**UNIVERSIDAD TECNOLÓGICA DE QUERÉTARO**

**CESEQ**



**Diplomado en Software Embebido**

Proyecto Integrador Control de velocidad de un Motor

9.1. Software Standards

DOCUMENT: Software Standards

Document No: #CESEQ9.1

Scrum Master: Guarneros, Héctor

Developer. Govea, Francisco

Date (YYYYMMDD): 20190405

Version: 1.0.0.

Project Version: x.x.x.

# Log

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date (yyyymmdd) | Description | Reviewer |
| 1.0.0 | 20190802 | Added Misra-C:2004 Standards description | Guarneros, Héctor |
|  |  |  |  |

Contenido

[1. Log 2](#_Toc18358117)

[1. MISRA-C: 2004 4](#_Toc18358118)

[2. Referencias 8](#_Toc18358119)

# MISRA-C: 2004

Normative text is provided for each item or group of related items. In order to conform with

MISRA-C:2004, it is necessary to meet the requirements of this normative text.

The normative text gives, where appropriate, some explanation of the underlying issues being addressed by the rule(s), and examples of how to apply the rule(s). If there is no explanatory text immediately following a rule then the relevant text will be found following the group of rules, and applies to all the rules which precede it. Similarly a source reference following a group of rules applies to the whole group.

The normative text is not intended as a tutorial in the relevant language feature, as the reader is assumed to have a working knowledge of the language. Further information on the language features can be obtained by consulting the relevant section of the language standard or other C language reference books. Where a source reference is given for one or more of the “Annex G” items in the ISO standard, then the original issue raised in the ISO standard may provide additional help in understanding the rule.

Within the rules and their normative text, the following font styles are used to represent C keywords and C code:

C *keywords* appear in italic text

C code appears in a monospaced font, either within other text or as separate code fragments;

Note that where code is quoted, the fragments may be incomplete (for example an *if* statement without its body). This is for the sake of brevity.

In code fragments, the following *typedef*’d types have been assumed (to comply with Rule 6.3):

char\_t plain 8 bit character.

uint8\_t unsigned 8 bit integer.

uint16\_t unsigned 16 bit integer.

uint32\_t unsigned 32 bit integer

int8\_t signed 8 bit integer

int16\_t signed 16 bit integer

int32\_t signed 32 bit integer

float32\_t 32 bit floating-point

float64\_t 64 bit floating-point

Non-specific variable names are constructed to give an indication of the type. For example:

uint8\_t u8a;

sint32\_t s32a;

**Rule 1.5 (advisory): 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 recognized standard.

An example of an appropriate standard is ANSI/IEEE Std 754 [21].

The definition of the floating‑point types, in accordance with Rule 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;

**Rule 2.2 (required): Source code shall only use** /\* **…** \*/ **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.

**Rule 6.3 (advisory): *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 document. 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.

**Rule 8.2 (required): 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 \*/

**Rule 8.8 (required): An external object or function shall be declared in one and only one file.**

[Koenig 66]

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.

**Rule 9.1 (required): All automatic variables shall have been assigned a value before being used.**

[Undefined 41]

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.

**Rule 10.5 (required): 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 orderbits 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.

**Rule 19.3 (required): 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

**Rule 19.4 (required): 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 \*/

# Referencias

C11 - <https://www.iso.org/standard/57853.html>

MISRA C - <https://www.misra.org.uk/>