C Language

Coding Standard

Contents

[1 Purpose and scope 4](#_Toc331016857)

[2 Rules 4](#_Toc331016858)

[2.1 Doxygen shall be used 4](#_Toc331016859)

[2.2 All files are modules 4](#_Toc331016860)

[2.3 Module format 4](#_Toc331016861)

[2.4 Global names within modules 4](#_Toc331016862)

[2.5 Module initialization and deinitialization 5](#_Toc331016863)

[2.6 Module heading 5](#_Toc331016864)

[2.7 Function heading 5](#_Toc331016865)

[2.8 Include guards 6](#_Toc331016866)

[2.9 Prefix comments 6](#_Toc331016867)

[2.10 Indentation 6](#_Toc331016868)

[2.11 Code blocks 7](#_Toc331016869)

[2.12 Follow on statements or code 7](#_Toc331016870)

[2.13 Commented out code 7](#_Toc331016871)

[2.14 80 character limit 7](#_Toc331016872)

[2.15 Character string line breaks 8](#_Toc331016873)

[2.16 Spacer lines 8](#_Toc331016874)

[2.17 Single entry/exit functions 9](#_Toc331016875)

[2.18 Return only on result returning functions 9](#_Toc331016876)

[2.19 No use of continue 9](#_Toc331016877)

[2.20 Use of break for case statements only 9](#_Toc331016878)

[2.21 No gotos 9](#_Toc331016879)

[2.22 Longjmp/setjmp 10](#_Toc331016880)

[2.23 Code and globals in header files 10](#_Toc331016881)

[2.24 Use of header files 10](#_Toc331016882)

[2.25 Size of functions 10](#_Toc331016883)

[2.26 No declarations on inner blocks 10](#_Toc331016884)

[2.27 Use automatic inlining 10](#_Toc331016885)

[2.28 No complex macros 10](#_Toc331016886)

[2.29 No specific inlining 11](#_Toc331016887)

[2.30 No register specifiers 11](#_Toc331016888)

[2.31 No initializations 11](#_Toc331016889)

[2.32 No reliance on initializations 11](#_Toc331016890)

[2.33 No macros 11](#_Toc331016891)

[2.34 No case dependence 11](#_Toc331016892)

[2.35 Lower case only 11](#_Toc331016893)

[2.36 Use break character to divide segmented names 11](#_Toc331016894)

[2.37 No unique namespace dependence 12](#_Toc331016895)

[2.38 Void for void parameter lists 12](#_Toc331016896)

[2.39 Only standard C 12](#_Toc331016897)

[2.40 No reliance on undefined or implementation specific behavior 12](#_Toc331016898)

[2.41 No asm statements 12](#_Toc331016899)

[2.42 English comments 12](#_Toc331016900)

[2.43 ISO 8859-1 character set 12](#_Toc331016901)

[2.44 Use “coined tag” structure declaractions 12](#_Toc331016902)

[2.45 No use of structure names 13](#_Toc331016903)

[2.46 Use explicit initializers 13](#_Toc331016904)

[2.47 No use of “=” in enumerations 13](#_Toc331016905)

[2.48 Do not ignore function returns 13](#_Toc331016906)

[2.49 Use of #ifdef and other macro process statements 14](#_Toc331016907)

[2.50 Comment coupiously 14](#_Toc331016908)

[2.51 Use modular structure 14](#_Toc331016909)

[2.52 Use asserts constantly 14](#_Toc331016910)

[2.53 Do not use asserts for runtime faults 15](#_Toc331016911)

[2.54 Use dynamic allocation instead of static 15](#_Toc331016912)

[2.55 Code reviews 15](#_Toc331016913)

[2.56 Source control 15](#_Toc331016914)

[2.57 Hanging vs. pinned blocks 15](#_Toc331016915)

[2.58 Run-on vs blocked code 16](#_Toc331016916)

[2.59 Use natural definition order 16](#_Toc331016917)

# Purpose and scope

This document contains the Moore/CAD coding standard for C based programs. This standard is to be followed for all Moore/CAD projects.

