**Embedded C Coding Standard**

by Michael Barr

# General Rules

## Which C?

**Rules**:

1. All programs shall be written to comply with the C99 version of the ISO C

Programming Language Standard.[[1]](#footnote-1)

1. Whenever a C++ compiler is used, appropriate compiler options shall be set to restrict the language to the selected version of ISO C.
2. The use of proprietary compiler language keyword extensions, #pragma, and inline assembly shall be kept to the minimum necessary to get the job done and be localized to a small number of device driver modules that interface directly to hardware.
3. Preprocessor directive #define shall not be used to alter or rename any keyword or other aspect of the programming language.

**Example**:

#define begin { // Don’t do something like this...

#define end } // ... nor this.

... for (int row = 0; row < MAX\_ROWS; row++) begin

... end // Let C be C, not some language you once loved.

## Line Widths

**Rules**:

a. The width of all lines in a program shall be limited to a maximum of 80 characters.

## Braces

**Rules**:

1. Braces shall always surround the blocks of code (a.k.a., compound statements), following if, else, switch, *while*, do, and for statements; single statements and empty statements following these keywords shall also always be surrounded by braces.
2. Each left brace ({) shall appear by itself on the line below the start of the block it opens. The corresponding right brace (}) shall appear by itself in the same position the appropriate number of lines later in the file.

**Example**:

{ if (depth\_in\_ft > 10) dive\_stage = DIVE\_DEEP; // This is legal...

else if (depth\_in\_ft > 0) dive\_stage = DIVE\_SHALLOW; // ... as is this. else

{ // But using braces is always safer.

dive\_stage = DIVE\_SURFACE;

}

...

}

## Parentheses

**Rules**:

1. Do not rely on C’s operator precedence rules, as they may not be obvious to those who maintain the code. To aid clarity, use parentheses (and/or break long statements into multiple lines of code) to ensure proper execution order within a sequence of operations.
2. Unless it is a single identifier or constant, each operand of the logical AND (&&) and logical OR (||) operators shall be surrounded by parentheses.

**Example**:

if ((depth\_in\_cm > 0) && (depth\_in\_cm < MAX\_DEPTH))

{ depth\_in\_ft = convert\_depth\_to\_ft(depth\_in\_cm);

}

## Common Abbreviations

**Rules**:

1. Abbreviations and acronyms should generally be avoided unless their meanings are widely and consistently understood in the engineering community.
2. A table of project-specific abbreviations and acronyms shall be maintained in a version-controlled document.

**Example**: *Appendix A* contains a sample table of abbreviations and their meanings.

## Casts

**Rules**:

a. Each cast shall feature an associated comment describing how the code ensures proper behavior across the range of possible values on the right side.

**Example**:

int abs (int arg)

{

return ((arg < 0) ? -arg : arg);

}

... uint16\_t sample = adc\_read(ADC\_CHANNEL\_1);

result = abs((int) sample); // WARNING: 32-bit int assumed.

## Keywords to Avoid

**Rules**:

1. The auto keyword shall not be used.
2. The register keyword shall not be used.
3. It is a preferred practice to avoid all use of the goto keyword. If goto is used it shall only jump to a label declared later in the same or an enclosing block.
4. It is a preferred practice to avoid all use of the continue keyword.

## Keywords to Frequent

**Rules**:

1. The static keyword shall be used to declare all functions and variables that do not need to be visible outside of the module in which they are declared.
2. The const keyword shall be used whenever appropriate. Examples include:
   1. To declare variables that should not be changed after initialization,
   2. To define call-by-reference function parameters that should not be modified (e.g., char const \* param),
   3. To define fields in a struct or union that should not be modified (e.g., in a struct overlay for memory-mapped I/O peripheral registers), and
   4. As a strongly typed alternative to #define for numerical constants.
3. The volatile keyword shall be used whenever appropriate. Examples include:
   1. To declare a global variable accessible (by current use or scope) by any interrupt service routine,
   2. To declare a global variable accessible (by current use or scope) by two or more threads,
   3. To declare a pointer to a memory-mapped I/O peripheral register set (e.g., timer\_t volatile \* const p\_timer), and iv. To declare a delay loop counter.

