (%1s) is being used as an array subscript.

4511 An expression of 'essentially character' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).

4512 An expression of 'essentially character' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).

4513 An expression of 'essentially character' type (%1s) is being used as the left-hand operand of this shift operator (%2s).

4514 An expression of 'essentially character' type (%1s) is being used as the right-hand operand of this shift operator (%2s).

4517 An expression of 'essentially character' type (%1s) is being used as the operand of this increment/decrement operator (%2s).

4518 An expression of 'essentially character' type (%1s) is being used as the %2s operand of this logical operator (%3s).

4519 An expression of 'essentially character' type (%1s) is being used as the first operand of this conditional operator (%2s).

Required 6.2 — Signed and unsigned char type shall be used only for the storage and use of numeric values.

There are three distinct char types, (plain) char, signed char and unsigned char. Signed char and unsigned char shall be used for numeric data and plain char shall be used for character data. The signedness of the plain char type is implementation-defined and should not be relied upon.
Character values/data are character constants or string literals such as 'A', '5', '\n', "a".
Numeric values/data are numbers such as 0, 5, 23, \x10, -3.

Character sets map text characters onto numeric values. Character values are the "text".
The permissible operators on plain char types are the simple assignment operator (=), equality operators (==, !=) and explicit casts to integral types. Additionally, the second and third operands of the ternary conditional operator may both be of plain char type.

Implemented by QAC messages:

4413 An expression of 'essentially character' type (%1s) is being converted to signed type, '%2s' on assignment.
4414 An expression of 'essentially character' type (%1s) is being converted to unsigned type, '%2s' on assignment.

Advisory 6.3 **Typedefs that indicate size and signedness should be used in place of the basic numerical types.**

The basic numerical types of signed and unsigned variants of char, int, short, long, and float, double should not be used, but specific-length typedefs should be used. Rule 6.3 helps to clarify the size of the storage, but does not guarantee portability because of the asymmetric behaviour of integral promotion. See discussion of integral promotion - section 6.10. It is still important to understand the integer size of the implementation.

Programmers should be aware of the actual implementation of the typedefs under these definitions.

For example, the ISO (POSIX) typedefs as shown below are recommended and are used for all basic numerical and character types in this standard. For a 32-bit integer machine, these are as follows:

```
typedef          char   char_t;
typedef signed   char   int8_t;
typedef signed   short  int16_t;
typedef signed   int    int32_t;
typedef signed   long   int64_t;
typedef unsigned char   uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned int    uint32_t;
typedef unsigned long   uint64_t;
typedef          float  float32_t;
typedef          double float64_t;
typedef long     double float128_t;
```

typedefs are not considered necessary in the specification of bit-field types.

See also Guideline 1.5, Rule 3.5

Implemented by QAC messages:

5013 Use of basic type '%s'.

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

Using int is implementation-defined because bitfields of type int can be either signed or unsigned. The use of enum, short or char types for bit fields is not allowed because the behaviour is undefined.

See also Rule 3.5

Reference Undefined 38;Implementation 29;

Implemented by QAC messages:

0634 [I] Bit-fields in this struct/union have not been declared explicitly as unsigned or signed.
0635 [E] Bit-fields in this struct/union have been declared with types other than int, signed int or unsigned int.

Required 6.5 **Bit fields of signed type shall be at least 2 bits long.**

A signed bit field of 1 bit length is not useful.

See also Rule 3.5

Implemented by QAC messages:

3659 Unnamed zero-width bit-field declared with a signed type.
3660 Named bit-field consisting of a single bit declared with a signed type.
3665 Unnamed bit-field consisting of a single bit declared with a signed type.

---

7 Constants

Required 7.1 **Octal constants (other than zero) and octal escape sequences shall not be used.**

Any integer constant beginning with a '0' (zero) is treated as octal. So there is a danger, for example, with writing fixed length constants. For example, the following array initialisation for 3-digit bus messages would not do as expected (052 is octal, i.e. 42 decimal):

```
code[1] = 109;      /* equivalent to decimal 109 */
code[2] = 100;      /* equivalent to decimal 100 */
code[3] = 052;      /* equivalent to decimal 42 */
code[4] = 071;      /* equivalent to decimal 57 */
```

Octal escape sequences can be problematic because the inadvertent introduction of a decimal digit ends the octal escape and introduces another character. The value of the first expression in the following example is implementation-defined because the character constant consists of two characters, '\10' and '9'. The second character constant expression below contains the single character '\100'. Its value will be implementation-defined if character 64 is not represented in the basic execution character set.

```
code[5] = '\109';   /* implementation-defined, two character constant */
code[6] = '\100';   /* set to 64, or implementation-defined           */
```

It is better not to use octal constants or escape sequences at all, and to statically check for any occurrences. The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule. Additionally "\0" is the only permitted octal escape sequence.

Reference Koenig 9;

Implemented by QAC messages:

0336	Macro defined as an octal constant.
0339	Octal constant used.
3628	Octal escape sequences used in a character constant or string literal.

---

8 Declarations and Definitions

Required 8.1 **Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.**

The use of prototypes enables the compiler to check the integrity of function definitions and calls. Without prototypes the compiler is not obliged to pick up certain errors in function calls (e.g. different number of arguments from the function body, mismatch in types of arguments between call and definition). Function interfaces have been shown to be a cause of considerable problems, and therefore this rule is considered very important.

The recommended method of implementing function prototypes for external functions is to declare the function (i.e. give the function prototype) in a header file, and then include the header file in all those code files that need the prototype (see Rule 8.8).

The provision of a prototype for a function with internal linkage is a good programming practice.

See also Rule 8.8, Rule 16.6

Reference Undefined 22,23;

Implemented by QAC messages:

3002	Defining '%s()' with an identifier list and separate parameter declarations is an obsolescent feature.
3335	No function declaration. Implicit declaration inserted: 'extern int %s();'.
3450	Function '%s', with internal linkage, is being defined without a previous declaration.

Required 8.2 **Whenever an object or function is declared or defined, its type shall be explicitly stated.**

```
extern         x;           /* Non-compliant - implicit int type */
extern int16_t x;           /* Compliant - explicit type         */
const          y;           /* Non-compliant - implicit int type */
const  int16_t y;           /* Compliant - explicit type         */
static         foo( void ); /* Non-compliant - implicit type     */
static int16_t foo( void ); /* Compliant - explicit type         */
```

See also Rule 16.5

Implemented by QAC messages:

2050	The 'int' type specifier has been omitted from a function declaration.
2051	The 'int' type specifier has been omitted from an object declaration.

Required 8.3

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

The types of the parameters and return values in the prototype and the definition must match. This requires identical types including typedef names and qualifiers, and not just identical base types.

Reference

Undefined 24;Koenig 59-62;

Implemented by QAC messages:

0624	Function '%s' is declared using typedefs which are different to those in a previous declaration.
1331	Type or number of arguments doesn't match previous use of the function.
1332	Type or number of arguments doesn't match prototype found later.
1333	Type or number of arguments doesn't match function definition found later.
3320	Type of argument no. %s differs from its type in definition of function.
3675	Function parameter declared with type qualification which differs from previous declaration.

Required 8.4

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

The definition of compatible types is lengthy and complex (ISO/IEC 9899:1990 [2], sections 6.1.2.6, 6.5.2, 6.5.3 and 6.5.4 give full details). Two identical types are compatible but two compatible types need not be identical. For example, the following pairs of types are compatible:

```
signed int              int
char [5]                char []
unsigned short int      unsigned short
```

Reference

Undefined 10;

Implemented by QAC messages:

0626	[U] '%s' has different type to previous declaration (which is no longer in scope).
0627	[C] '%s' has different type to previous declaration in the same scope.
0628	[C] '%s' has different type to previous declaration at wider scope.
1510	'%1s' has external linkage and has incompatible declarations.

Required 8.5

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

Header files should be used to declare objects, functions, typedefs, and macros. Header files shall not contain or produce definitions of objects or functions (or fragments of functions or objects) that occupy storage. This makes it clear that only C files contain executable source code and that header files only contain declarations. A "header file" is defined as any file that is included via the #include directive, regardless of name or suffix.

Implemented by QAC messages:

3406	Object/function '%s', with external linkage, has been defined in a header file.
3480	Object/function '%s', with internal linkage, has been defined in a header file.

Required 8.6

Functions shall be declared at file scope.

Declaring functions at block scope may be confusing, and can lead to undefined behaviour.

Reference

Undefined 36;

Implemented by QAC messages:

3221	Function with external linkage declared at block scope.

Required 8.7

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

The scope of objects shall be restricted to functions where possible. File scope shall only be used where objects need to have either internal or external linkage. Where objects are declared at file scope Rule 8.10 applies. It is considered good practice to avoid making identifiers global except where necessary. Whether objects are declared at the outermost or innermost block is largely a matter of style. "Accessing" means using the identifier to read from, write to, or take the address of the object.

See also

Rule 8.10

Implemented by QAC messages:

1514	The object '%1s' is only referenced by function '%2s', in the translation unit where it is defined
3218	File scope static, '%s', is only accessed in one function.

Required 8.8 **An external object or function shall be declared in one and only one file.**

Normally this will mean declaring an external identifier in a header file, that will be included in any file where the identifier is defined or used. For example:

```
extern int16_t a;
```

in featureX.h, then to define a:

```
#include <featureX.h>
int16_t a = 0;
```

There may be one or there may be many header files in a project, but each external object or function shall only be declared in one header file.

See also Rule 8.1

Reference Koenig 66;

Implemented by QAC messages:

1513	Identifier '%1s' with external linkage has separate non-defining declarations in more than one location.
3222	Object with external linkage declared at block scope.
3408	'%s' has external linkage and is being defined without any previous declaration.
3447	'%s' is being declared with external linkage but this declaration is not in a header file.
3451	The global identifier '%s' has been declared in more than one file.

Required 8.9 **An identifier with external linkage shall have exactly one external definition.**

Behaviour is undefined if an identifier is used for which multiple definitions exist (in different files) or no definition exists at all. Multiple definitions in different files are not permitted even if the definitions are the same, and it is obviously serious if they are different, or initialise the identifier to different values.

Reference Undefined 44;Koenig 55,63-65;

Implemented by QAC messages:

0630	[U] More than one definition of '%s' (with external linkage).
1509	'%1s' has external linkage and has multiple definitions.

Required 8.10 **All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.**

If a variable is only to be used by functions within the same file then use static. Similarly if a function is only called from elsewhere within the same file, use static. Use of the static storage-class specifier will ensure that the identifier is only visible in the file in which it is declared and avoids any possibility of confusion with an identical identifier in another file or a library.

See also Rule 8.7

Reference Koenig 56-57;

Implemented by QAC messages:

1504	The object '%1s' is only referenced in the translation unit where it is defined.
1505	The function '%1s' is only referenced in the translation unit where it is defined.

Required 8.11 **The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.**

The static and extern storage class specifiers can be a source of confusion. It is good practice to apply the static keyword consistently to all declarations of objects and functions with internal linkage.

Implemented by QAC messages:

3224	This identifier has previously been declared with internal linkage but is not declared here with the static storage class specifier.

Required 8.12 **When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.**

```
int array1[ 10 ];                       /* Compliant     */
```

```
extern int array2[ ];                    /* Not compliant */
int array2[ ] = { 0, 10, 15 };           /* Compliant     */
```

Although it is possible to declare an array of incomplete type and access its elements, it is safer to do so when the size of the array may be explicitly determined.

Implemented by QAC messages:

3684 Array declared with unknown size.

---

9 Initialisation

Required 9.1 **All automatic variables shall have been assigned a value before being used.**

The intent of this rule is that all variables shall have been written to before they are read. This does not necessarily require initialisation at declaration.

Note that according to the ISO C standard, variables with static storage duration are automatically initialised to zero by default, unless explicitly initialised. In practice, many embedded environments do not implement this behaviour. Static storage duration is a property of all variables declared with the static storage class specifier, or with external linkage. Variables with automatic storage duration are not usually automatically initialised.

Reference

Undefined 41;

Implemented by QAC messages:

2883 This 'goto' statement will always bypass the initialization of local variables.
2961 Definite: Using value of uninitialized automatic object '%s'.
2962 Apparent: Using value of uninitialized automatic object '%s'.
2971 Definite: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.
2972 Apparent: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.

Required 9.2 **Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.**

ISO C requires initialiser lists for arrays, structures and union types to be enclosed in a single pair of braces (though the behaviour if this is not done is undefined). The rule given here goes further in requiring the use of additional braces to indicate nested structures. This forces the programmer to explicitly consider and demonstrate the order in which elements of complex data types are initialised (e.g. multi-dimensional arrays).

For example, below are two valid (in ISO C) ways of initialising the elements of a two dimensional array, but the first does not adhere to the rule:

```
int16_t y[3][2] = { 1, 2, 3, 4, 5, 6 };             /* not compliant */
int16_t y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };  /* compliant     */
```

A similar principle applies to structures, and nested combinations of structures, arrays and other types.

Note also that all the elements of arrays or structures can be initialised (to zero or NULL) by giving an explicit initialiser for the first element only. If this method of initialisation is chosen then the first element should be initialised to zero (or NULL), and nested braces need not be used.

The ISO standard [2] contains extensive examples of initialisation.