This standard uses MISRA as a guideline:

<http://www.ristancase.com/html/dac/manual/3.12.02%20MISRA-C%202004%20Rules.html>

But does not implement all of its recommendations. Nor in fact do we agree with them all. For example, there is no restrictions, nor in our opinion need to be, against unions. Similarly, MISRA forbids recursion and dynamic memory allocation, and that is not a problem here. There are other examples.

# Rules

## Doxygen shall be used

All comments within the source code will be doxygen compatible, as outlined here.

All globals, functions, parameters to functions, and variables declared within functions shall bear a doxygen comment.

## All files are modules

All files are considered modules, including header files. Each file shall have the module format.

## Module format

Each module shall have the following format:

1. The module comment header.
2. Includes.
3. External definitions.
4. Constants (defines or const).
5. Complex defintions.
6. Global variables.
7. Forwarded definitions.
8. Function definitions.
9. Initialize and deinitialize functions.

The sections will appear in the order shown.

## Global names within modules

All globals (not “static” variables and functions), shall be coined to the module. That is, for module “foo”, the names of functions and variables in the module shall be:

foo\_name

If the name is not exported, the rule does not apply. However, in that case the local variable or function should be declared “static”.

The “main” procedure is an exception to this rule.

Modules that are used system wide, and not just modules used within the target program, are excepted from the rule. For example, “printf” is in the system library, and does not need coining.

## Module initialization and deinitialization

If a module needs its globals initialized, it shall have the function:

foo\_init

At the end of the file. The initialization functions within each module are called during the startup of the main program.

If a module needs its globals deinitialized, is shall have the function:

foo\_deinit

At the end of the file, after the initialization function.

If a deinitialization function exists, then so shall an initialization function, even if that function is empty.

Header modules or modules without globals do not need initialize or deinitialize functions.

## Module heading

Each module shall have a heading without exception. The heading shall appear:

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Module FOO

Description

Description of module and purpose.

Exports

List of exported functions and variables. These consistute the

Interface for the module.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## Function heading

Each function shall have a heading unless:

1. It is in a group of related functions that vary from each other trivially, and a header exists for all of them.
2. It is a function only used within another function (it is essentially nested within that function). In this case it shall appear after the function heading comment for the including function and be documented by that function heading.

The function heading shall appear as:

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function FOO

Description

Description of function and purpose.

Returns

Description of the return value, or “none” if the function is

void.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

The parameters to the function shall be prefixed with doxygen comments. For example:

void foo(/\*\* This is the first parameter \*/ param1)

{

}

## Include guards

Header files (.h) shall have “include guards” of the form:

#ifndef \_\_FOO\_H\_\_

#define \_\_FOO\_H\_\_

Statements in header

#endif

## Prefix comments

All comments for:

* Variables (global or local)
* Functions
* Constants
* Defines
* Function parameters

Shall be prefixed commented in doxygen format (/\*\* xxx \*/).

## Indentation

Indentation shall be done using 4 (four) spaces, as in the whitebook [The C Programming language K&R]. No tabs are ever to be used in source code. There are no exceptions.

## Code blocks

All code blocks are indented:

if (x) {

Perform(y);

}

All code blocks will start with a blank line and end with a blank line.

## Follow on statements or code

All statements or code that follows other code shall be indented. If the follow on is a statement, then it shall be indented 4 spaces:

If (x)

perform(y)

If the follow on code is a parameter or expression part, it shall be indented FOLLOWING THE PARENTHSIS THAT STARTED IT:

If (x ||

y ||

z)

Perform(y)

or

perform(x,

y,

z);

## Commented out code

The use of “#if 0”, “if (0)”, or comments, used to remove source code from being compiled, WILL NOT BE DONE. There are no exceptions. If code may or may not be used according to a configuration option, then use of a constant if statement is acceptable.

## 80 character limit

A limit of 80 characters shall apply to all source files. No line shall exceed 80 characters in width, no exceptions. Statements shall be broken into sections and indented as required. Strings are broken into concatenated sections such as:

Printf(“this is a very”

“long string”);

(note that ANSI C rules state that successive strings are concatentated during preprocessing).