**Example**:

typedef struct

{

uint16\_t count; uint16\_t max\_count;

uint16\_t const \_unused; // read-only register uint16\_t control;

} timer\_reg\_t;

timer\_reg\_t volatile \* const p\_timer = (timer\_reg\_t \*) HW\_TIMER\_ADDR;

# Comment Rules

## Acceptable Formats

**Rules**:

1. Single-line comments in the C++ style (i.e., preceded by //) are a useful and acceptable alternative to traditional C style comments (i.e., /\* … \*/).
2. Comments shall never contain the preprocessor tokens /\*, //, or \.
3. Code shall never be commented out, even temporarily.
   1. To temporarily disable a block of code, use the preprocessor’s conditional compilation feature (e.g., #if 0 … #endif).
   2. Any line or block of code that exists specifically to increase the level of debug output information shall be surrounded by #ifndef NDEBUG … #endif.

**Example**:

/\* The following code was meant to be part of the build...

...

safety\_checker(); ...

/\* ... but an end of comment character sequence was omitted. \*/

## Locations and Content

**Rules**:

1. All comments shall be written in clear and complete sentences, with proper spelling and grammar and appropriate punctuation.
2. The most useful comments generally precede a block of code that performs one step of a larger algorithm. A blank line shall follow each such code block.

The comments in front of the block should be at the same indentation level.

1. Avoid explaining the obvious. Assume the reader knows the C programming language. For example, end-of-line comments should only be used where the meaning of that one line of code may be unclear from the variable and function names and operations alone but where a short comment makes it clear. Specifically, avoid writing unhelpful and redundant comments, e.g., “numero <<= 2; // Shift numero left 2 bits.”.
2. The number and length of individual comment blocks shall be proportional to the complexity of the code they describe.
3. Whenever an algorithm or technical detail is defined in an external reference—e.g., a design specification, patent, or textbook—a comment shall include a sufficient reference to the original source to allow a reader of the code to locate the document.
4. Whenever a flow chart or other diagram is needed to sufficiently document the code, the drawing shall be maintained with the source code under version control and the comments should reference the diagram by file name or title.
5. All assumptions shall be spelled out in comments.[[2]](#footnote-2)
6. Each module and function shall be commented in a manner suitable for automatic documentation generation, e.g., via Doxygen.

1. Use the following capitalized comment markers to highlight important issues:
   1. “WARNING:” alerts a maintainer there is risk in changing this code. For example, that a delay loop counter’s terminal value was determined empirically and may need to change when the code is ported or the optimization level tweaked.
   2. “NOTE:” provides descriptive comments about the “why” of a chunk of code—as distinguished from the “how” usually placed in comments. For example, that a chunk of driver code deviates from the datasheet because there was an errata in the chip. Or that an assumption is being made by the original programmer.
   3. “TODO:” indicates an area of the code is still under construction and explains what remains to be done. When appropriate, an all-caps programmer name or set of initials may be included before the word TODO (e.g., “MJB TODO:”).

**Example**:

// Step 1: Batten down the hatches. for (int hatch = 0; hatch < NUM\_HATCHES; hatch++)

{ if (hatch\_is\_open(hatches[hatch]))

{ hatch\_close(hatches[hatch]);

}

}

// Step 2: Raise the mizzenmast.

// TODO: Define mizzenmast driver API.

# White Space Rules

## Spaces

**Rules**:

1. Each of the keywords if, while, for, switch, and return shall be followed by one space when there is additional program text on the same line.
2. Each of the assignment operators =, +=, -=, \*=, /=, %=, &=, |=, ^=, ~=, and != shall always be preceded and followed by one space.
3. Each of the binary operators +, -, \*, /, %, <, <=, >, >=, ==,!=, <<, >>, &, |, ^, &&, and || shall always be preceded and followed by one space.
4. Each of the unary operators +, -, ++, --, ! , and ~, shall be written without a space on the operand side.
5. The pointer operators \* and & shall be written with white space on each side within declarations but otherwise without a space on the operand side.
6. The ? and : characters that comprise the ternary operator shall each always be preceded and followed by one space.
7. The structure pointer and structure member operators (-> and ., respectively) shall always be without surrounding spaces.
8. The left and right brackets of the array subscript operator ([ and ]) shall be without surrounding spaces, except as required by another white space rule.
9. Expressions within parentheses shall always have no spaces adjacent to the left and right parenthesis characters.
10. The left and right parentheses of the function call operator shall always be without surrounding spaces, except that the function declaration shall feature one space between the function name and the left parenthesis to allow that one particular mention of the function name to be easily located.
11. Except when at the end of a line, each comma separating function parameters shall always be followed by one space.
12. Each semicolon separating the elements of a *for* statement shall always be followed by one space.
13. Each semicolon shall follow the statement it terminates without a preceding space.