The intent of Rule 9.2 is that the non-zero initialisation of arrays and structures shall require an explicit initialiser for each element, e.g.

```
int16_t arraya1[5] = { 1, 2, 3, 0, 0 };
/* Compliant - non-zero initialisation */
int16_t arraya2[5] = { 0 };
  /* Compliant - zero initialisation*/
int16_t arraya3[5] = { 1, 2, 3 };
  /* Not Compliant - non-zero initialisation */
int16_t arraya4[2][2] = { 0 };
/* Compliant - zero initialisation at top-level */
int16_t arraya5[2][2] = { { 0 }, { 1, 2 }};
/* Not Compliant - zero initialisation at sub-level */
```

Zero or NULL initialisation shall only be applied at the top level of the array or structure.

Reference

Undefined 42;

Implemented by QAC messages:

0686 Array has fewer initializers than its declared size. Default initialization is applied to the remainder of the array elements.
0693 Struct initializer is missing the optional {.
0694 Array initializer is missing the optional {.

Required 9.3

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

If an enumerator list is given with no explicit initialisation of members, then C allocates a sequence of integers starting at 0 for the first element and increasing by 1 for each subsequent element.

An explicit initialisation of the first element, as permitted by the above rule, forces the allocation of integers to start at the given value. When adopting this approach it is essential to ensure that the initialisation value used is small enough that no subsequent value in the list will exceed the int storage used by enumeration constants.

Explicit initialisation of all items in the list, which is also permissible, prevents the mixing of automatic and manual allocation, which is error prone. However it is then the responsibility of the programmer to ensure that all values are in the required range, and that values are not unintentionally duplicated.

```
enum colour { red=3, blue, green, yellow=5 };     /* non compliant */
/* green and yellow represent the same value - this is duplication */

enum colour { red=3, blue=4, green=5, yellow=5 };     /* compliant */
/* green and yellow represent the same value - this is duplication */
```

Implemented by QAC messages:

0723 Initialize none, first only, or all entries in this enumerator list.

---

10 Arithmetic Type Conversions

Required 10.1

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

Notice also that in describing integer conversions, the concern is always with underlying type rather than actual type.

Rule 10.1 broadly encapsulates the following principles:

* No implicit conversions between signed and unsigned types

* No implicit conversions between integer and floating types

* No implicit conversions from wider to narrower types

* No implicit conversions of function arguments

* No implicit conversions of function return expressions

* No implicit conversions of complex expressions

The intention when restricting implicit conversion of complex expressions is to require that in a sequence of arithmetic operations within an expression, all operations should be conducted in exactly the same arithmetic type. Notice that this does not imply that all operands in an expression are of the same type.

The expression u32a + u16b + u16c is compliant - both additions will notionally be performed in type U32.

The expression u16a + u16b + u32c is not compliant - the first addition is notionally performed in type U16 and the second in type U32.

The word "notionally" is used because, in practice, the type in which arithmetic will be conducted will depend on the implemented size of an int. By observing the principle whereby all operations are performed in a consistent (underlying) type, it is possible to avoid programmer confusion and some of the dangers associated with integral promotion.

```
extern void foo1( uint8_t x );

int16_t t1( void )
{
    ...
    foo1( u8a );                          /* compliant     */
    foo1( u8a + u8b );                    /* compliant     */
```

```
    foo1( s8a );                        /* not compliant */
    foo1( u16a );                       /* not compliant */
    foo1( 2 );                          /* not compliant */
    foo1( 2U );                         /* compliant     */
    foo1( ( uint8_t )2 );               /* compliant     */
    ... s8a + u8a                       /* not compliant */
    ... s8a + ( int8_t )u8a             /* compliant     */
    s8b = u8a;                          /* not compliant */
    ... u8a + 5                         /* not compliant */
    ... u8a + 5U                        /* compliant     */
    ... u8a + ( uint8_t )5              /* compliant     */
    u8a = u16a;                         /* not compliant */
    u8a = ( uint8_t )u16a;              /* compliant     */
    u8a = 5UL;                          /* not compliant */
    ... u8a + 10UL                      /* compliant     */
    u8a = 5U;                           /* compliant     */
    ... u8a + 3                         /* not compliant */
    ... u8a >> 3                        /* compliant     */
    ... u8a >> 3U                       /* compliant     */
    pca = "P";                          /* compliant     */
    ... s32a + 80000                    /* compliant     */
    ... s32a + 80000L                   /* compliant     */
    u8a = u8b + u8c;                    /* compliant     */
    s16a = u8b + u8b;                   /* not compliant */
    s32a = u8b + u8c;                   /* not compliant */
    u8a  = f32a;                        /* not compliant */
    s32a = 1.0;                         /* not compliant */
    ...
    return s32a;                        /* not compliant */
    ...
    return s16a;                        /* compliant     */
    ...
    return 20000;                       /* compliant     */
    ...
    return 20000L;                      /* not compliant */
    ...
    return s8a;                         /* not compliant */
    ...
    return u16a;                        /* not compliant */
}

int16_t foo2( void )
{
    ...
    ... ( u16a + u16b ) + u32a          /* not compliant */
    ... s32a + s8a + s8b                /* compliant     */
    ... s8a + s8b + s32a                /* not compliant */
    f64a = s32a / s32b;                 /* not compliant */
    u32a = u16a + u16a;                 /* not compliant */
    s16a = s8a;                         /* compliant     */
    s16a = s16b + 20000;                /* compliant     */
    s32a = s16a + 20000;                /* not compliant */
    s32a = s16a + ( int32_t )20000;     /* compliant     */
    u16a = u16b + u8a;                  /* compliant     */
    foo1( u16a );                       /* not compliant */
    foo1( u8a + u8b );                  /* compliant     */
    ...
    return s16a;                        /* compliant     */
    ...
    return s8a;                         /* not compliant */
}
```

Implemented by QAC messages:

0570	This switch case label of 'essential type' '%1s', is not consistent with a controlling expression of essential type '%2s'.
0572	This switch case label of 'essential type' '%1s' is not consistent with a controlling expression which has an essential type of lower rank (%2s).
1256	An integer constant suffixed with L is being converted to type signed or unsigned long long on assignment.
1257	An integer constant suffixed with L or LL is being converted to a type of lower rank on assignment.
1290	An integer constant of 'essentially signed' type is being converted to unsigned type on assignment.
1291	An integer constant of 'essentially unsigned' type is being converted to signed type on assignment.
1292	An integer constant of 'essentially signed' type is being converted to type char on assignment.
1293	An integer constant of 'essentially unsigned' type is being converted to type char on assignment.
1294	An integer constant of 'essentially signed' type is being converted to type _Bool on assignment.
1295	An integer constant of 'essentially unsigned' type is being converted to type _Bool on assignment.
1296	An integer constant of 'essentially signed' type is being converted to enum type on assignment.
1297	An integer constant of 'essentially unsigned' type is being converted to enum type on assignment.
1298	An integer constant of 'essentially signed' type is being converted to floating type on assignment.
1299	An integer constant of 'essentially unsigned' type is being converted to floating type on assignment.

1317	Value of constant expression is not in the enum type to which it is being converted.
1461	Value of constant expression is not in the enum type to which it is being converted, but is bitwise OR of constants in the enum type.
1800	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this arithmetic operation.
1802	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this relational operation.
1803	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this equality operation.
1804	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this conditional operation.
1810	An operand of 'essentially character' type is being added to another operand of 'essentially character' type.
1811	An operand of 'essentially character' type is being subtracted from an operand of 'essentially signed' type.
1812	An operand of 'essentially character' type is being subtracted from an operand of 'essentially unsigned' type.
1813	An operand of 'essentially character' type is being balanced with an operand of 'essentially floating' type in this arithmetic operation.
1820	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
1821	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
1822	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this relational operation.
1823	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this equality operation.
1824	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
1830	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
1831	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
1832	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this relational operation.
1833	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this equality operation.
1834	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
1840	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
1841	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
1842	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this relational operation.
1843	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this equality operation.
1844	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
1850	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this arithmetic operation.
1851	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this bitwise operation.
1852	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this relational operation.
1853	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this equality operation.
1854	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this conditional operation.
1860	The operands of this arithmetic operator are of different 'essential signedness' but will generate a result of type 'signed int'.
1861	The operands of this bitwise operator are of different 'essential signedness' but will generate a result of type 'signed int'.
1862	The operands of this relational operator are of different 'essential signedness' but will both be promoted to 'signed int' for comparison.
1863	The operands of this equality operator are of different 'essential signedness' but will both be promoted to 'signed int' for comparison.
1864	The 2nd and 3rd operands of this conditional operator are of different 'essential signedness'. The result will be in the promoted type 'signed int'.
1880	The operands of this relational operator are expressions of different 'essential type' categories (%1s and %2s).
1881	The operands of this equality operator are expressions of different 'essential type' categories (%1s and %2s).
1882	The 2nd and 3rd operands of this conditional operator are expressions of different 'essential type' categories (%1s and

%2s).

1890	A composite expression of 'essentially signed' type (%1s) is being implicitly converted to a wider signed type, '%2s'.
1891	A composite expression of 'essentially unsigned' type (%1s) is being implicitly converted to a wider unsigned type, '%2s'.
2900	Constant: Positive integer value truncated by implicit conversion to a smaller unsigned type.
4117	Result of integer division operation implicitly converted to a floating type.
4401	An expression of 'essentially Boolean' type (%1s) is being converted to character type, '%2s' on assignment.
4402	An expression of 'essentially Boolean' type (%1s) is being converted to enum type, '%2s' on assignment.
4403	An expression of 'essentially Boolean' type (%1s) is being converted to signed type, '%2s' on assignment.
4404	An expression of 'essentially Boolean' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4405	An expression of 'essentially Boolean' type (%1s) is being converted to floating type, '%2s' on assignment.
4410	An expression of 'essentially character' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4412	An expression of 'essentially character' type (%1s) is being converted to enum type, '%2s' on assignment.
4413	An expression of 'essentially character' type (%1s) is being converted to signed type, '%2s' on assignment.
4414	An expression of 'essentially character' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4415	An expression of 'essentially character' type (%1s) is being converted to floating type, '%2s' on assignment.
4420	An expression of 'essentially enum' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4421	An expression of 'essentially enum' type (%1s) is being converted to character type, '%2s' on assignment.
4422	An expression of 'essentially enum' type (%1s) is being converted to a different enum type, '%2s' on assignment.
4423	An expression of 'essentially enum' type (%1s) is being converted to signed type, '%2s' on assignment.
4424	An expression of 'essentially enum' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4425	An expression of 'essentially enum' type (%1s) is being converted to floating type, '%2s' on assignment.
4430	An expression of 'essentially signed' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4431	An expression of 'essentially signed' type (%1s) is being converted to character type, '%2s' on assignment.
4432	An expression of 'essentially signed' type (%1s) is being converted to enum type, '%2s' on assignment.
4434	A non-constant expression of 'essentially signed' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4435	A non-constant expression of 'essentially signed' type (%1s) is being converted to floating type, '%2s' on assignment.
4436	A constant expression of 'essentially signed' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4437	A constant expression of 'essentially signed' type (%1s) is being converted to floating type, '%2s' on assignment.
4440	An expression of 'essentially unsigned' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4441	An expression of 'essentially unsigned' type (%1s) is being converted to character type, '%2s' on assignment.
4442	An expression of 'essentially unsigned' type (%1s) is being converted to enum type, '%2s' on assignment.
4443	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to a wider signed type, '%2s' on assignment.
4445	An expression of 'essentially unsigned' type (%1s) is being converted to floating type, '%2s' on assignment.
4446	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to signed type, '%2s' on assignment.
4447	A constant expression of 'essentially unsigned' type (%1s) is being converted to signed type, '%2s' on assignment.
4451	An expression of 'essentially floating' type (%1s) is being converted to character type, '%2s' on assignment.
4460	A non-constant expression of 'essentially signed' type (%1s) is being converted to narrower signed type, '%2s' on assignment.
4461	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to narrower unsigned type, '%2s' on assignment.
4463	A constant expression of 'essentially signed' type (%1s) is being converted to narrower signed type, '%2s' on assignment.
4464	A constant expression of 'essentially unsigned' type (%1s) is being converted to narrower unsigned type, '%2s' on assignment.
4470	A non-constant expression of 'essentially signed' type (%1s) is being passed to a function parameter of wider signed type, '%2s'.
4471	A non-constant expression of 'essentially unsigned' type (%1s) is being passed to a function parameter of wider unsigned type, '%2s'.
4480	A non-constant expression of 'essentially signed' type (%1s) is being returned from a function defined with a wider signed return type, '%2s'.
4481	A non-constant expression of 'essentially unsigned' type (%1s) is being returned from a function defined with a wider unsigned return type, '%2s'.
4490	A composite expression of 'essentially signed' type (%1s) is being converted to wider signed type, '%2s' on assignment.
4491	A composite expression of 'essentially unsigned' type (%1s) is being converted to wider unsigned type, '%2s' on assignment.
4500	An expression of 'essentially Boolean' type (%1s) is being used as an array subscript.
4501	An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
4502	An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
4503	An expression of 'essentially Boolean' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
4504	An expression of 'essentially Boolean' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
4505	An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this relational operator (%3s).

4507 An expression of 'essentially Boolean' type (%1s) is being used as the operand of this increment/decrement operator (%2s).