## Character string line breaks

The force character (“\”) shall not be used to carry a string into the next line. Use the fact that multiple successive strings are concatenated instead.

## Spacer lines

A blank line shall appear:

1. Before and after each function heading (the name, return specification and the parameters).
2. Before and after each block bracket (“{“, and “}”).
3. Before the return statement in a function.
4. After the function declaration section and before the code in the block appears.
5. After each function block ends.

This program illustrates the formatting:

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function main

Description

This is the main function of the program.

Returns

0 if all is well, otherwise != 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main (/\*\* Number of parameters \*/ argc,

/\*\* Parameter array \*/ argv)

{

int i;

printf(“hello, world\n”);

if (argc > 0) {

printf(“there are arguments\n”);

ior (i = 0; i < 10; i++) printf(“i: %d\n”, i);

}

return 0;

}

## Single entry/exit functions

Each function has only one entry at the top, and one exit at the bottom, no exceptions. The return statement shall never appear anywhere but the end of the function.

## Return only on result returning functions

A return statement shall appear in a function only if the function returns a result (is not void).

## No use of continue

The continue statement is never to be used, no exceptions. The effect of continue is achieved by carrying the flow of control to the end of the loop and placing the condition in the containing loop. Example:

while (x > 1) {

if (x == 10) continue;

if (x == 5) {

/\* perform statements \*/

}

}

Should be written:

while (x > 1) {

if (x != 10) {

if (x == 5) {

/\* perform statements \*/

}

}

Structured statements are single entry single exit just as functions are.The use of a continue essentially places an early exit to the structured loop statement. This style of programming takes getting used to, but soon becomes second nature, and results in dramatically more readable code.

## Use of break for case statements only

Break shall only be used to exit from a case clause in a switch statement. It will not be used to exit any other structured statement.

## No gotos

The goto statement will not be used. There are no exceptions. If you think you have to have a goto in a given function, this usually indicates your function is too large and needs to be broken into smaller functions.

## Longjmp/setjmp

Use of longjmp/setjmp is to be depreciated. The preferred use is to use an error return from each inner function until the outter function is reached. The use of longjmp/setjmp is to be reserved for when no other construct will do the same job.

## Code and globals in header files

No code or globals are to appear in header files. No exceptions.

## Use of header files

One header module must appear for each code module. It contains all of the global functions in that module, and may contain other global definitions. The header file for the module must be included by that module.

## Size of functions

Functions should be no more than one or two pages long, where a page is defined as about 60 lines long. If the function is longer than that, there should be a good fundamental reason for it, such as needing to express a unusually large case statement. A CPU emulator routine would be an example of such a routine.

## No declarations on inner blocks

Inner blocks are not to have declarations. This code is incorrect:

void foo(void)

{

{

int i;

for (i = 0; i < 10; i++) dothis();

}

}

If you need to create such an inner block, it typically indicates you need to break the code in the block into a separate function.

## Use automatic inlining

Use small functions when that makes the code clear, even if they are only called once. Find out how to set up automatic inlining and use it.

## No complex macros

Do not use complex macros. Examples include using expressions with any operand but other constants. Do not place a function in a macro, that is what automatic inlining is for.

## No specific inlining

Find out how to specify automatic inlining and use it. Manually specifying what functions to inline makes the code inflexible. The compiler is better at picking what functions to inline than you are, and it has the added advantage that it knows the target processor features such as cache size. In addition, things change. The code gets recompiled on new processors with different requirements. Leave inlining to the compiler. Many compilers simply ignore the inline keyword (it only exists in C99 in any case).

## No register specifiers

Do not use the register specifier. This is a leftover from the days before reasonable optimizing compilers. The compiler does a better job of picking which variables to place in registers than you do, and has the knowledge of the target machine. Leave it to the compiler.

## No initializations

Variables are not to be initialized on declaration, no exceptions. Inside a function, the initializations applied to variables are turned into code in any case, which means that such code should be made explicit and placed into the code section of the function. For globals, not using the initialize feature means that the .data segment of the resulting image is empty, which saves the storage and initialization of the .data segement, a very good property in an imbedded, ROM based system.