## Alignment

**Rules**:

1. The names of variables within a series of declarations shall have their first characters aligned.
2. The names of struct and union members shall have their first characters aligned.
3. The assignment operators within a block of adjacent assignment statements shall be aligned.
4. The # in a preprocessor directive shall always be located at the start of a line, though the directives themselves may be indented within a #if or #ifdef sequence.

**Example**:

#ifdef USE\_UNICODE\_STRINGS

# define BUFFER\_BYTES 128

#else

# define BUFFER\_BYTES 64

#endif … typedef struct

{

uint8\_t buffer[BUFFER\_BYTES]; uint8\_t checksum;

} string\_t;

## Blank Lines

**Rules**:

1. No line of code shall contain more than one statement.
2. There shall be a blank line before and after each natural block of code. Examples of natural blocks of code are loops, if…else and switch statements, and consecutive declarations.
3. Each source file shall terminate with a comment marking the end of file followed by a blank line.

## Indentation

**Rules**:

1. Each indentation level should align at a multiple of 4 characters from the start of the line.
2. Within a switch statement, the case labels shall be aligned; the contents of each case block shall be indented once from there.
3. Whenever a line of code is too long to fit within the maximum line width, indent the second and any subsequent lines in the most readable manner possible.

**Example**:

sys\_error\_handler(int err)

{ switch (err)

{ case ERR\_THE\_FIRST:

... break; default:

... break;

}

// Purposefully misaligned indentation; see why?

if ((first\_very\_long\_comparison\_here

&& second\_very\_long\_comparison\_here)

|| third\_very\_long\_comparison\_here)

{

...

}

}

## Tabs

**Rules**:

a. The tab character (ASCII 0x09) shall never appear within any source code file.

**Example**:

// When tabs are needed inside a string, use the ‘\t’ character.

#define COPYRIGHT “Copyright (c) 2018 Barr Group.\tAll rights reserved.”

// When indents are needed in the source code, align via spaces instead.

void main (void)

{

// If not, you can encounter

// all sorts

// of weird and

// uneven

// alignment of code and comments... across tools.

}

## Non-Printing Characters

**Rules**:

1. Whenever possible, all source code lines shall end only with the single character ‘LF’ (ASCII 0x0A), not with the pair ‘CR’-‘LF’ (0x0D 0x0A).
2. The only other non-printable character permitted in a source code file is the form feed character ‘FF’ (ASCII 0x0C).

**Example**: It’s not possible to demonstrate non-printing characters in print.

# Module Rules

## Naming Conventions

**Rules**:

1. All module names shall consist entirely of lowercase letters, numbers, and underscores. No spaces shall appear within the module’s header and source file names.
2. All module names shall be unique in their first 8 characters and end with suffices .h and .c for the header and source file names, respectively.
3. No module’s header file name shall share the name of a header file from the C Standard Library or C++ Standard Library. For example, modules shall not be named “stdio” or “math”.
4. Any module containing a main() function shall have the word “main” as part of its source file name.

## Header Files

**Rules**:

1. There shall always be precisely one header file for each source file and they shall always have the same root name.
2. Each header file shall contain a preprocessor guard against multiple inclusion, as shown in the example below.[[3]](#footnote-3)
3. The header file shall identify only the procedures, constants, and data types (via prototypes or macros, #define, and typedefs, respectively) about which it is strictly necessary for other modules to be informed.
   1. It is a preferred practice that no variable ever be declared (via extern) in a header file.
   2. No storage for any variable shall be allocated in a header file.
4. No public header file shall contain a #include of any private header file.

**Example**:

#ifndef ADC\_H

#define ADC\_H ...

#endif /\* ADC\_H \*/

## Source Files

**Rules**:

1. Each source file shall include only the behaviors appropriate to control one “entity”. Examples of entities include encapsulated data types, active objects, peripheral drivers (e.g., for a UART), and communication protocols or layers (e.g., ARP).
2. Each source file shall be comprised of some or all of the following sections, in the order listed: comment block; include statements; data type, constant, and macro definitions; static data declarations; private function prototypes; public function bodies; then private function bodies.
3. Each source file shall always #include the header file of the same name (e.g., file adc.c should #include “adc.h”), to allow the compiler to confirm that each public function and its prototype match.
4. Absolute paths shall not be used in include file names.
5. Each source file shall be free of unused include files.
6. No source file shall #include another source file.