Required 10.2

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

Rule 10.2 broadly encapsulates the following principles:

* No implicit conversions between integer and floating types

* No implicit conversions from wider to narrower types

* No implicit conversions of function arguments

* No implicit conversions of function return expressions

* No implicit conversions of complex expressions

The intention when restricting implicit conversion of complex expressions is to require that in a sequence of arithmetic operations within an expression, all operations should be conducted in exactly the same arithmetic type. Notice that this does not imply that all operands in an expression are of the same type.

```
extern void foo1( uint8_t x );

int16_t t1( void )
{
    ...
    f32a = f64a;                        /* not compliant */
    f32a = 2.5;                         /* not compliant -
                                           unsuffixed floating
                                           constants are of type
                                           double */
    s32a = u8b + u8c;                   /* not compliant */
    f32a = 2.5F;                        /* compliant     */
    u8a  = f32a;                        /* not compliant */
    f32a = 1;                           /* not compliant */
    f32a = s16a;                        /* not compliant */
    ... f32a + 1                        /* not compliant */
    ... f64a * s32a                     /* not compliant */
}

int16_t foo2( void )
{
    ...
    f64a = f32a + f32b;                 /* not compliant */
    f64a = f64b + f32a;                 /* compliant     */
    f64a = s32a / s32b;                 /* not compliant */
    ...
}
```

Implemented by QAC messages:

1264 A suffixed floating constant is being converted to a different floating type on assignment.
1265 An unsuffixed floating constant is being converted to a different floating type on assignment.
1266 A floating constant is being converted to integral type on assignment.
1880 The operands of this relational operator are expressions of different 'essential type' categories (%1s and %2s).
1881 The operands of this equality operator are expressions of different 'essential type' categories (%1s and %2s).
1882 The 2nd and 3rd operands of this conditional operator are expressions of different 'essential type' categories (%1s and %2s).
1892 A composite expression of 'essentially floating' type (%1s) is being implicitly converted to a wider floating type, '%2s'.
1893 The 2nd and 3rd operands of this conditional operator are both 'essentially signed' ('%1s' and '%2s') but one is a composite expression of a narrower type than the other.
1894 The 2nd and 3rd operands of this conditional operator are both 'essentially unsigned' ('%1s' and '%2s') but one is a composite expression of a narrower type than the other.
1895 The 2nd and 3rd operands of this conditional operator are both 'essentially floating' ('%1s' and '%2s') but one is a composite expression of a narrower type than the other.
4450 An expression of 'essentially floating' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4452 An expression of 'essentially floating' type (%1s) is being converted to enum type, '%2s' on assignment.
4453 An expression of 'essentially floating' type (%1s) is being converted to signed type, '%2s' on assignment.
4454 An expression of 'essentially floating' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4462 A non-constant expression of 'essentially floating' type (%1s) is being converted to narrower floating type, '%2s' on

assignment.

4465 A constant expression of 'essentially floating' type (%1s) is being converted to narrower floating type, '%2s' on assignment.

4472 A non-constant expression of 'essentially floating' type (%1s) is being passed to a function parameter of wider floating type, '%2s'.

4482 A non-constant expression of 'essentially floating' type (%1s) is being returned from a function defined with a wider floating return type, '%2s'.

4492 A composite expression of 'essentially floating' type (%1s) is being converted to wider floating type, '%2s' on assignment.

Required 10.3

The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.

If a cast is to be used on any complex expression, the type of cast that may be applied is severely restricted. Conversions on complex expressions are often a source of confusion and it is therefore wise to be cautious. In order to comply with these rules, it may be necessary to use a temporary variable and introduce an extra statement.

```
... ( uint32_t )( u16a + u16b )              /* not compliant */
... ( uint32_t )u16a + u16b                  /* compliant     */
... ( uint32_t )u16a + ( uint32_t )u16b      /* compliant     */
... ( int16_t )( s32a - 12345 )              /* compliant     */
... ( uint8_t )( u16a * u16b )               /* compliant     */
... ( uint16_t )( u8a * u8b )                /* not compliant */
... ( int16_t )( s32a * s32b )               /* compliant     */
... ( int32_t )( s16a * s16b )               /* not compliant */
... ( uint16_t )( f64a + f64b )              /* not compliant */
... ( float32_t )( u16a + u16b )             /* not compliant */
... ( float64_t )foo1( u16a + u16b )         /* compliant     */
... ( int32_t )buf16a[ u16a + u16b ]         /* compliant     */
```

Implemented by QAC messages:

4118 Result of integer division operation cast to a floating type.

4390 A composite expression of 'essentially signed' type (%1s) is being cast to a wider signed type, '%2s'.

4391 A composite expression of 'essentially unsigned' type (%1s) is being cast to a wider unsigned type, '%2s'.

4393 A composite expression of 'essentially signed' type (%1s) is being cast to a different type category, '%2s'.

4394 A composite expression of 'essentially unsigned' type (%1s) is being cast to a different type category, '%2s'.

Required 10.4

The value of a complex expression of floating type shall only be cast to a floating type that is narrower or of the same size.

If a cast is to be used on any complex expression, the type of cast that may be applied is severely restricted. Conversions on complex expressions are often a source of confusion and it is therefore wise to be cautious. In order to comply with these rules, it may be necessary to use a temporary variable and introduce an extra statement.

```
... ( float32_t )( f64a + f64b )             /* compliant     */
... ( float64_t )( f32a + f32b )             /* not compliant */
... ( float64_t )f32a                        /* compliant     */
... ( float64_t )( s32a / s32b )             /* not compliant */
... ( float64_t )( s32a > s32b )             /* not compliant */
... ( float64_t )s32a / ( float32_t )s32b    /* compliant     */
```

Implemented by QAC messages:

4392 A composite expression of 'essentially floating' type (%1s) is being cast to a wider floating type, '%2s'.

4395 A composite expression of 'essentially floating' type (%1s) is being cast to a different type category, '%2s'.

Required 10.5

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

When these operators (~ and <<) are applied to small integer types (unsigned char or unsigned short), the operations are preceded by integral promotion, and the result may contain high order bits which have not been anticipated. For example:

```
uint8_t port = 0x5aU;
uint8_t result_8;
uint16_t result_16;
uint16_t mode;

result_8 = ( ~port ) >> 4;                   /* not compliant */
```

~port is 0xffa5 on a 16-bit machine but 0xffffffa5 on a 32-bit machine. In either case the value of result is 0xfa, but 0x0a may have been expected. This danger is avoided by inclusion of the cast as shown below:

```
result_8 = ( ( uint8_t )( ~port ) ) >> 4;                /* compliant */
result_16 = ( ( uint16_t )( ~( uint16_t )port ) ) >> 4;  /* compliant */
```

A similar problem exists when the << operator is used on small integer types and high order bits are retained. For example:

```
result_16 = ( ( port << 4 ) & mode ) >> 6;           /* not compliant */
```

The value in result_16 will depend on the implemented size of an int. Addition of a cast avoids any ambiguity.

```
result_16 = ( ( uint16_t )( ( uint16_t )port << 4 ) & mode ) >> 6;
                                                     /* compliant */
```

No cast is required if the result of the bitwise operation is:

a. immediately assigned to an object of the same underlying type as the operand;
b. used as a function argument of the same underlying type as the operand;
c. used as a return expression of a function whose return type is of the same underlying type as the operand.

Implemented by QAC messages:

4397	An expression which is the result of a ~ or << operation has not been cast to its essential type.
4398	An expression which is the result of a ~ or << operation has been cast to a different essential type category.
4399	An expression which is the result of a ~ or << operation has been cast to a wider type.
4498	An expression which is the result of a ~ or << operation has been converted to a different essential type category on assignment.
4499	An expression which is the result of a ~ or << operation has been converted to a wider essential type on assignment.

Required 10.6 — A "U" suffix shall be applied to all constants of unsigned type.

The type of an integer constant is a potential source of confusion, because it is dependent on a complex combination of factors including:

1. The magnitude of the constant

2. The implemented sizes of the integer types

3. The presence of any suffixes

4. The number base in which the value is expressed (i.e. decimal, octal or hexadecimal).

For example, the integer constant "40000" is of type int in a 32-bit environment but of type long in a 16-bit environment. The value 0x8000 is of type unsigned int in a 16-bit environment, but of type (signed) int in a 32-bit environment.

Note the following:

* Any value with a "U" suffix is of unsigned type

* An unsuffixed decimal value less than 2^{31} is of signed type

But:

* An unsuffixed hexadecimal value greater than or equal to 2^{15} may be of signed or unsigned type

* An unsuffixed decimal value greater than or equal to 2^{31} may be of signed or unsigned type

Signedness of constants should be explicit. Consistent signedness is an important principle in constructing well formed expressions. If a constant is of an unsigned type, it is helpful to avoid ambiguity by applying a "U" suffix. When applied to larger values, the suffix may be redundant (in the sense that it does not influence the type of the constant); however its presence is a valuable contribution towards clarity.

Implemented by QAC messages:

1281	Integer literal constant is of an unsigned type but does not include a "U" suffix.

---

11 Pointer Type Conversions

Required 11.1 — Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Conversion of a function pointer to a different type of pointer results in undefined behaviour. This means that a function pointer can be

converted to or from an integral type. No other conversions involving function pointers are permitted.

Reference Undefined 27-28;

Implemented by QAC messages:

0302 [u] Cast between a pointer to function and a floating type.
0307 [u] Cast between a pointer to object and a pointer to function.
0313 Casting to different function pointer type.

Required 11.2 — Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

Such conversions are undefined. This rule means that an object pointer can be converted to or from:

a. An integral type;
b. Another pointer to object type;
c. A pointer to void.

No other conversions involving object pointers are permitted.

Reference Undefined 29;

Implemented by QAC messages:

0301 [u] Cast between a pointer to object and a floating type.

Advisory 11.3 — A cast should not be performed between a pointer type and an integral type.

The size of integer that is required when a pointer is converted to an integer is implementation-defined. Casting between a pointer and an integer type should be avoided where possible, but may be unavoidable when addressing memory mapped registers or other hardware specific features.

Reference Implementation 24;

Implemented by QAC messages:

0303 [l] Cast between a pointer to volatile object and an integral type.
0305 [l] Cast between a pointer to function and an integral type.
0306 [l] Cast between a pointer to object and an integral type.
0360 An expression of pointer type is being converted to type _Bool on assignment.
0361 An expression of pointer type is being cast to type _Bool.
0362 An expression of essentially Boolean type is being cast to a pointer.

Advisory 11.4 — A cast should not be performed between a pointer to object type and a different pointer to object type.

Conversions of this type may be invalid if the new pointer type requires a stricter alignment.

```
uint8_t  *p1;
uint32_t *p2;

p2 = ( uint32_t * )p1;    /* Incompatible alignment ? */
```

Implemented by QAC messages:

0310 Casting to different object pointer type.
0316 [l] Cast from a pointer to void to a pointer to object type.
0317 [l] Implicit conversion from a pointer to void to a pointer to object type.

Required 11.5 — A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.

Any attempt to remove the qualification associated with the addressed type by using casting is a violation of the principle of type qualification. Notice that the qualification referred to here is not the same as any qualification that may be applied to the pointer itself.

```
uint16_t                x;
uint16_t * const        cpi = &x;     /* const pointer              */
uint16_t * const      * pcpi;         /* pointer to const pointer   */
const uint16_t *      * ppci;         /* pointer to pointer to const */
uint16_t *            * ppi;
```

```
const uint16_t         * pci;          /* pointer to const           */
volatile uint16_t      * pvi;          /* pointer to volatile        */
uint16_t               * pi;
...
pi = cpi;                              /* Compliant - no conversion
                                          no cast required          */
pi  = ( uint16_t * )pci;               /* Not compliant              */
pi  = ( uint16_t * )pvi;               /* Not compliant              */
ppi = ( uint16_t * * )pcpi;            /* Not compliant              */
ppi = ( uint16_t * * )ppci;            /* Not compliant              */
```

Reference Undefined 39-40;

Implemented by QAC messages:

0311 Dangerous pointer cast results in loss of const qualification.

0312 Dangerous pointer cast results in loss of volatile qualification.

12 Expressions

Advisory 12.1 Limited dependence should be placed on C's operator precedence rules in expressions.

In addition to the use of parentheses to override default operator precedence, parentheses should also be used to emphasise it. It is easy to make a mistake with the rather complicated precedence rules of C, and this approach helps to avoid such errors, and helps to make the code easier to read. However, do not add too many parentheses so as to clutter the code and make it unreadable.

The following guidelines are suggested in deciding when parentheses are required:

* no parentheses are required for the right-hand operand of an assignment operator unless the right-hand side itself contains an assignment expression:

```
x = a + b;             /* acceptable      */
x = ( a + b );         /* () not required */
```

* no parentheses are required for the operand of a unary operator:

```
x = a * -1;            /* acceptable      */
x = a * ( -1 );        /* () not required */
```

* otherwise, the operands of binary and ternary operators shall be cast-expressions (see section 6.3.4 of ISO/IEC 9899:1990 [2]) unless all the operators in the expression are the same.

```
x = a + b + c;                 /* acceptable, but care needed */
x = f( a + b, c );             /* no () required for a + b    */
x = ( a == b ) ? a : ( a - b );
if ( a && b && c )             /* acceptable                  */
x = ( a + b ) - ( c + d );
x = ( a * 3 ) + c + d;
x = ( uint16_t ) a + b;        /* no need for ( ( uint16_t ) a )  */
```

* even if all operators are the same, parentheses may be used to control the order of operation. Some operators (e.g. addition and multiplication) that are associative in algebra are not necessarily associative in C. Similarly, integer operations involving mixed types (prohibited by several rules) may produce different results because of the integral promotions. The following example written for a 16-bit implementation demonstrates that addition is not associative and that it is important to be clear about the structure of an expression:

```
uint16_t a = 10U;
uint16_t b = 65535U;
uint32_t c = 0U;
uint32_t d;

d = ( a + b ) + c;   /* d is 9; a + b wraps modulo 65536 */
d = a + ( b + c );   /* d is 65545                       */
/* this example also deviates from several other rules */
```

Note that Rule 12.5 is a special case of this rule applicable solely to the logical operators, && and ||.