Note that const declaration initializations are fine, since they are entirely performed at compile time.

## No reliance on initializations

The code shall not rely on the fact that the variables are initialized to zero. This means both that the code is restartable (can be restarted from the main() function) without clearing out the .bss section, and makes it explicit what values are being relied upon in the code.

## No macros

Macros are strongly depreciated. Use const declarations instead. Macros are a cut and paste operation with no compiler intelligence applied. They create unreadable code and complicate debugging. They are not to be used unless there is no other way.

## No case dependence

The code shall not be written so that there are two symbols of the same name separated only by case:

foo

And

Foo

Are not to be used.

## Lower case only

All symbols are to appear in lower case only. There are no exceptions.

## Use break character to divide segmented names

Use the “\_” character to separate parts of a symbol:

my\_special\_function

NOT:

MySpecialFunction

## No unique namespace dependence

The ability of C to represent separate name spaces by context is not be used:

struct foo;

int foo;

Is legal in C, since they are in two name spaces (“struct foo” is always required to access the structure namespace). This is not to be done.

## Void for void parameter lists

All functions having a void parameter list shall have the void symbol:

void foo(void);

## Only standard C

C99 is the operative standard here. No feature of the local compiler shall be used unless there is no other way to perform the code. In that case, each use of a local feature should bear a comment to that effect.

## No reliance on undefined or implementation specific behavior

Features listed in the standard as undefined or implementation definedare not be used. Exceptions include features clearly meant to be implementation defined, such as the maxium value of int, etc.

## No asm statements

Asm statements are not to be used, no exceptions. Instead put the code in an assembly language module in C callable form, and call it. Placing asm statements in the middle of code invokes serious questions as to if the compiler can guard the registers used from other, C code in the same block (by devining what the asm code does), and calls into question if that is efficient. Asm code living alone in a function simply uses the setup, teardown and parameter access features of the compiler. Learn how to do this yourself, and make the code and explicit assembly language produre. Note that many assemblers also know how to setup, tear down and perform parameter accesses in a C compatible function.

## English comments

Comments shall be in English.

## ISO 8859-1 character set

The ISO 8859-1 character set shall be used in code and comments without exception.

## Use “coined tag” structure declaractions

Where the name of the structure being declared is to be used in the structure itself (self referencing), it is to be coined:

typedef struct \_x {

\_x\* next;

Int d;

} x;

The name of the structure within the structure shall be prepended with an underscore (“\_”). This name shall not appear anywhere outside of the structure definition. The structure name shall be coined from the typedef name, if there is one.

## No use of structure names

If a structure is not anonymous, it is to appear in a typedef and be given a name. The use of the structure name itself is forbidden outside of the structure declaration itself.

## Use explicit initializers

Complex initializers shall express their structure:

typedef struct \_x {

Int a;

Int b;

} x;

const x table[] = {

{ 1, 3 },

{ 4, 5 }

}

Thus each substructure is explicitly marked.

## No use of “=” in enumerations

Enumerations shall be used to create 0 to n values only. No part of the enumeration shall bear an initializer:

enum {

One,

Two = 3,

Three

}

If non-contigous values are needed, use constant declarations or a table.

## Do not ignore function returns

The value of a function shall not be ignored. If necessary, a dummy variable shall be used to hold the unused return value. A function returning a value should never be used as a procedure.

An exception is made for system routines, for example “printf”. However, even a system routine that returns an error should not be ignored. For example, malloc() must always be checked for valid allocation. There is never an exception for this, unless the malloc() in use handles errors internally.

## Use of #ifdef and other macro process statements

#if, #ifdef, #ifndef and other similar preprocessing level statements shall not be used. Use instead normal “if” statements with constant expressions:

const int feature\_enabled = 0;

void foo(void)

{

if (feature\_enabled) {

Functions to be enabled/disabled by option

}

}

Modern compilers can eliminate dead code automatically without use of a macro. In addition, in the future it may be wanted to change constant options to runtime options. Use of compiler constants allows this change without reworking the source code.