## File Templates

**Rules**:

a. A set of templates for header files and source files shall be maintained at the project level.

**Example**: See *Appendix B* and *Appendix C* for sample file templates.

# Data Type Rules

## Naming Conventions

**Rules**:

1. The names of all new data types, including structures, unions, and enumerations, shall consist only of lowercase characters and internal underscores and end with ‘\_t’.
2. All new structures, unions, and enumerations shall be named via a typedef.
3. The name of all public data types shall be prefixed with their module name and an underscore.

**Example**:

typedef struct

{ uint16\_t count; uint16\_t max\_count; uint16\_t \_unused; uint16\_t control;

} timer\_reg\_t;

## Fixed-Width Integers

**Rules**:

1. Whenever the width, in bits or bytes, of an integer value matters in the program, one of the fixed width data types shall be used in place of char, short, int, long, or long long. The signed and unsigned fixed-width integer types shall be as shown in the table below.

|  |  |  |
| --- | --- | --- |
| **Integer Width** | **Signed Type** | **Unsigned Type** |
| 8 bits | int8\_t | uint8\_t |
| 16 bits | int16\_t | uint16\_t |
| 32 bits | int32\_t | uint32\_t |
| 64 bits | int64\_t | uint64\_t |

1. The keywords short and long shall not be used.
2. Use of the keyword char shall be restricted to the declaration of and operations concerning strings.

## Signed and Unsigned Integers

**Rules**:

1. Bit-fields shall not be defined within signed integer types.
2. None of the bitwise operators (i.e., &, |, ~, ^, <<, and >>) shall be used to manipulate signed integer data.
3. Signed integers shall not be combined with unsigned integers in comparisons or expressions. In support of this, decimal constants meant to be unsigned should be declared with a ‘u’ at the end.

**Example**:

uint16\_t unsigned\_a = 6u; int16\_t signed\_b = -9;

if (unsigned\_a + signed\_b < 4)

{

// Execution of this block appears reliably logical, as -9 + 6 is -3 ...

}

// ... but compilers with 16-bit int may legally perform (0xFFFF – 9) + 6.

## Floating Point

**Rules**:

1. Avoid the use of floating point constants and variables whenever possible.

Fixed-point math may be an alternative.

1. When floating point calculations are necessary:

i. Use the C99 type names float32\_t, float64\_t, and float128\_t. ii. Append an ‘f’ to all single-precision constants (e.g., pi = 3.141592f).

* 1. Ensure that the compiler supports double precision, if your math depends on it.
  2. Never test for equality or inequality of floating point values.
  3. Always invoke the isfinite() macro to check that prior calculations have resulted in neither INFINITY nor NAN.

**Example**:

#include <limits.h>

#if (DBL\_DIG < 10) // Ensure the compiler supports double precision.

# error “Double precision is not available!”

#endif

## Structures and Unions

**Rules**:

1. Appropriate care shall be taken to prevent the compiler from inserting padding bytes within struct or union types used to communicate to or from a peripheral or over a bus or network to another processor.
2. Appropriate care shall be taken to prevent the compiler from altering the intended order of the bits within bit-fields.

**Example**:

typedef struct

{ uint16\_t count; // offset 0 uint16\_t max\_count; // offset 2 uint16\_t \_unused; // offset 4

uint16\_t enable : 2; // offset 6 bits 15-14 uint16\_t b\_interrupt : 1; // offset 6 bit 13 uint16\_t \_unused1 : 7; // offset 6 bits 12-6 uint16\_t b\_complete : 1; // offset 6 bit 5 uint16\_t \_unused2 : 4; // offset 6 bits 4-1 uint16\_t b\_periodic : 1; // offset 6 bit 0

} timer\_reg\_t;

// Preprocessor check of timer register layout byte count.

#if ((8 != sizeof(timer\_reg\_t))

# error “timer\_reg\_t struct size incorrect (expected 8 bytes)”

#endif

## Booleans

**Rules**:

1. Boolean variables shall be declared as type bool*.*
2. Non-Boolean values shall be converted to Boolean via use of relational operators (e.g., < or !=), not via casts.