See also Rule 12.5

Implemented by QAC messages:

3389 Extra parentheses recommended to clarify the ordering of a % operator and another arithmetic operator (* / % + -).

3391 Extra parentheses recommended. A conditional operation is the operand of another conditional operator.

3392	Extra parentheses recommended. A shift, relational or equality operation is the operand of a second identical operator.
3393	Extra parentheses recommended. An arithmetic operation (* / + -) is the operand of a different operator with the same precedence.
3394	Extra parentheses recommended. A shift, relational or equality operation is the operand of a different operator with the same precedence.
3395	Extra parentheses recommended. A * or / operation is the operand of a + or - operator.
3396	Extra parentheses recommended. A binary operation is the operand of a conditional operator.
3397	Extra parentheses recommended. A binary operation is the operand of a binary operator with different precedence.

Required 12.2 — The value of an expression shall be the same under any order of evaluation that the standard permits.

Apart from a few operators (notably the function call operator (), &&, ||, ?: and , (comma)) the order in which sub-expressions are evaluated is unspecified and can vary. This means that no reliance can be placed on the order of evaluation of sub-expressions, and in particular no reliance can be placed on the order in which side effects occur. Those points in the evaluation of an expression at which all previous side effects can be guaranteed to have taken place are called 'sequence points'. Sequence points and side effects are described in sections 5.1.2.3, 6.3 and 6.6 of ISO/IEC 9899:1990 [2].

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

Reference Unspecified 7-9;Undefined 18;

Implemented by QAC messages:

0400	[U] '%s' is modified more than once between sequence points - evaluation order unspecified.
0401	[U] '%s' may be modified more than once between sequence points - evaluation order unspecified.
0402	[U] '%s' is modified and accessed between sequence points - evaluation order unspecified.
0403	[U] '%s' may be modified and accessed between sequence points - evaluation order unspecified.

Required 12.3 — The sizeof operator shall not be used on expressions that contain side effects.

A possible programming error in C is to apply the sizeof operator to an expression and expect the expression to be evaluated. However the expression is not evaluated: sizeof only acts on the type of the expression. To avoid this error, sizeof shall not be used on expressions that contain side effects, as the side effects will not occur. sizeof() shall only be applied to an operand which is a type or an object. This may include volatile objects. For example:

```
int32_t i;
int32_t j;
j = sizeof( i = 1234 );
     /* j is set to the sizeof the type of i which is an int */
     /* i is not set to 1234.                                */
```

Implemented by QAC messages:

3307	The operand of 'sizeof' is an expression with implied side effects, but they will not be evaluated.

Required 12.4 — The right hand operand of a logical && or || operator shall not contain side effects.

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

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

For example:

```
if ( ishigh && ( x == i++ ) )     /* Not compliant                */
if ( ishigh && ( x == f( x ) ) )  /* Only acceptable if f( x ) is
                                     known to have no side effects */
```

The operations that cause side effects are described in section 5.1.2.3 of ISO/IEC 9899:1990 [2] as accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations, which cause changes in the state of the execution environment of the calling function.

Implemented by QAC messages:

3415	Right hand operand of '&&' or '||' is an expression with possible side effects.

Required 12.5 — The operands of a logical && or || shall be primary-expressions.

'Primary expressions' are defined in ISO/IEC 9899:1990 [2], section 6.3.1. Essentially they are either a single identifier, or a constant, or a parenthesised expression. The effect of this rule is to require that if an operand is other than a single identifier or constant then it must be parenthesised. Parentheses are important in this situation both for readability of code and for ensuring that the behaviour is as the programmer intended. Where an expression consists of either a sequence of only logical && or a sequence of only logical ||, extra parentheses are not required.

For example write:

```
if ( ( x == 0 ) && ishigh )   /* make x == 0 primary                  */
if ( x || y || z )            /* exception allowed, if x, y and z are
                                 Boolean                              */
if ( x || ( y && z ) )        /* make y && z primary                  */
if ( x && ( !y ) )            /* make !y primary                      */
if ( ( is_odd ( y ) ) && x )  /* make call primary                    */
```

Where an expression consists of either a sequence of only logical && or a sequence of only logical ||, extra parentheses are not required.

```
if ( ( x > c1 ) && ( y > c2 ) && ( z > c3 ) )      /* Compliant       */
if ( ( x > c1 ) && ( y > c2 ) || ( z > c3 ) )      /* not compliant   */
if ( ( x > c1 ) && ( ( y > c2 ) || ( z > c3 ) ) )  /* Compliant
                                                        extra () used */
```

Note that this rule is a special case of Guideline 12.1.

See also

Guideline 12.1

Implemented by QAC messages:

3398	Extra parentheses recommended. A function call, array subscript, or member operation is the operand of a logical && or ||.
3399	Extra parentheses recommended. A unary operation is the operand of a logical && or ||.
3400	Extra parentheses recommended. A binary operation is the operand of a logical && or ||.

Advisory 12.6 — The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !, =, ==, != and ?:).

The logical operators &&, || and ! can be easily confused with the bitwise operators &, | and ~. See 'Boolean expressions' in the glossary.

Reference

Koenig 48;

Implemented by QAC messages:

3322	Operand of a logical ! operator is a constant expression which is not a 'Boolean' value.
3377	Operand of a logical && or || operator is a constant expression which is not a 'Boolean' value.
4101	Both operands of & operator are 'Boolean' expressions.
4102	Both operands of | operator are 'Boolean' expressions.
4103	Both operands of arithmetic or bitwise operator are 'Boolean' expressions.
4104	Left hand operand of arithmetic or bitwise operator is a 'Boolean' expression.
4105	Right hand operand of arithmetic or bitwise operator is a 'Boolean' expression.
4106	Both operands of && operator are arithmetic or bitwise expressions.
4107	Both operands of || operator are arithmetic or bitwise expressions.
4108	Left hand operand of logical operator is an arithmetic or bitwise expression.
4109	Right hand operand of logical operator is an arithmetic or bitwise expression.
4110	Operand of ! operator is an arithmetic or bitwise expression.
4111	Right hand operand of relational operator is a 'Boolean' expression.
4112	Left hand operand of relational operator is a 'Boolean' expression.
4113	Both operands of relational operator are 'Boolean' expressions.
4114	Operand of ~ operator is a 'Boolean' expression.
4115	Operand of logical && or || operator is not an 'essentially Boolean' expression.
4116	Operand of logical ! operator is not an 'essentially Boolean' expression.

Required 12.7 — Bitwise operators shall not be applied to operands whose underlying type is signed.

Bitwise operations (~, <<, <<=, >>, >>=, &, &=, ^, ^=, | and |=) are not normally meaningful on signed integers. Problems can arise if,

for example, a right shift moves the sign bit into the number, or a left shift moves a numeric bit into the sign bit.

See section 6.10 in the MISRA-C:2004 document for a description of underlying type.

Reference Implementation 17,19;

Implemented by QAC messages:

4532 An expression of 'essentially signed' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
4533 An expression of 'essentially signed' type (%1s) is being used as the left-hand operand of this shift operator (%2s).

Required 12.8 — The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

If, for example, the left hand operand of a left-shift or right-shift is a 16-bit integer, then it is important to ensure that this is shifted only by a number between 0 and 15 inclusive.

See section 6.10 for a description of underlying type.

There are various ways of ensuring this rule is followed. The simplest is for the right hand operand to be a constant (whose value can then be statically checked). Use of an unsigned integer type will ensure that the operand is non-negative, so then only the upper limit needs to be checked (dynamically at run time or by review). Otherwise both limits will need to be checked. s

```
u8a  = ( uint8_t ) ( u8a << 7 );                /* compliant     */
u8a  = ( uint8_t ) ( u8a << 9 );                /* not compliant */
u16a = ( uint16_t )( ( uint16_t ) u8a << 9 );   /* compliant     */
```

Reference Undefined 32;

Implemented by QAC messages:

0499 Right operand of shift operator is greater than or equal to the width of the essential type of the left operand.
2790 Constant: Right hand operand of shift operator is negative or too large.

Required 12.9 — The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Applying the unary minus operator to an expression of type unsigned int or unsigned long generates a result of type unsigned int or unsigned long respectively and is not a meaningful operation. Applying unary minus to an operand of smaller unsigned integer type may generate a meaningful signed result due to integral promotion, but this is not good practice.

See section 6.10 for a description of underlying type.

Implemented by QAC messages:

3101 Unary '-' applied to an operand of type unsigned int or unsigned long gives an unsigned result.
3102 Unary '-' applied to an operand whose underlying type is unsigned.

Required 12.10 — The comma operator shall not be used.

Use of the comma operator is generally detrimental to the readability of code, and the same effect can be achieved by other means.

Implemented by QAC messages:

3417 The comma operator has been used outside a 'for' statement.
3418 The comma operator has been used in a 'for' statement.

Advisory 12.11 — Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Because unsigned integer expressions do not strictly overflow, but instead wrap around in a modular way, any constant unsigned integer expressions which in effect 'overflow' will not be detected by the compiler. Although there may be good reasons at run-time to rely on the modular arithmetic provided by unsigned integer types, the reasons for using it at compile-time to evaluate a constant expression are less obvious. Any instance of an unsigned integer constant expression wrapping around is therefore likely to indicate a programming error.

This rule applies equally to all phases of the translation process. Constant expressions that the compiler chooses to evaluate at compile time are evaluated in such a way that the results are identical to those that would be obtained by evaluation on the target with the exception of those appearing in conditional preprocessing directives. For such directives, the usual rules of arithmetic (see section 6.4 of ISO/IEC 9899:1990 [2]) apply but the int and unsigned int types behave instead as if they were long and unsigned long respectively.

For example, on a machine with a 16-bit int type and a 32-bit long type:

```
#define START   0x8000
#define END     0xFFFF
#define LEN     0x8000

#if ( ( START + LEN ) > END )
#error Buffer Overrun /* OK because START and LEN are unsigned long */
#endif

#if ( ( ( END - START ) - LEN ) < 0 )
        #error Buffer Overrun
        /* Not OK: subtraction result wraps around to 0xFFFFFFFF */
#endif

/* contrast the above START + LEN with the following */
if ( ( START + LEN ) > END )
{
   error ( "Buffer overrun" );
      /* Not OK: START + LEN wraps around to 0x0000 due to unsigned
                                                        int arithmetic */
}
```

Implemented by QAC messages:

2910 Constant: Wraparound in unsigned arithmetic operation.

Required 12.12 **The underlying bit representations of floating-point values shall not be used.**

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

Reference Unspecified 6;Implementation 20;

Implemented by QAC messages:

3629 Union contains member of floating type.

Advisory 12.13 **The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.**

It is the intention of the rule that when the increment or decrement operator is used, it should be the only side effect in the statement. The use of increment and decrement operators in combination with other arithmetic operators is not recommended because:

* It can significantly impair the readability of the code.

* It introduces additional side effects into a statement with the potential for undefined behaviour.

It is safer to use these operations in isolation from any other arithmetic operators.

For example a statement such as the following is not compliant:

```
u8a = ++u8b + u8c--;      /* Not compliant */
```

The following sequence is clearer and therefore safer:

```
++u8b;
u8a = u8b + u8c;
u8c--;
```

Implemented by QAC messages:

3440 Using the value resulting from a ++ or -- operation.

---

13 Control Statement Expressions

Required 13.1 **Assignment operators shall not be used in expressions that yield a Boolean value.**

No assignments are permitted in any expression which is considered to have a Boolean value. This precludes the use of both simple and compound assignment operators in the operands of a Boolean-valued expression. However, it does not preclude assigning a Boolean value to a variable.

If assignments are required in the operands of a Boolean-valued expression then they must be performed separately outside of those

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

See 'Boolean expressions' in the glossary.

For example write:

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

and not:

```
if ( ( x = y ) != 0 )   /* Boolean by context */
{
    foo();
}
```

or even worse:

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

Reference Koenig 6;

Implemented by QAC messages:

3326 The result of an assignment is being used in a logical operation.

Advisory 13.2 **Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.**

Where a data value is to be tested against zero then the test should be made explicit. The exception to this rule is when data represents a Boolean value, even though in C this will in practice be an integer. This rule is in the interests of clarity, and makes clear the distinction between integers and logical values.

For example, if x is an integer, then:

```
if ( x != 0 )    /* Correct way of testing x is non-zero */
if ( y )         /* Not compliant, unless y is effectively
                    Boolean data (e.g. a flag) */
```

Implemented by QAC messages:

3344 Controlling expression is not an 'essentially Boolean' expression.
4528 An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4529 An expression of 'essentially enum' type (%1s) is being used as the first operand of this conditional operator (%2s).
4538 An expression of 'essentially signed' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4539 An expression of 'essentially signed' type (%1s) is being used as the first operand of this conditional operator (%2s).
4548 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4549 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the first operand of this conditional operator (%2s).
4558 An expression of 'essentially unsigned' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4559 An expression of 'essentially unsigned' type (%1s) is being used as the first operand of this conditional operator (%2s).
4568 An expression of 'essentially floating' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4569 An expression of 'essentially floating' type (%1s) is being used as the first operand of this conditional operator (%2s).