Note a common exception to this rule is header file guards, covered elsewhere here.

## Comment coupiously

Functions without comment headers or function comment headers without adequate descriptions are not allowed. Modules without adequate comment headers describing their contents, purpose and all of the exposed globals are not allowed.

## Use modular structure

Modules are not just arbitrary collections of code and definitions. Each module should be a group of related functions and declarations. Ask yourself:

1. What capability is being performed by this module?
2. Do the globals exposed actually need to be exposed?
3. Could this module be better expressed by smaller modules, or combined with other modules?
4. Could you see this module being used in other projects? Does it constitute a good API?

## Use asserts constantly

Asserts are free, in terms of the fact that they can be automatically removed on compiler option. They should be used anytime an invalid condition could occur.

## Do not use asserts for runtime faults

Asserts should never be used for possible runtime faults. For example, using assert to detect if a malloc() succeeded is never a good idea. That would mean that code with asserts turned off would crash at runtime.

## Use dynamic allocation instead of static

Large structures or structures with size that varies according to use should be allocated dynamically. Code should not fail simply because a fixed table size was exceeded. The maximum limit of allocation should, in virtually all cases, be the amount of memory available to the program, not an arbitrary limit choosen by the programmer.

## Code reviews

All code will be peer reviewed by at least one other programmer. Typically, the branch the new code lives on is examined by the reviewer(s), using an appropriate visual difference tool.

## Source control

All source is placed under source control. We will use GIT because of its branching capabilities. Every change is to be a branch, and all changes, even in progress, are to be visible to other programmers. There is no “hidden” or private code. No work in progress is to be left out of source control for either more than one day or more than one page.

## Hanging vs. pinned blocks

We don’t have a particular style preference. The two major styles in use are hanging or pinned block starts:

if (true) {

dothis();

} else {

dothat();

}

for hanging brackets vs:

if (true)

{

dothis();

{

else

}

dothat();

}

However the styles should be used consistently in the same program, so that the reader learns to expect a particular appearance of the code.

Note that the K&R book uses hanging block starts.

## Run-on vs blocked code

There is no preference between these styles:

if (true) dothis();

For run-on vs:

if (true) {

dothis();

}

Obviously run-on lines only work for a single statement.

Note that K&R uses the run on style.

## Use natural definition order

Optimally, functions in the same module should appear so that the functions that use another function appear after it. Many programmers find this style unnatural since it forces the placement of smaller, utility routines first in the file, followed by increasingly larger routines, such as main(). However, the opposite style forces predeclaration of everything. Natural definition is the style in use at Moore/CAD. This ordering becomes second nature rapidly.

## Pointer is part of type

Pointer declarations are considered part of the type. Thus:

int\* p;

void func(char\* s);

That is, the “\*” character is placed with the type declaration, not with the name. When a series of declarations are made, do not mix pointer and non pointer definitions:

int\* a;

int b;

Not:

int \*a, b;

## Use bool type and constants

C99 adds a type macro “bool” and the constants “true” and “false”. They should be used instead of int, 0 and 1.

Use NULL

The standard constant NULL should always be used instead of 0 when standing for a pointer that points to nothing.

## Explicit comparisions are preferred

Using explicit comparisions instead of comparisions with zero are preferred:

while (p) p = p->next;

Would be preferred as:

while (p != NULL) p = p->next;

However, values that are Boolean are an exception. Just as you would not write:

If ((a != b) != 0) dothis();

Neither would you write:

BOOL findtrue(void);

if (findtrue() != 0) dothis();

## Use array notation when possible

Use standard array notation when it is possible to do so:

void func(char\* p)

{

while (\*p) p++; // find end of string

}

Instead this is preferred:

void func(char p[])

{

int I;

i = 0;

while (p[i] == 0) i++; // find end of string

}

Although the second example is more verbose, it makes it clear we are traversing an array, and what the starting point of the array is, since p remains logically at the start of the array. The code generated is equivalent. The compiler knows how to convert the second form into pointer only accesses. But the second form is more readable.