**Example**:

#include <stdbool.h> ... bool b\_in\_motion = (0 != speed\_in\_mph);

# Procedure Rules

## Naming Conventions

**Rules**:

1. No procedure shall have a name that is a keyword of any standard version of the C or C++ programming language. Restricted names include interrupt, inline, class, true, false, public, private, friend, protected, and many others.
2. No procedure shall have a name that overlaps a function in the C Standard Library. Examples of such names include strlen, atoi, and memset.
3. No procedure shall have a name that begins with an underscore.
4. No procedure name shall be longer than 31 characters.
5. No function name shall contain any uppercase letters.
6. No macro name shall contain any lowercase letters.
7. Underscores shall be used to separate words in procedure names.
8. Each procedure’s name shall be descriptive of its purpose. Note that procedures encapsulate the “actions” of a program and thus benefit from the use of verbs in their names (e.g., adc\_read()); this “noun-verb” word ordering is recommended. Alternatively, procedures may be named according to the question they answer (e.g., led\_is\_on()).
9. The names of all public functions shall be prefixed with their module name and an underscore (e.g., sensor\_read()).

## Functions

**Rules**:

1. All reasonable effort shall be taken to keep the length of each function limited to one printed page, or a maximum of 100 lines.
2. Whenever possible, all functions shall be made to start at the top of a printed page, except when several small functions can fit onto a single page.[[4]](#footnote-4)
3. It is a preferred practice that all functions shall have just one exit point and it shall be via a return at the bottom of the function.
4. A prototype shall be declared for each public function in the module header file.
5. All private functions shall be declared static*.*
6. Each parameter shall be explicitly declared and meaningfully named.

**Example**:

int

state\_change (int event)

{

int result = ERROR;

if (EVENT\_A == event)

{ result = STATE\_A;

} else { result = STATE\_B;

} return (result); }

## Function-Like Macros

**Rules**:

1. Parameterized macros shall not be used if a function can be written to accomplish the same behavior.
2. If parameterized macros are used for some reason, these rules apply:

i. Surround the entire macro body with parentheses. ii. Surround each use of a parameter with parentheses.

* 1. Use each parameter no more than once, to avoid unintended side effects.
  2. Never include a transfer of control (e.g., return keyword).

**Example**:

// Don’t do this ...

#define MAX(A, B) ((A) > (B) ? (A) : (B))

// ... when you can do this instead. inline int max(int num1, int num2)

## Threads of Execution

**Rules**:

a. All functions that encapsulate threads of execution (a.k.a., tasks, processes) shall be given names ending with “\_thread” (or “\_task”, “\_process”).

**Example**:

void alarm\_thread (void \* p\_data)

{

alarm\_t alarm = ALARM\_NONE; int err = OS\_NO\_ERR;

for (;;) { alarm = OSMboxPend(alarm\_mbox, &err); // Process alarm here.

}

}

## Interrupt Service Routines

**Rules**:

1. Interrupt service routines (ISRs) are not ordinary functions. The compiler must be informed that the function is an ISR by way of a #pragma or compiler-specific keyword, such as “\_\_interrupt”.
2. All functions that implement ISRs shall be given names ending with “\_isr”.
3. To ensure that ISRs are not inadvertently called from other parts of the software (they may corrupt the CPU and call stack if this happens), each ISR function shall be declared static and/or be located at the end of the associated driver module as permitted by the target platform.
4. A stub or default ISR shall be installed in the vector table at the location of all unexpected or otherwise unhandled interrupt sources. Each such stub could attempt to disable future interrupts of the same type, say at the interrupt controller, and assert().

**Example**:

#pragma irq\_entry void timer\_isr (void)

{ uint8\_t static prev = 0x00; // prev button states uint8\_t curr = \*gp\_button\_reg; // curr button states

// Compare current and previous button states.

g\_debounced |= (prev & curr); // record all closes g\_debounced &= (prev | curr); // record all opens

// Save current pin states for next interrupt prev = curr;

// Acknowledge timer interrupt at hardware, if necessary.