Required 13.3 **Floating-point expressions shall not be tested for equality or inequality.**

The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true even when they are expected to. In addition the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another. For example the result of the test in the following code is unpredictable:

```
float32_t x, y;
/* some calculations here          */
if ( x == y )     /* not compliant */
{ /* ... */ }
```

```
if ( x == 0.0f )  /* not compliant */
{ /* ... */ }
```

An indirect test is equally problematic and is also forbidden by this rule, for example:

```
if ( ( x <= y ) && ( x >= y ) )
{ /* ... */ }
```

The recommended method for achieving deterministic floating-point comparisons is to write a library that implements the comparison operations. The library should take into account the floating-point granularity (FLT_EPSILON) and the magnitude of the numbers being compared.

See also Rule 13.4, Rule 20.3

Implemented by QAC messages:

3341	Comparing floating point expressions for equality (with '==' or '!=').

Required 13.4 — The controlling expression of a for statement shall not contain any objects of floating type.

The controlling expression may include a loop counter, whose value is tested to determine termination of the loop. Floating-point variables shall not be used for this purpose. Rounding and truncation errors can be propagated through the iterations of the loop, causing significant inaccuracies in the loop variable, and possibly giving unexpected results when the test is performed. For example the number of times the loop is performed may vary from one implementation to another, and may be unpredictable.

See also Rule 13.3

Implemented by QAC messages:

3340	Floating point variable used as 'for' loop control variable.
3342	Controlling expression of 'for' loop is a floating point comparison.

Required 13.5 — The three expressions of a for statement shall be concerned only with loop control.

The three expressions of a for statement shall be used only for these purposes:

First expression - Initialising the loop counter

Second expression - Shall include testing the loop counter, and optionally other loop control variables

Third expression - Increment or decrement of the loop counter

The following options are allowed:

a. All three expressions shall be present;
b. The second and third expressions shall be present with prior initialisation of the loop counter;
c. All three expressions shall be empty for a deliberate infinite loop.

Implemented by QAC messages:

2462	The variable initialized in the first expression of this 'for' statement is not the variable identified as the 'loop control variable' (%s).
2463	The variable incremented in the third expression of this 'for' statement is not the variable identified as the 'loop control variable' (%s).

Required 13.6 — Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

Loop counters shall not be modified in the body of the loop. However other loop control variables representing logical values may be modified in the loop, for example a flag to indicate that something has been completed, which is then tested in the for statement.

```
flag = 1;
for ( i = 0; ( i < 5 ) && ( flag == 1 ); i++ )
{
    flag = 0;   /* Compliant - allows early termination of loop */
    i = i + 3;  /* Not compliant - altering the loop counter    */
}
```

Implemented by QAC messages:

2469	Loop control variable in this 'for' statement, %s, is modified in the body of the loop.

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

If a Boolean operator yields a result that can be proven to be always "true" or always "false", it is highly likely that there is a programming error.

```
enum ec {RED, BLUE, GREEN} col;
...
if ( u16a < 0 )             /* Not compliant - u16a is always >= 0 */
...
if ( u16a <= 0xffff )               /* Not compliant - always true */
...
if ( s8a < 130 )                    /* Not compliant - always true */
...
if ( ( s8a < 10 ) && ( s8a > 20 ) ) /* Not compliant - always false */
...
if ( ( s8a < 10 ) || ( s8a > 5 ) )  /* Not compliant - always true */
...
if ( col <= GREEN )                 /* Not compliant - always true */
...
if ( s8a > 10 )
{
    if ( s8a > 5 )          /* Not compliant - s8a is not volatile */
    {
    }
}
```

Implemented by QAC messages:

2990	The value of this loop controlling expression is always 'true'.
2991	The value of this 'if' controlling expression is always 'true'.
2992	The value of this 'if' controlling expression is always 'false'.
2993	The value of this 'do - while' loop controlling expression is always 'false'. The loop will only be executed once.
2994	The value of this 'while' or 'for' loop controlling expression is always 'false'. The loop will not be entered.
2995	The result of this logical operation is always 'true'.
2996	The result of this logical operation is always 'false'.

14 Control Flow

Required 14.1 There shall be no unreachable code.

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

A portion of code is unreachable if there is no control flow path from the relevant entry point to that code. For example, unlabelled code following an unconditional control transfer is unreachable:

```
switch ( event )
{
case E_wakeup:
    do_wakeup();
    break;          /* unconditional control transfer   */
    do_more();      /* Not compliant - unreachable code */
    /* ... */
default:
    /* ... */
    break;
}
```

A whole function will be unreachable if there is no means by which it can be called.

Code that is excluded by pre-processor directives is not present following pre-processing, and is therefore not subject to this rule.

Implemented by QAC messages:

0594	Negative 'case' label expression is incompatible with unsigned controlling expression in 'switch' statement.
1460	'Switch' label value, %s, not contained in enum type.
1503	The function '%1s' is defined but is not used within this project.
2008	Code statements precede the first label in this 'switch' construct.
2742	This 'if' controlling expression is a constant expression and its value is 'false'.
2744	This 'while' or 'for' loop controlling expression is a constant expression and its value is 'false'. The loop will not be entered.
2880	This code is unreachable.
2882	This 'switch' statement will bypass the initialization of local variables.
3219	Static function '%s()' is not used within this translation unit.

Required 14.2 — All non-null statements shall either (i) have at least one side effect however executed, or (ii) cause control flow to change.

Any statement (other than a null statement) which has no side effect and does not result in a change of control flow will normally indicate a programming error, and therefore a static check for such statements shall be performed. For example, the following statements do not necessarily have side effects when executed:

```
/* assume uint16_t x;
   and    uint16_t i; */
...
x >= 3u;              /* not compliant: x is compared to 3,
                         and the answer is discarded */
```

Note that 'null statement' and 'side effect' are defined in ISO/IEC 9899:1990 [2] sections 6.6.3 and 5.1.2.3 respectively.

Implemented by QAC messages:

3110	The left-hand operand of this ',' has no side effects.
3112	This statement has no side-effect - it can be removed.
3425	One branch of this conditional operation is a redundant expression.
3426	Right hand side of comma expression has no side effect and its value is not used.
3427	Right hand side of logical operator has no side effect and its value is not used.

Required 14.3 — Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

Null statements should not normally be deliberately included, but where they are used they shall appear on a line by themselves. White-space characters may precede the null statement to preserve indentation. If a comment follows the null statement then at least one white-space character shall separate the null statement from the comment. The use of a white-space character to separate the null statement from any following comment is required on the grounds that it provides an important visual cue to reviewers. Following this rule enables a static checking tool to warn of null statements appearing on a line with other text, which would normally indicate a programming error. For example:

```
while ( ( port & 0x80 ) == 0 )
{
    ; /* wait for pin - Compliant */
    /* wait for pin */ ; /* Not compliant, comment before ; */
    ;/* wait for pin - Not compliant, no white-space char after ; */
}
```

Implemented by QAC messages:

3138	Null statement is located close to other code or comments.

Required 14.4 — The goto statement shall not be used.

Implemented by QAC messages:

2001	A 'goto' statement has been used.

Required 14.5 — The continue statement shall not be used.

Implemented by QAC messages:

0770	A 'continue' statement has been used.

Required 14.6 — For any iteration statement there shall be at most one break statement used for loop termination.

These rules are in the interests of good structured programming. One break statement is allowed in a loop since this allows, for example, for dual outcome loops or for optimal coding.

Implemented by QAC messages:

0771	More than one 'break' statement has been used to terminate this iteration statement.

Required 14.7 — A function shall have a single point of exit at the end of the function.

This is required by IEC 61508, under good programming style.

Reference IEC 61508 Part 3 Table B.9;

Implemented by QAC messages:

2889 This function has more than one 'return' path.

Required 14.8 **The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.**

The statement that forms the body of a switch statement or a while, do ... while or for loop, shall be a compound statement (enclosed within braces), even if that compound statement contains a single statement.

For example:

```
for ( i = 0; i < N_ELEMENTS; ++i )
{
    buffer[i] = 0;       /* Even a single statement must be in braces */
}

while ( new_data_available )
    process_data();       /* Incorrectly not enclosed in braces */
    service_watchdog();   /* Added later but, despite the appearance
                             (from the indent) it is actually not
                             part of the body of the while statement,
                             and is executed only after the loop has
                             terminated */
```

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

Implemented by QAC messages:

2212 Body of control statement is not enclosed within braces.

2214 Body of control statement is on the same line and is not enclosed within braces.

Required 14.9 **An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.**

For example:

```
if ( test1 )
{
    x = 1;              /* Even a single statement must be in braces */
}
else if ( test2 )       /* No need for braces in else if */
{
    x = 0;              /* Single statement must be in braces */
}
else
    x = 3; /* This was (incorrectly) not enclosed in braces */
    y = 2; /* This line was added later but, despite the appearance
              (from the indent) it is actually not part of the else, and
               is executed unconditionally */
```

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

Reference Koenig 24;

Implemented by QAC messages:

2212 Body of control statement is not enclosed within braces.

2214 Body of control statement is on the same line and is not enclosed within braces.

3402 Braces are needed to clarify the structure of this 'if'-'if'-'else' statement.

Required 14.10 **All if ... else if constructs shall be terminated with an else clause.**

This rule applies whenever an if statement is followed by one or more else if statements; the final else if shall be followed by an else statement. In the case of a simple if statement then the else statement need not be included.

The requirement for a final else statement is defensive programming. The else statement shall either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final default clause in a switch statement (15.3)

For example this code is a simple if statement:

```
if ( x < 0 )
{
    log_error( 3 );
     x = 0;
}                                /* else not needed */
```

whereas the following code demonstrates an if, else if construct

```
if ( x < 0 )
{
    log_error( 3 );
    x = 0;
}
else if ( y < 0 )
{
    x = 3;
}
else   /* this else clause is required, even if the     */
{      /* programmer expects this will never be reached */
       /* no change in value of x */
}
```

See also Rule 15.3

Implemented by QAC messages:

2004 No concluding 'else' exists in this 'if'-'else'-'if' statement.

---

15 Switch Statements

The syntax for the switch in C is weak, allowing complex, unstructured behaviour. The following text describes the syntax for switch statements as defined by MISRA-C and is normative. This, and the associated rules, enforces a simple and consistent structure on the switch statement.

The following syntax rules are additional to the C standard syntax rules. All syntax rules not defined below are as defined in the C standard.

switch-statement:

> **switch (** expression **) {** case-label-clause-list default-label-clause **}**

case-label-clause-list:

> case-label case-clause$_{opt}$
> case-label-clause-list case-label case-clause$_{opt}$

case-label:

> **case** constant-expression **:**

case-clause:

> statement-list$_{opt}$ **break ;**
> **{** declaration-list$_{opt}$ statement-list$_{opt}$ **break ; }**

default-label-clause:

> default-label default-clause

default-label:

> **default :**

default-clause:

> case-clause

and

statement:

> /* labelled_statement removed */
> compound_statement
> expression_statement
> selection_statement

iteration_statement
/* jump_statement removed or just restricted to return depending on other rules */

The following terms are also used within the text of the rules:

switch label	Either a case label or default label.
case clause	The code between any two switch labels.
default clause	The code between the default label and the end of the switch statement.
switch clause	Either a case clause or a default clause.

Required 15.0 — The MISRA C switch syntax shall be used.

The syntax for the switch in C is weak, allowing complex, unstructured behaviour. The following text describes the syntax for switch statements as defined by MISRA-C and is normative. This, and the associated rules, enforces a simple and consistent structure on the switch statement.

The following syntax rules are additional to the C standard syntax rules. All syntax rules not defined below are as defined in the C standard.

switch-statement:

switch (expression **) {** case-label-clause-list default-label-clause **}**

case-label-clause-list:

case-label case-clause$_{opt}$
case-label-clause-list case-label case-clause$_{opt}$

case-label:

case constant-expression **:**

case-clause:

statement-list$_{opt}$ **break ;**
{ declaration-list$_{opt}$ statement-list$_{opt}$ **break ; }**

default-label-clause:

default-label default-clause

default-label:

default :

default-clause:

case-clause

and

statement:

/* labelled_statement removed */
compound_statement
expression_statement
selection_statement
iteration_statement
/* jump_statement removed or just restricted to return depending on other rules */

The following terms are also used within the text of the rules:

switch label	Either a case label or default label.
case clause	The code between any two switch labels.
default clause	The code between the default label and the end of the switch statement.
switch clause	Either a case clause or a default clause.

Implemented by QAC messages:

3234 Declarations precede the first label in this 'switch' construct.

Required 15.1 — A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

The scope of a case or default label shall be the compound statement, which is the body of a switch statement. All case clauses and the default clause shall be at the same scope.

Implemented by QAC messages:

2019 'Switch' label is located within a nested code block.

Required 15.2 — An unconditional break statement shall terminate every non-empty switch clause.

The last statement in every switch clause shall be a break statement, or if the switch clause is a compound statement, then the last statement in the compound statement shall be a break statement.

Reference

Koenig 22-24;

Implemented by QAC messages:

2003 The preceding 'switch' clause is not empty and does not end with a 'jump' statement. Execution will fall through.

2020 Final 'switch' clause does not end with an explicit 'jump' statement.

Required 15.3 — The final clause of a switch statement shall be the default clause.

