# The CoAutomation Simple C Style Guide

Craig E. Goldman

(2013-11-06) (updated 2018-12-08)

The purpose of a style guide is to enable an individual programmer or a programming team to write clear, consistent and reproducible code. This is a "simple" style guide; it contains the minimum to accomplish this goal.



Even the simplest C Style Guide should have these six elements

- Naming Conventions
- Width of Text
- Use of Indentation and Braces
- Text Alignment of Comments
- File Order
- Debug Mark

## Naming Conventions

All 'named' elements of the C code should have a unique style so that the element is easily recognized.

### a. Function/Procedure Names

The function/procedure name must begin with a capital letter; succeeding characters may be a capital or small letter or a numerical digit (0-9) or the underscore character. The character following an underscore should be a capital letter. An underscore character can not follow an underscore character. The total number of characters must be 31 or less.

Example: `Filter_Init`
Example: `Value8_CountUp`

### b. Variable Names

#### i. Automatic (stack) Variable Names

The automatic variable name must begin with a small letter; succeeding characters may be a capital or small letter or a numerical digit (`0-9`). The underscore character is not allowed in the variable name. The total number of characters must be 29 or less.

Example: `analogValue`
Example: `countIndx`

#### ii. Static Variable Names

The static variable name must begin with the two characters '`s_`' followed by a small letter; succeeding characters may be capital or small letter or a numerical digit (`0-9`). Other than the 2nd character, the underscore character is not allowed in the variable name. The total number of characters must be 31 or less.

Example: `s_filterSize`

#### iii. Global Variable Names

*This is not a recommendation for global variables. Global variables should be avoided. Sometimes a global variable is created by the C language tool chain; if the name can be changed by the programmer, apply this rule.*

The global variable name must begin with the two characters '`g_`' followed by a small letter; succeeding characters may be a capital or small letter or a numerical digit (`0-9`). Other than the 2nd character, the underscore character is not allowed in the variable name. The total number of characters must be 31 or less.

Example: `g_faultFlag`

### c. Type Names (programmer-defined types)

Type names declared by a `typedef` statement must begin with a small letter; succeeding characters are limited to small letters, except the type name must end with the characters '`_t`'. The total number of characters in a type name must be 17 characters or less.

Example:
```
typedef signed long int sint32_t;
```

### d. Defined Constants

Defined constant names (created by a #define statement) must begin with a capital letter; succeeding characters are limited to a capital letter, a numerical digit (`0-9`) and the underscore character. The total number of characters must be 31 or less.

Example:
```
#define ARRAY_SIZE  5
```

### e. Constants of an Enumerated Type

The constants declared for an enumerated type must begin with the small letter 'e' followed by a capital letter; succeeding characters are limited to a capital letter, a numerical digit (`0-9`) and the underscore character. The total number of characters must be 31 or less.

Example:
```
typedef enum {ePASS=0, eFAIL} teststatus_t;
```
Example:
```
typedef enum {eCOUNT_BY2, eCOUNT_BY4} updateby_t;
```

## Width of Text (line length)

Few things are more annoying than trying to review code and discovering that the right-most portion of the code is missing because it does not "fit". "Wrapped" text also is very hard to read. Programmers should remember that some programming tools place additional information to the left of the code during output. This should be accounted for when setting a line length. Despite advances in displays and printers, I have found that the old line length limit of 80 characters is about right. Since I hate re-formatting an entire line just to make a variable a bit longer, I use a "soft" limit of 80 characters with a hard limit of 90 characters.

All lines of code text should be 90 characters or less. The last 'word' of the code line must begin before the 80th character in the line.

## Use of Indentation and Braces

I have read that a basic indent of 3 or 4 spaces provides the best compromise between making it easier to decipher the code elements and making the indentation too large. Some sources claim that "studies" have been done. I have not seen the studies, but I prefer 3 spaces. Indentation must be done using spaces; do not use a TAB. Tab settings can change (and often do change) screwing up the format of the code.