}

# Variable Rules

## Naming Conventions

**Rules**:

1. No variable shall have a name that is a keyword of C, C++, or any other well-known extension of the C programming language, including specifically K&R C and C99. Restricted names include interrupt, inline, restrict, class, true, false, public, private, friend, and protected.
2. No variable shall have a name that overlaps with a variable name from the C Standard Library (e.g., errno).
3. No variable shall have a name that begins with an underscore.
4. No variable name shall be longer than 31 characters.
5. No variable name shall be shorter than 3 characters, including loop counters.
6. No variable name shall contain any uppercase letters.
7. No variable name shall contain any numeric value that is called out elsewhere, such as the number of elements in an array or the number of bits in the underlying type.
8. Underscores shall be used to separate words in variable names.
9. Each variable’s name shall be descriptive of its purpose.
10. The names of any global variables shall begin with the letter ‘g’.

For example, g\_zero\_offset.

1. The names of any pointer variables shall begin with the letter ‘p’.

For example, p\_led\_reg.

1. The names of any pointer-to-pointer variables shall begin with the letters ‘pp’.

For example, pp\_vector\_table.

1. The names of all integer variables containing Boolean information (including 0 vs. non-zero) shall begin with the letter ‘b’ and phrased as the question they answer. For example, b\_done\_yet or b\_is\_buffer\_full.
2. The names of any variables representing non-pointer handles for objects, e.g., file handles, shall begin with the letter ‘h’. For example, h\_input\_file.
3. In the case of a variable name requiring multiple of the above prefixes, the order of their inclusion before the first underscore shall be [g][p|pp][b|h].

## Initialization

**Rules**:

1. All variables shall be initialized before use.
2. It is preferable to define local variables as you need them, rather than all at the top of a function.
3. If project- or file-global variables are used, their definitions shall be grouped together and placed at the top of a source code file.
4. Any pointer variable lacking an initial address shall be initialized to NULL.

**Example**:

uint32\_t g\_array[NUM\_ROWS][NUM\_COLS] = { ... }; ... for (int col = 0; col < NUM\_COLS; col++)

{ g\_array[row][col] = ...;

}

# Statement Rules

## Variable Declarations

**Rules**:

a. The comma operator (,) shall not be used within variable declarations.

**Example**:

char \* x, y; // Was y intended to be a pointer also? Don’t do this.

## Conditional Statements

**Rules**:

1. It is a preferred practice that the shortest (measured in lines of code) of the if and else if clauses should be placed first.
2. Nested if…else statements shall not be deeper than two levels. Use function calls or switch statements to reduce complexity and aid understanding.
3. Assignments shall not be made within an if or else if test.
4. Any if statement with an else if clause shall end with an else clause.[[5]](#footnote-5)

**Example**:

if (NULL == p\_object)

{ result = ERR\_NULL\_PTR;

} else if (p\_object = malloc(sizeof(object\_t))) // No assignments!

{

... } else

{

// Normal processing steps, which require many lines of code.

...

}

## Switch Statements

**Rules**:

1. The break for each case shall be indented to align with the associated case, rather than with the contents of the case code block.
2. All switch statements shall contain a default block.
3. Any case designed to fall through to the next shall be commented to clearly explain the absence of the corresponding break.

**Example**:

switch (err)

{ case ERR\_A:

... break; case ERR\_B:

...

// Also perform the steps for ERR\_C.

case ERR\_C:

... break; default:

... break;

}

## Loops

**Rules**:

1. Magic numbers shall not be used as the initial value or in the endpoint test of a while, do…while, or for loop.[[6]](#footnote-6)
2. With the exception of the initialization of a loop counter in the first clause of a for statement and the change to the same variable in the third, no assignment shall be made in any loop’s controlling expression.
3. Infinite loops shall be implemented via controlling expression for (;;).[[7]](#footnote-7)
4. Each loop with an empty body shall feature a set of braces enclosing a comment to explain why nothing needs to be done until after the loop terminates.

**Example**:

// Why would anyone bury a magic number (e.g., “100”) in their code?

for (int row = 0; row < 100; row++)

{

// Descriptively-named constants prevent defects and aid readability.

for (int col = 0; col < NUM\_COLS; col++)

{

...

}

## Jumps

**Rules**:

1. The use of goto statements shall be restricted as per Rule 1.7.c.
2. C Standard Library functions abort(), exit(), setjmp(), and longjmp() shall not be used.

## Equivalence Tests

**Rules**:

a. When evaluating the equality of a variable against a constant, the constant shall always be placed to the left of the equal-to operator (==).

**Example**:

if (NULL == p\_object)

{

return (ERR\_NULL\_PTR);

}

1. [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. [↑](#footnote-ref-5)
6. [↑](#footnote-ref-6)
7. [↑](#footnote-ref-7)