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

See also

Rule 14.10

Implemented by QAC messages:

2002 No 'default' label found in this 'switch' statement.

2009 This 'default' label is not the final 'case' label within the 'switch' block.

Required 15.4 — A switch expression shall not represent a value that is effectively Boolean.

See 'Boolean expressions' in the glossary.

For example:

```
switch ( x == 0 )          /* not compliant - effectively Boolean */
{
   ...
```

Implemented by QAC messages:

0735 Using relational or logical operators in a 'switch' expression is usually a programming error.

Required 15.5 — Every switch statement shall have at least one case clause.

For example:

```
switch ( x )
{
    uint8_t var;        /* not compliant - decl before 1st case     */
case 0:
    a = b;
    break;              /* break is required here                   */
case 1:                 /* empty clause, break not required         */
case 2:
    a = c;              /* executed if x is 1 or 2                  */
    if ( a == b )
    {
        case 3:         /* Not compliant - case is not allowed here */
    }
    break;              /* break is required here                   */
case 4:
    a = b;              /* Not compliant - non empty drop through   */
case 5:
    a = c;
    break;
default:                /* default clause is required               */
    errorflag = 1;      /* should be non-empty if possible          */
    break;              /* break is required here, in case a
                           future modification turns this into a
                           case clause                              */
```

```
}
```

Implemented by QAC messages:

3315 This 'switch' statement contains only a single path - it is redundant.

---

16 Functions

Required 16.1 **Functions shall not be defined with a variable number of arguments.**

There are a lot of potential problems with this feature. Users shall not write additional functions that use a variable number of arguments. This precludes the use of use of stdarg.h, va_arg, va_start and va_end.

Reference Unspecified 15;Undefined 25,45,61,70-76;

Implemented by QAC messages:

1337 Function defined with a variable number of parameters.

Required 16.2 **Functions shall not call themselves, either directly or indirectly.**

This means that recursive function calls cannot be used in safety-related systems. Recursion carries with it the danger of exceeding available stack space, which can be a serious error. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

Implemented by QAC messages:

1520 Functions are indirectly recursive.
3670 Recursive call to function containing this call.

Required 16.3 **Identifiers shall be given for all of the parameters in a function prototype declaration.**

Names shall be given for all the parameters in the function declaration for reasons of compatibility, clarity and maintainability.

Implemented by QAC messages:

1335 Parameter identifiers missing in function prototype declaration.
1336 Parameter identifiers missing in declaration of a function type.

Required 16.4 **The identifiers used in the declaration and definition of a function shall be identical.**

Implemented by QAC messages:

1330 The parameter identifiers in this function declaration differ from those in a previous declaration.

Required 16.5 **Functions with no parameters shall be declared and defined with the parameter list void.**

Functions shall be declared with a return type (see Rule 8.2), that type being void if the function does not return any data. Similarly, if the function has no parameters, the parameter list shall be declared as void. Thus for example, a function, myfunc, which neither takes parameters nor returns a value would be declared as:

```
void myfunc( void );
```

See also Rule 8.2

Reference Koenig 59-62;

Implemented by QAC messages:

3001 Function has been declared with an empty parameter list.
3007 "void" has been omitted when defining a function with no parameters.

Required 16.6 **The number of arguments passed to a function shall match the number of parameters.**

This problem is completely avoided by the use of function prototypes - see Rule 8.1. This rule is retained since compilers may not flag this constraint error.

See also Rule 8.1

Reference

Undefined 22;

Implemented by QAC messages:

0422 [C] Function call contains fewer arguments than prototype specifies.
0423 [C] Function call contains more arguments than prototype specifies.
3319 [U] Function called with number of arguments which differs from number of parameters in definition.

Advisory 16.7

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

This rule leads to greater precision in the definition of the function interface. The const qualification should be applied to the object pointed to, not to the pointer, since it is the object itself that is being protected.

For example:

```
void myfunc( int16_t * param1, const int16_t * param2, int16_t * param3 )
/* param1: Addresses an object which is modified - no const
   param2: Addresses an object which is not modified - const required
   param3: Addresses an object which is not modified - const missing */
{
    *param1 = *param2 + *param3;
    return;
}
/* data at address param3 has not been changed,
   but this is not const therefore not compliant */
```

Implemented by QAC messages:

3673 The object addressed by the pointer parameter '%s' is not modified and so the pointer could be of type 'pointer to const'.

Required 16.8

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

This expression gives the value that the function returns. The absence of a return with an expression leads to undefined behaviour (and the compiler may not give an error).

Reference

Undefined 43;

Implemented by QAC messages:

0745 [U] 'return;' found in '%s()', which has been defined with a non-'void' return type.
2887 Function 'main' ends with an implicit 'return' statement.
2888 This function has been declared with a non-void 'return' type but ends with an implicit 'return ;' statement.
3113 [U] 'return' statement includes no expression but function '%s()' is implicitly of type 'int'.
3114 [U] Function '%s()' is implicitly of type 'int' but ends without returning a value.

Required 16.9

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

If the programmer writes:

```
if ( f )    /* not compliant - gives a constant non-zero value which is
  {            the address of f - use either f() or &f                */
    /* ... */
}
```

then it is not clear if the intent is to test if the address of the function is not NULL or that a call to the function f() should be made.

Reference

Koenig 24;

Implemented by QAC messages:

3635 Function identifier used as a pointer without a preceding & operator.

Required 16.10

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

A function (whether it is part of the standard library, a third party library or a user defined function) may provide some means of indicating the occurrence of an error. This may be via an error flag, some special return value or some other means. Whenever such a mechanism is provided by a function the calling program shall check for the indication of an error as soon as the function returns.

However, note that the checking of input values to functions is considered a more robust means of error prevention than trying to detect errors after the function has completed (see Rule 20.3). Note also that the use of errno (to return error information from

functions) is clumsy and should be used with care (see Rule 20.5).

See also Rule 20.3, Rule 20.5

Implemented by QAC messages:

3200 '%s' returns a value which is not being used.

---

17 Pointers and Arrays

Required 17.1 Pointer arithmetic shall only be applied to pointers that address an array or array element.

Addition and subtraction of integers (including increment and decrement) from pointers that do not point to an array or array element results in undefined behaviour.

See also Rule 21.1

Reference Undefined 30

Implemented by QAC messages:

2930 Constant: Computing an invalid pointer value.
2931 Definite: Computing an invalid pointer value.
2932 Apparent: Computing an invalid pointer value.

Required 17.2 Pointer subtraction shall only be applied to pointers that address elements of the same array.

Subtraction of pointers only gives well-defined results if the two pointers point (or at least behave as if they point) into the same array object.

See also Rule 21.1

Reference Undefined 31;

Implemented by QAC messages:

2771 Definite: These pointers address different objects.
2772 Apparent: These pointers address different objects.

Required 17.3 >, >=, <, <= shall not be applied to pointer types except where they point to the same array.

Attempting to make comparisons between pointers will produce undefined behaviour if the two pointers do not point to the same object. Note: it is permissible to address the next element beyond the end of an array, but accessing this element is not allowed.

Reference Undefined 33;

Implemented by QAC messages:

2771 Definite: These pointers address different objects.
2772 Apparent: These pointers address different objects.

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

Array indexing is the only acceptable form of pointer arithmetic, because it is clearer and hence less error prone than pointer manipulation. This rule bans the explicit calculation of pointer values. Array indexing shall only be applied to objects defined as an array type. Any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Pointers may go out of bounds of arrays or structures, or may even point to effectively arbitrary locations.

```
void my_fn( uint8_t * p1, uint8_t p2[] )
{
   uint8_t index = 0U;
   uint8_t * p3;
   uint8_t * p4;

   *p1 = 0U;
   p1 ++;       /* not compliant - pointer increment              */
   p1 = p1 + 5; /* not compliant - pointer increment              */
   p1[5] = 0U   /* not compliant - p1 was not declared as an array */
   p3 = &p1[5]; /* not compliant - p1 was not declared as an array */
   p2[0] = 0U;
   index ++;
   index = index + 5U;
```

```
   p2[index] = 0U; /* compliant     */
   p4 = &p2[5];    /* compliant     */
}

uint8_t a1[16];
uint8_t a2[16];

my_fn( a1, a2 );

my_fn( &a1[4], &a2[4] );

uint8_t a[10];
uint8_t * p;
p = a;
*( p+5 ) = 0U; /* not compliant */
p[5] = 0U;     /* not compliant */
```

See also

Rule 21.1

Implemented by QAC messages:

0488	Performing pointer arithmetic.
0489	The integer value 1 is being added or subtracted from a pointer.
0491	Array subscripting applied to an object of pointer type.
0492	Array subscripting applied to a function parameter declared as a pointer.

Advisory 17.5

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

Use of more than 2 levels of indirection can seriously impair the ability to understand the behaviour of the code, and should therefore be avoided.

```
typedef int8_t * INTPTR;

struct s
{
   int8_t *   s1; /* compliant     */
   int8_t **  s2; /* compliant     */
   int8_t *** s3; /* not compliant */
};

struct s *   ps1; /* compliant     */
struct s **  ps2; /* compliant     */
struct s *** ps3; /* not compliant */

int8_t **  (   *pfunc1 )(); /* compliant     */
int8_t **  (  **pfunc2 )(); /* compliant     */
int8_t **  ( ***pfunc3 )(); /* not compliant */
int8_t *** (  **pfunc4 )(); /* not compliant */

void function( int8_t *   par1,
               int8_t **  par2,
               int8_t *** par3,               /* not compliant */
               INTPTR *   par4,
               INTPTR *   const * const par5,  /* not compliant */
               int8_t *   par6[],
               int8_t **  par7[] )             /* not compliant */
{
   int8_t *   ptr1;
   int8_t **  ptr2;
   int8_t *** ptr3;               /* not compliant */
   INTPTR *   ptr4;
   INTPTR *   const * const ptr5; /* not compliant */
   int8_t *   ptr6[10];
   int8_t **  ptr7[10];
}
```

Explanation of types:

* par1 and ptr1 are of type pointer to int8_t.

* par2 and ptr2 are of type pointer to pointer to int8_t.

* par3 and ptr3 are of type pointer to a pointer to a pointer to int8_t. This is three levels and is not compliant.

* par4 and ptr4 are expanded to a type of pointer to a pointer to int8_t.

* par5 and ptr5 are expanded to a type of const pointer to a const pointer to a pointer to int8_t. This is three levels and is not compliant.

* par6 is of type pointer to pointer to int8_t because arrays are converted to a pointer to the initial element of the array.

* ptr6 is of type pointer to array of int8_t.

* par7 is of type pointer to pointer to pointer to int8_t because arrays are converted to a pointer to the initial element of the array. This is three levels and is not compliant.

* ptr7 is of type array of pointer to pointer to int8_t. This is compliant.

Implemented by QAC messages:

3260	Typedef defined with more than 2 levels of indirection.
3261	Member of struct/union defined with more than 2 levels of indirection.
3262	Object defined or declared with more than 2 levels of indirection.
3263	Function defined or declared with a return type which has more than 2 levels of indirection.

Required 17.6 — The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

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

For example

```
int8_t * foobar( void )
{
   int8_t local_auto;
   return &local_auto;      /* not compliant */
}
```

Reference

Undefined 9,26;

Implemented by QAC messages:

3217	Address of automatic object exported to a pointer with linkage or wider scope.
3225	Address of automatic object exported using a function parameter.
3230	Address of automatic object assigned to local pointer with static storage duration.
4140	Address of automatic object exported in function return value.

---

18 Structures and Unions

Required 18.1 — All structure and union types shall be complete at the end of a translation unit.

A complete declaration of the structure or union shall be included within any translation unit that reads from or writes to that structure. A pointer to an incomplete type is itself complete and is permitted, and therefore the use of opaque pointers is permitted. See section 6.1.2.5 of ISO/IEC 9899:1990 [2] for a full description of incomplete types.

```
struct tnode * pt;        /* tnode is incomplete at this point */

struct tnode
{
   int count;
   struct tnode *left;
   struct tnode * right;
};                        /* type tnode is now complete */
```

Reference

Undefined 35;

Implemented by QAC messages:

0544	[U] The value of an incomplete 'union' may not be used.
0545	[U] The value of an incomplete 'struct' may not be used.
0623	[U] '%s' has incomplete type and no linkage - this is undefined.
0636	[U] There are no named members in this 'struct' or 'union'.

Required 18.2 — An object shall not be assigned to an overlapping object.

The behaviour is undefined when two objects are created which have some overlap in memory and one is copied to the other.

Reference Undefined 34, 55;

Implemented by QAC messages:

2776 Definite: Copy between overlapping objects.
2777 Apparent: Copy between overlapping objects.

Required 18.3 **An area of memory shall not be reused for unrelated purposes.**

This rule refers to the technique of using memory to store some data, and then using the same memory to store unrelated data at some other time during the execution of the program. Clearly it relies on the two different pieces of data existing at disjoint periods of the program's execution, and never being required simultaneously.

This practice is not recommended for safety-related systems as it brings with it a number of dangers. For example: a program might try to access data of one type from the location when actually it is storing a value of the other type (e.g. due to an interrupt). The two types of data may align differently in the storage, and encroach upon other data. Therefore the data may not be correctly initialised every time the usage switches. The practice is particularly dangerous in concurrent systems.

However it is recognised that sometimes such storage sharing may be required for reasons of efficiency. Where this is the case it is essential that measures are taken to ensure that the wrong type of data can never be accessed, that data is always properly initialised and that it is not possible to access parts of other data (e.g. due to alignment differences). The measures taken shall be documented and justified in the deviation that is raised against this rule.