When using braces, the closing brace is always horizontally aligned (directly under) the opening brace. No exceptions.

a. **Function/Procedure declarations**
   Function/Procedure declarations always start at the left-edge of the page.

b. **The body of a Function/Procedure**
   The opening and closing braces are located at the left-edge of the page. The code that is the body of a function/procedure is indented 3 spaces from the declaration of the function/procedure.

c. **The body of a '`for`', '`while`' or '`do`'**
   The code that is the body of a '`for`', '`while`' or '`do`' statement is indented 3 spaces from the '`for`', '`while`' or '`do`' statement. The opening brace is directly below the '`f`' (of the '`for`'), the '`w`' (of the '`while`') or the '`d`' (of the '`do`'). The closing brace is horizontally aligned with the opening brace.

d. **The body of an '`if`' or '`else`'**
   The code that is the body of an '`if`' or '`else`' statement is indented 3 spaces from the '`if`' or '`else`' statement.

e. **The '`switch`' statement**
   The opening and closing braces of the '`switch`' statement are horizontally aligned with the '`s`' of the '`switch`'.

f. **The '`case`' and '`default`' statements of a '`switch`'**
   The '`case`' or '`default`' statements within a '`switch`' are indented 3 spaces from the '`switch`' statement.

g. **The body of '`case`' and '`default`' statements in a '`switch`'**
   The code that is the body of a '`case`' or '`default`' statement within a '`switch`' is indented 6 spaces from the '`case`' or '`default`' statement.

(Lower portion of page left intentionally blank)

Here is a quick, nonsense code example that shows indentation.

```
void SillyFunction( void)
{
   // Function body is indented 3 spaces
   Int16_t i;
   Int16_t j;
   for( i=16; i>0; i-=1)
   {
      // body of a for statement is indented 3 spaces
      j += 5;
   }
   while( j > 0)
   {
      // body of a ‘while’ is indented 3 space
      j -=4;
   }
   switch( j)
   {
      // ‘case’ is indented 3 spaces
      case 0:
            // body of a ‘case’ is indented 6 spaces
            j = i;
            break;
      case 1:
            j = i - 1;
            break;
      default:
            j = j - 1;
            break;
   }
   if( i == j)
   {
      // body of an if is indented 3 spaces
      i += 10;
   }
   else
   {
      // body of an else is indented 3 spaces
      j -= 1;
   }
   return;
}
```

## Text Alignment of Comments

It is desirable that all of the comments in a code project have a similar text arrangement. This makes the code not only more pleasant to read, it prevents the comments from becoming a distraction.

- Comments outside of a function or procedure are always aligned to the left-edge of the page.
- Comments within a procedure or function are always aligned to the code.
- Comments within a code line should be avoided. (Yes, really avoided.)
- No comments are placed to the right of a code line, except when there is a list of declared variables (as in a structure), a list of enumerated constants or a list of defined constants.

Comments within a procedure distract from the reading of the code. These types of comments should be minimized. In particular, comments placed to the right of the code for the most part cause more harm than benefit.

## File Order

Many programmers simply "toss" definitions, type declarations, function, prototypes etc. into a file. This not only makes searching for a particular declaration difficult, it can actually cause code problems. This is fixed simply by having an order to the items within a file.

Every code file should have the following sections in this order:

1) A module definition statement
2) A module Title Block
3) All include files
4) File Documentation Block
5) Definitions and Macros (#define)
6) Type Definitions (typedef)
7) External and global variables (if any)
8) File-scope static variables
9) Function prototypes of static functions and procedures
10) Public/Linkable Initialization and/or Reset procedure
11) Other Public/Linkable procedures in alphabetical order
12) Interrupt handlers (if any)
13) Local/Static procedures in alphabetical order
14) A "File End" marker

## Debugging Mark

A Debugging Mark is a special sequence of characters that are inserted into the code to call attention to that area. Although technically not part of code style, the consistent use of a Debugging Mark is too valuable a "device" to leave out of even a simple style guide.

The Debugging Mark is used for marking places in the code that need to be reviewed or changed at a later time, but before any release. If you change a constant during debug, add the Debugging Mark and a few words as a reminder so that this change can be undone before the release. While working on one problem, you notice something "funny", add the Debugging Mark plus a few words and focus on the original problem. If a procedure is only "partially-written" for testing, add the Debugging Mark plus a

note to finish the procedure. At a later time, you can search for debugging marks for code lines that need work.

To be useful, the Debugging Mark must be a sequence of characters that would never occur in code. Personally, I use **????** (four question marks). It is easy to add and easy to find using a text search. A project should only have one Debugging Mark even if there is more than one programmer. If code is constructed by several developers, the Debugging Mark should be followed by the initials of the developer who added the mark to the code. This is much better than each person creating their own Debugging Mark, which would make it much more difficult to scan a document during reviews.