This might be achieved by the use of unions, or various other means.

Note that there is no intention in the MISRA-C guidelines to place restrictions on how a compiler actually translates source code as the user generally has no effective control over this. So for example, memory allocation within the compiler whether dynamic heap, dynamic stack or static, may vary significantly with only slight code changes even at the same level of optimisation. (Note also that some optimisation may legitimately take place even when the user makes no specific request for this.)

See also Rule 18.4

Required 18.4 **Unions shall not be used.**

Rule 18.3 prohibits the reuse of memory areas for unrelated purposes. However, even when memory is being reused for related purposes, there is still a risk that the data may be misinterpreted. Therefore, this rule prohibits the use of unions for any purpose.

It is recognised nonetheless that there are situations in which the careful use of unions is desirable in constructing an efficient implementation. In such situations, deviations to this rule are considered acceptable provided that all relevant implementation-defined behaviour is documented. This might be achieved in practice by referencing the implementation section of the compiler manuals from the design documentation. The kinds of implementation behaviour that might be relevant are:

* padding - how much padding is inserted at the end of the union

* alignment - how are members of any structures within the union aligned

* endianness - is the most significant byte of a word stored at the lowest or highest memory address

* bit-order - how are bits numbered within bytes and how are bits allocated to bit fields

The use of deviations is acceptable for (a) packing and unpacking of data, for example when sending and receiving messages, and (b) implementing variant records provided that the variants are differentiated by a common field. Variant records without a differentiator are not considered suitable for use in any situation.

See also Rule 3.5, Rule 18.3
Reference Implementation 27;

Implemented by QAC messages:

0750 A union type specifier has been defined.
0759 An object of union type has been defined.

---

19 Preprocessing Directives

Advisory 19.1 **#include statements in a file should only be preceded by other preprocessor directives or comments.**

All the #include statements in a particular code file should be grouped together near the head of the file. The rule states that the only items which may precede a #include in a file are other preprocessor directives or comments.

Implemented by QAC messages:

5087 #include statements in a file should only be preceded by other preprocessor directives or comments.

Advisory 19.2

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

If the ', \, ", or /* characters are used between < and > delimiters or the ', \, or /* characters are used between the " delimiters in a header name preprocessing token, then the behaviour is undefined. Use of the \ character is permitted in filename paths without the need for a deviation if required by the host operating system of the development environment.

Reference

Undefined 14;

Implemented by QAC messages:

0813 [U] Using any of the characters ' " or /* in '#include <%s>' gives undefined behaviour.
0814 [U] Using the characters ' or /* in '#include "%s"' gives undefined behaviour.
0831 [E] Use of '\\' in this '#include' line is a PC extension - this usage is non-portable.

Required 19.3

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

For example, the following are allowed.

```
#include "filename.h"
#include <filename.h>
#define FILE_A "filename.h"
#include FILE_A
```

Reference

Undefined 48;

Implemented by QAC messages:

0809 [U] The '#include' preprocessing directive has not been followed by <h-char-sequence> or "s-char-sequence".

Required 19.4

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

These are the only permitted uses of macros. Storage class specifiers and type qualifiers include keywords such as extern, static and const. Any other use of #define could lead to unexpected behaviour when substitution is made, or to very hard-to-read code.

In particular macros shall not be used to define statements or parts of statements except the use of the do-while construct. Nor shall macros redefine the syntax of the language. All brackets of whatever type ( ) { } [ ] in the macro replacement list shall be balanced.

The do-while-zero construct (see example below) is the only permitted mechanism for having complete statements in a macro body. The do-while-zero construct is used to wrap a series of one or more statements and ensure correct behaviour. Note: the semicolon must be omitted from the end of the macro body.

For example:

```
/* The following are compliant */
#define PI 3.14159F                 /* Constant                        */
#define XSTAL 10000000              /* Constant                        */
#define CLOCK ( XSTAL/16 )          /* Constant expression             */
#define PLUS2( X ) ( ( X ) + 2 )    /* Macro expanding to expression */
#define STOR extern                 /* storage class specifier         */
#define INIT( value ) { ( value ), 0, 0}   /* braced initialiser       */
#define CAT (PI)                    /* parenthesised expression        */
#define FILE_A "filename.h"         /* string literal                  */
#define READ_TIME_32() \
   do \
   { \
      DISABLE_INTERRUPTS(); \
      time_now = ( uint32_t)TIMER_HI << 16; \
      time_now = time_now | ( uint32_t )TIMER_LO; \
      ENABLE_INTERRUPTS(); \
   } while ( 0 )                    /* example of do-while-zero        */


/* the following are NOT compliant */
#define int32_t long       /* use typedef instead                      */
#define STARTIF if(        /* unbalanced () and language redefinition */
#define CAT PI             /* non-parenthesised expression             */
```

Reference

Koenig 82-84;

Implemented by QAC messages:

3409 The replacement list of function-like macro '%s' is not enclosed in ().
3411 Macro defined with unbalanced brackets, parentheses or braces.
3412 Macro defines an unrecognized code-fragment.
3413 Macro definition could be replaced by a typedef.
3431 Macro defines an operator, a punctuator or a control statement keyword.
3452 The replacement list of object-like macro '%s' is not enclosed in ().
3458 Macro defines a braced code statement block.
3460 Macro defines a type specifier keyword.
3461 Macro defines a storage-class specifier/type qualifier sequence.

Required 19.5 **Macros shall not be #define'd or #undef'd within a block.**

While it is legal C to place #define or #undef directives anywhere in a code file, placing them inside blocks is misleading as it implies a scope restricted to that block, which is not the case.

Normally, #define directives will be placed near the start of a file, before the first function definition. Normally, #undef directives will not be needed. (see Rule 19.6)

See also
Rule 19.6

Implemented by QAC messages:
0842 Using #define or #undef inside a function.

Required 19.6 **#undef shall not be used.**

#undef should not normally be needed. Its use can lead to confusion with respect to the existence or meaning of a macro when it is used in the code.

See also
Rule 19.5, Rule 20.1

Implemented by QAC messages:
0841 Using '#undef'.

Advisory 19.7 **A function should be used in preference to a function-like macro.**

While macros can provide a speed advantage over functions, functions provide a safer and more robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times.

Reference
Koenig 78-81;

Implemented by QAC messages:
3453 A function could probably be used instead of this function-like macro.

Required 19.8 **A function-like macro shall not be invoked without all of its arguments.**

This is a constraint error, but preprocessors have been known to ignore this problem. Each argument in a function-like macro must consist of at least one preprocessing token otherwise the behaviour is undefined.

Reference
Undefined 49;

Implemented by QAC messages:
0850 [C99] Macro argument is empty.
0856 [C] Fewer arguments in macro call than specified in definition.

Required 19.9 **Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.**

If any of the arguments act like preprocessor directives, the behaviour when macro substitution is made can be unpredictable.

Reference
Undefined 50;

Implemented by QAC messages:
0853 [U] Macro arguments contain a sequence of tokens that has the form of a preprocessing directive.

Required 19.10 **In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.**

Within a definition of a function-like macro, the arguments shall be enclosed in parentheses. For example define an abs function using:

```
#define abs( x ) ( ( ( x ) >= 0 ) ? ( x ) : -( x ) )
```

and not:

```
#define abs( x ) ( ( x >= 0 ) ? x : -x )
```

If this rule is not adhered to then when the preprocessor substitutes the macro into the code the operator precedence may not give the desired results.

Consider what happens if the second, incorrect, definition is substituted into the expression:

```
z = abs( a - b );
```

giving:

```
z = ( ( a - b >= 0 ) ? a - b : -a - b );
```

The sub-expression -a - b is equivalent to (-a)-b rather than -(a-b) as intended. Putting all the parameters in parentheses in the macro definition avoids this problem.

Reference

Koenig 78-81;

Implemented by QAC messages:

3410 Macro parameter not enclosed in ().

Required 19.11

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

If an attempt is made to use an identifier in a preprocessor directive, and that identifier has not been defined, the preprocessor will sometimes not give any warning but will assume the value zero. #ifdef, #ifndef and defined() are provided to test the existence of a macro, and are therefore excluded.

For example:

```
#if x < 0           /* x assumed to be zero if not defined */
```

Consideration should be given to the use of a #ifdef test before an identifier is used.

Note that preprocessing identifiers may be defined either by use of #define directives or by options specified at compiler invocation. However the use of the #define directive is preferred.

Implemented by QAC messages:

3332 The macro '%s' used in this '#if' or '#elif' expression is not defined.

Required 19.12

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

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

Reference

Unspecified 12;

Implemented by QAC messages:

0880 Using # and ## operators in the same macro definition.
0881 Using multiple ## operators in the same macro definition.
0884 Using multiple # operators in the same macro definition.

Advisory 19.13

The # and ## preprocessor operators should not be used.

There is an issue of unspecified order of evaluation associated with the # and ## preprocessor operators. Compilers have been inconsistent in the implementation of these operators. To avoid these problems do not use them.

Reference

Unspecified 12;

Implemented by QAC messages:

0341 Using the stringify operator '#'.
0342 Using the glue operator '##'.

Required 19.14 — The defined preprocessor operator shall only be used in one of the two standard forms.

The only two permissible forms for the defined preprocessor operator are:

```
defined( identifier )
defined identifier
```

Any other form leads to undefined behaviour, for example:

```
#if defined( X > Y )          /* not compliant - undefined behaviour */
```

Generation of the token defined during expansion of a #if or #elif preprocessing directive also leads to undefined behaviour and shall be avoided, for example:

```
#define DEFINED defined
#if DEFINED( X )              /* not compliant - undefined behaviour */
```

Reference

Undefined 47;

Implemented by QAC messages:

0885 [U] The token 'defined' is generated in the expansion of this macro.
0887 [U] Use of 'defined' must match either 'defined(identifier)' or 'defined identifier'.
0888 [U] 'defined' requires an identifier as an argument.

Required 19.15 — Precautions shall be taken in order to prevent the contents of a header file being included twice.

When a translation unit contains a complex hierarchy of nested header files it can happen that a particular header file is included more than once. This can be, at best, a source of confusion. If it leads to multiple or conflicting definitions, the result can be undefined or erroneous behaviour.

Multiple inclusions of a header file can sometimes be avoided by careful design. If this is not possible, a mechanism must be in place to prevent the file contents from being included more than once. A common approach is to associate a macro with each file; the macro is defined when the file is included for the first time and used subsequently when the file is included again to exclude the contents of the file.

For example a file called "ahdr.h" might be structured as follows:

```
#ifndef AHDR_H
#define AHDR_H
/* The following lines will be excluded by the
   preprocessor if the file is included more
   than once */

...

#endif
```

Alternatively, the following may be used.

```
#ifdef AHDR_H
#error Header file is already included
#else
#define AHDR_H
/* The following lines will be excluded by the
   preprocessor if the file is included more
   than once */

...

#endif
```

Implemented by QAC messages:

0883 Include file code is not protected against repeated inclusion

---

Required 19.16 — Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

When a section of source code is excluded by preprocessor directives, the content of each excluded statement is ignored until a #else, #elif or #endif directive is encountered (depending on the context). If one of these excluded directives is badly formed, it may be ignored without warning by a compiler with unfortunate consequences.

The requirement of this rule is that all preprocessor directives shall be syntactically valid even when they occur within an excluded block of code.

In particular, ensure that #else and #endif directives are not followed by any characters other than white-space. Compilers are not always consistent in enforcing this ISO requirement.

```
#define AAA 2
...
int foo( void )
{
    int x = 0;
    ...
#ifndef AAA
    x = 1;
#else1                      /* Not compliant */
    x = AAA;
#endif
    ...
    return x;
}
```

Implemented by QAC messages:

3115 Unrecognized preprocessing directive has been ignored because of conditional inclusion directives.

Required 19.17 **All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.**

When the inclusion and exclusion of blocks of statements is controlled by a series of preprocessor directives, confusion can arise if all of the relevant directives do not occur within one file. This rule requires that all preprocessor directives in a sequence of the form #if / #ifdef ... #elif ... #else ... #endif shall reside in the same file. Observance of this rule preserves good code structure and avoids maintenance problems.

Notice that this does not preclude the possibility that such directives may exist within included files so long as all directives that relate to the same sequence are located in one file.

file.c

```
#define A
...
#ifdef A
...
#include "file1.h"
#
#endif
...
#if 1
#include "file2.h"
...
EOF
```

file1.h

```
#if 1
...
#endif                      /* Compliant */
EOF
```

file2.h

```
...
#endif                      /* Not compliant */
```

Implemented by QAC messages:

3317 '#if...' not matched by '#endif' in included file. This is probably an error.

3318 '#else'/'#elif'/'#endif' in included file matched '#if...' in parent file. This is probably an error.

---

20 Standard Libraries

Required 20.1 — Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

It is generally bad practice to #undef a macro which is defined in the standard library. It is also bad practice to #define a macro name which is a C reserved identifier, a C keyword or the name of any macro, object or function in the standard library. For example, there are some specific reserved words and function names which are known to give rise to undefined behaviour if they are redefined or undefined, including defined, _ _LINE_ _, _ _FILE_ _, _ _DATE_ _, _ _TIME_ _, _ _STDC_ _, errno and assert.

See also Rule 19.6 regarding the use of #undef.

Reserved identifiers are defined by ISO/IEC 9899:1990 [2] Sections 7.1.3 "Reserved identifiers" and 6.8.8 "Predefined macro names". Macros in the standard library are examples of reserved identifiers. Functions in the standard library are examples of reserved identifiers. Any identifier in the standard library is considered a reserved identifier in any context i.e. at any scope or regardless of header files.

The defining, redefining or undefining of the reserved identifiers defined in 7.13 "Future library directions" is advisory.

Rule 20.1 applies regardless of which, if any, header files are included.

See also Rule 19.6

Reference Undefined 54,57,58,62;

Implemented by QAC messages:

0836	[U] Definition of macro named 'defined'.
0848	[U] Attempting to #undef '%s', which is a predefined macro name.
0854	[U] Attempting to #define '%s', which is a predefined macro name.
3439	Macro redefines a keyword.
4600	The macro '%1s' is also defined in '<%2s>'.
4601	The macro '%1s' is the name of an identifier in '<%2s>'.

Required 20.2 — The names of standard library macros, objects and functions shall not be reused.

Where new versions of standard library macros, objects or functions are used by the programmer (e.g. enhanced functionality or checks of input values) the modified macro, object or function shall have a new name. This is to avoid any confusion as to whether a standard macro, object or function is being used or whether a modified version of that function is being used. So, for example, if a new version of the sqrt function is written to check that the input is not negative, the new function shall not be named "sqrt", but shall be given a new name.

Implemented by QAC messages:

0602	[U] The identifier '%s' is reserved for use by the library.
4602	The identifier '%1s' is declared as a macro in '<%2s>'.
4603	The object/function '%1s'is being defined with the same name as an ordinary identifier defined in '<%2s>'.
4604	The object/function '%1s' is being declared with the same name as an ordinary identifier defined in '<%2s>'.
4605	The typedef '%1s' is also defined in '<%2s>'.
4606	The typedef '%1s' has the same name as another ordinary identifier in '<%2s>'.
4607	The enum constant '%1s' has the same name as another ordinary identifier in '<%2s>'.
4608	The tag '%1s' is also defined in '<%2s>'.

Required 20.3 — The validity of values passed to library functions shall be checked.

Many functions in the standard C libraries are not required by the ISO standard [2] to check the validity of parameters passed to them. Even where checking is required by the standard, or where compiler writers claim to check parameters, there is no guarantee that adequate checking will take place. Therefore the programmer shall provide appropriate checks of input values for all library functions which have a restricted input domain (standard libraries, other bought in libraries, and in-house libraries).

Examples of functions that have a restricted domain and need checking are:

* many of the maths functions in math.h, for example: negative numbers must not be passed to the sqrt or log functions; the second parameter of fmod should not be zero

* toupper and tolower: some implementations can produce unexpected results when the function toupper is passed a parameter which is not a lower case letter (and similarly for tolower)

* the character testing functions in ctype.h exhibit undefined behaviour if passed invalid values.

* the abs function applied to the most negative integer gives undefined behaviour.

Although most of the math library functions in math.h define allowed input domains, the values they return when a domain error occurs

may vary from one compiler to another. Therefore pre-checking the validity of the input values is particularly important for these functions.

The programmer should identify any domain constraints which should sensibly apply to a function being used (which may or may not be documented in the standard), and provide appropriate checks that the input value(s) lies within this domain. Of course, the value may be restricted further, if required, by knowledge of what the parameter represents and what constitutes a sensible range of values for the parameter.

There are a number of ways in which the requirements of this rule might be satisfied, including the following:

* Check the values before calling the function

* Design checks into the function. This is particularly applicable for in-house designed libraries, though it could apply to bought-in libraries if the supplier can demonstrate that they have built in the checks.

* Produce "wrapped" versions of functions, that perform the checks then call the original function.

* Demonstrate statically that the input parameters can never take invalid values.

Note that when checking a floating-point parameter to a function, that has a singularity at zero, it may be appropriate to perform a test of equality to zero. This will require a deviation to Rule 13.3. However it will usually still be necessary to test to a tolerance around zero (or any other singularity) if the magnitude of the value of the function tends to infinity as the parameter tends to zero, so as to prevent overflow occurring.

See also Rule 13.3, Rule 16.10, Rule 21.1

Reference Undefined 60,63;Implementation 45,47;

Implemented by QAC messages:

2810	Constant: Dereference of NULL pointer.
2811	Definite: Dereference of NULL pointer.
2812	Apparent: Dereference of NULL pointer.
2840	Constant: Dereference of an invalid pointer value.
2841	Definite: Dereference of an invalid pointer value.
2842	Apparent: Dereference of an invalid pointer value.

Required 20.4 **Dynamic heap memory allocation shall not be used.**

This precludes the use of the functions calloc, malloc, realloc and free.

There is a whole range of unspecified, undefined and implementation-defined behaviour associated with dynamic memory allocation, as well as a number of other potential pitfalls. Dynamic heap memory allocation may lead to memory leaks, data inconsistency, memory exhaustion, non-deterministic behaviour.

Note that some implementations may use dynamic heap memory allocation to implement other functions (for example functions in the library string.h). If this is the case then these functions shall also be avoided.

Reference Unspecified 19;Undefined 91,92;Implementation 69;Koenig 32;

Implemented by QAC messages:

5118	Dynamic heap memory allocation shall not be used.

Required 20.5 **The error indicator errno shall not be used.**

errno is a facility of C, which in theory should be useful, but which in practice is poorly defined by the standard. A non zero value may or may not indicate that a problem has occurred; as a result it shall not be used. Even for those functions for which the behaviour of errno is well defined, it is preferable to check the values of inputs before calling the function rather than rely on using errno to trap errors.

See also Rule 16.10

Reference Implementation 46, Koenig 73;

Implemented by QAC messages:

5119	The error indicator errno shall not be used.

Required 20.6 **The macro offsetof, in library <stddef.h>, shall not be used.**

Use of this macro can lead to undefined behaviour when the types of the operands are incompatible or when bit fields are used.

Reference Undefined 59;

Implemented by QAC messages:

5120 The macro offsetof, in library <stddef.h>, shall not be used.

Required 20.7 The setjmp macro and the longjmp function shall not be used.

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

Reference

Unspecified 14;Undefined 64-67, Koenig 74;

Implemented by QAC messages:

5122 The setjmp macro and the longjmp function shall not be used.

Required 20.8 The signal handling facilities of <signal.h> shall not be used.

Signal handling contains implementation-defined and undefined behaviour.

Reference

Undefined 68,69;Implementation 48-52;Koenig 74;

Implemented by QAC messages:

5123 The signal handling facilities of <signal.h> shall not be used.

Required 20.9 The input/output library <stdio.h> shall not be used in production code.

This includes file and I/O functions fgetpos, fopen, ftell, gets, perror, remove, rename, and ungetc.

Streams and file I/O have a large number of unspecified, undefined and implementation-defined behaviours associated with them. It is assumed within this standard that they will not normally be needed in production code in embedded systems.

If any of the features of stdio.h need to be used in production code, then the issues associated with the feature need to be understood.

Reference

Unspecified 2-5,16-18;Undefined 77-89;Implementation 53-68;

Implemented by QAC messages:

5124 The input/output library <stdio.h> shall not be used in production code.

Required 20.10 The library functions atof, atoi and atol from library <stdlib.h> shall not be used.

These functions have undefined behaviour associated with them when the string cannot be converted.

Reference

Undefined 90;

Implemented by QAC messages:

5125 The library functions atof, atoi and atol from library <stdlib.h> shall not be used.

Required 20.11 The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.

These functions will not normally be required in an embedded system, which does not normally need to communicate with an environment. If the functions are found necessary in an application, then it is essential to check on the implementation-defined behaviour of the function in the environment in question.

Reference

Undefined 93;Implementation 70-73;

Implemented by QAC messages:

5126 The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.

Required 20.12 The time handling functions of library <time.h> shall not be used.

Includes time, strftime. This library is associated with clock times. Various aspects are implementation dependent or unspecified, such as the formats of times. If any of the facilities of time.h are used then the exact implementation for the compiler being used must be determined, and a deviation raised.

Reference

Unspecified 22;Undefined 97;Implementation 75,76;

Implemented by QAC messages:

5127 The time handling functions of library <time.h> shall not be used.

---

21 Run-Time Failures

Required 21.1 **Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults**

Run-time checking is an issue, which is not specific to C, but it is an issue which C programmers need to pay special attention to. This is because the C language is weak in its provision of any run-time checking. C implementations are not required to perform many of the dynamic checks that are necessary for robust software. It is therefore an issue that C programmers need to consider carefully, adding dynamic checks to code wherever there is potential for run-time errors to occur.

Where expressions consist only of values within a well-defined range, a run-time check may not be necessary, provided it can be demonstrated that for all values within the defined range the exception cannot occur. Such a demonstration, if used, should be documented along with the assumptions on which it depends. However if adopting this approach, be very careful about subsequent modifications of the code which may invalidate the assumptions, or of the assumptions changing for any other reason.

See also Rule 17.1, Rule 17.2, Rule 17.4, Rule 20.3

Reference [Undefined 19,26,94];

Implemented by QAC messages:

2791	Definite: Right hand operand of shift operator is negative or too large.
2792	Apparent: Right hand operand of shift operator is negative or too large.
2801	Definite: Overflow in signed arithmetic operation.
2802	Apparent: Overflow in signed arithmetic operation.
2811	Definite: Dereference of NULL pointer.
2812	Apparent: Dereference of NULL pointer.
2821	Definite: Arithmetic operation on NULL pointer.
2822	Apparent: Arithmetic operation on NULL pointer.
2831	Definite: Division by zero.
2832	Apparent: Division by zero.
2841	Definite: Dereference of an invalid pointer value.
2842	Apparent: Dereference of an invalid pointer value.
2845	Constant: Maximum number of characters to be written is larger than the target buffer size.
2846	Definite: Maximum number of characters to be written is larger than the target buffer size.
2847	Apparent: Maximum number of characters to be written is larger than the target buffer size.
2871	Infinite loop identified.
2872	This loop, if entered, will never terminate.
2877	This loop will never be executed more than once.
2891	Definite: Negative value implicitly converted to an unsigned type.
2892	Apparent: Negative value implicitly converted to an unsigned type.
2896	Definite: Negative value cast to an unsigned type.
2897	Apparent: Negative value cast to an unsigned type.
2901	Definite: Positive integer value truncated by implicit conversion to a smaller unsigned type.
2902	Apparent: Positive integer value truncated by implicit conversion to a smaller unsigned type.
2911	Definite: Wraparound in unsigned arithmetic operation.
2912	Apparent: Wraparound in unsigned arithmetic operation.
2980	The value of this function parameter is never used before being modified.
2981	This initialization is redundant. The value of this object is never used before being modified.
2982	This assignment is redundant. The value of this object is never used before being modified.
2983	This assignment is redundant. The value of this object is never subsequently used.
2984	This operation is redundant. The value of the result is always '%1s'.
2985	This operation is redundant. The value of the result is always that of the left-hand operand.
2986	This operation is redundant. The value of the result is always that of the right-hand operand.

References

[1] MISRA Guidelines for the Use of the C Language In Vehicle Based Software, ISBN 0-9524159-9-0, Motor Industry Research Association, Nuneaton, April 1998

[2] ISO/IEC 9899:1990, Programming languages - C, ISO, 1990

[3] Hatton L., Safer C - Developing Software for High-integrity and Safety-critical Systems, ISBN 0-07-707640-0, McGraw-Hill, 1994

[4] ISO/IEC 9899:COR1:1995, Technical Corrigendum 1, 1995

[5] ISO/IEC 9899:AMD1:1995, Amendment 1, 1995

[6] ISO/IEC 9899:COR2:1996, Technical Corrigendum 2, 1996

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

[8] ISO/IEC 9899:1999, Programming languages - C, ISO, 1999

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

[10] CRR80, The Use of Commercial Off-the-Shelf (COTS) Software in Safety Related Applications, ISBN 0-7176-0984-7, HSE Books

[11] ISO 9001:2000, Quality management systems - Requirements, ISO, 2000

[12] ISO 90003:2004, Software engineering - Guidelines for the application of ISO 9001:2000 to computer software, ISO, 2004

[13] The TickIT Guide, Using ISO 9001:2000 for Software Quality Management System Construction, Certification and Continual Improvement, Issue 5, BSI, 2001

[14] Straker D., C. -Style: Standards and Guidelines, ISBN 0-13-116898-3, Prentice Hall 1991

[15] Fenton N.E. and Pfleeger S.L., Software Metrics: A Rigorous and Practical Approach, 2nd Edition, ISBN 0-534-95429-1, PWS, 1998

[16] MISRA Report 5 Software Metrics, Motor Industry Research Association, Nuneaton, February 1995

[17] MISRA Report 6 Verification and Validation, Motor Industry Research Association, Nuneaton, February 1995

[18] Kernighan B.W., Ritchie D.M., The C programming language, 2nd edition, ISBN 0-13-110362-8, Prentice Hall, 1988 (note: The 1st edition is not a suitable reference document as it does not describe ANSI/ISO C)

[19] Koenig A., C Traps and Pitfalls, ISBN 0-201-17928-8, Addison-Wesley, 1988

[20] IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, International Electromechanical Commission, in 7 parts published between 1998 and 2000

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

[22] ISO/IEC 10646:2003, Information technology - Universal Multiple-Octet Coded Character Set (UCS), 2003

[23] MISRA-C:2004 - Guidelines for the use of the C language in critical systems, ISBN 0 9524156 2 3, Motor Industry Research Association, October 2004

End of Coding Standard Manual