Coding Standards for Embedded Software Development Using C

5/20/2021

X

John Coloccia
Engineering Review - Director Software Dev...
Signed by: John Coloccia

5/20/2021

X

Roni-Sue Myles
Quality Review - Senior QA
Signed by: Roni-Sue Myles

60 Prestige Park Road ~ East Hartford, CT 06108 ~ Toll-Free Phone: 800.952.2535 ~ Fax: 860.812.2465 ~
www.GDSoftwareServices.com

Revision History

Revision	Description	Author	Date
-	Original.	J. Coloccia	31-Jul-2017
A	Changed document Level.	R. Blanco	29-Mar-2019
B	1. Changed Document Number, Level, and Rev. 2. Fixed font size in Section 1.6.3	R. Myles	22-Nov-2019
C	1. Updated footer to reflect GDC's new address and fax number. 2. Updated header and Signature Page to conform to document standards. 3. Cleaned-up formatting throughout document to conform to document standards. 4. Fixed number sequence in section 1. 5. Added table header in section 2.1. 6. Updated title from 999-1504-018 to 999-1503-031 in the Document Properties. 7. Removed blank heading number's 1 - 4.1.13 in heading 3.3.1, bullet 1. 8. Add bullets in section 3.3.6. 9. Updated Document Name in Header to include C in the name. 10. Removed 1.3 in section 1.1, paragraph 2, end of last sentence.	R. Myles	19-May-2021

Table of Contents

REVISION HISTORY **2**

1. INTRODUCTION **5**

1.1. PURPOSE 5
1.2. SCOPE 5
1.3. PROCESS OVERVIEW 6
1.4. EXCEPTIONS 7
1.5. DOCUMENT CONVENTIONS 7
1.5.1. Wording Conventions *7*
1.5.2. Symbolic Conventions *7*
1.5.3. Coding Standard Identification *7*

2. REFERENCES **8**

2.1. REFERENCE DOCUMENTS 8
2.2. ABBREVIATIONS AND DEFINITIONS 8
2.3. CONTROL AND MAINTENANCE 8

3. SOFTWARE GUIDELINES **8**

3.1. GENERAL GUIDELINES 8
3.1.1. External Standards [CS_Rule_001] *8*
3.1.2. Consistency [CS_Rule_002] *9*
3.1.3. Compiler Specific Behavior and Compiler Options [CS_Rule_003] *9*
3.1.4. File Size [CS_Rule_004] *9*
3.1.5. One Statement Per Line [CS_Rule_005] *9*
3.1.6. Files, Function, Variable Naming [CS_Rule_006] *9*
3.1.7. Undefined Behavior [CS_Rule_007] *10*
3.2. STYLE GUIDE 10
3.2.1. File/Directory names [CS_Rule_008] *10*
3.2.2. Directory Structure [CS_Rule_009] *10*
3.2.3. Special File Names [CS_Rule_010] *11*
3.2.4. Line Length [CS_Rule_011] *11*
3.2.5. Braces *11*
3.2.6. Indentation *11*
3.2.6.1. Indents 11
3.2.6.1. Tabs 11
3.2.6.3. switch Statements 12
3.2.6.4. Multi-line condition expressions 12
3.2.7. do…while *12*
3.2.8. return statements *12*
3.2.9. Spacing *13*
3.2.10. General C Naming Convention *13*
3.2.11. Public Symbol Naming *14*
3.2.12. Private Symbol Naming *14*
3.2.13. Comments *14*
3.2.14. Comment style *15*
3.3. IMPLEMENTATION GUIDE 17
3.3.1. Header File Inclusion *17*
3.3.2. Control Structure Nesting Levels *17*

3.3.3. Function Entry/Exit points 17
3.3.4. typedef usage 18
3.3.5. Avoid typedefs for pointers 18
3.3.6. Standard Types 18
3.3.7. Macros 19
3.3.8. Macro Expressions 19
3.3.9. Multiple Statement Macros 19
3.3.10. switch Statements 19
3.3.11. Operator Precedence 20
3.3.12. Unconditional Jumps 20
3.3.13. Prefer constants to #define 21
3.3.14. Variable Declarations 21
3.3.15. Global Variables and externs 22
3.3.16. Variable Description 22
3.3.17. Variable Initialization 22
3.3.18. Function Return Values and Parameters 22
3.3.19. Ternary Operator 23
3.3.20. Pointer Arithmetic 23
3.3.21. Unbounded String Functions 23
3.3.22. Recursion 23
3.3.23. Assignment Statements 24
3.3.24. Parameter Names in Function Prototypes 24
3.3.25. Parameter Limitation 24
3.3.26. Variable Length Parameter Lists 24
3.3.27. Code Simplicity 24
3.3.28. Cyclomatic Complexity 24

1. Introduction

1.1. Purpose

This Work Instruction provides instruction and guidance to aid in the development of software source code for embedded systems generated and/or maintained at General Digital Software Services (GDSS). These guidelines are intended to increase the consistency of embedded software developed within GDSS. Development of consistent software will simplify the reading, reviewing, and maintaining of code.

This Work Instruction is intended for software developers generating/maintaining source code developed in C language and associated with embedded systems. The reader should have a general knowledge of C.

This Work Instruction shall be tailored as necessary on a project-by-project basis with the permission of the Software Project Lead.

1.2. Scope

This Work Instruction should be used on all embedded software development projects requiring the development of software in C language. This Work Instruction should be used to aid in the development of software source code to assure consistency of all software source code developed on the project.

However, it is recognized that software development at GDSS may span a large range of effort, from the small single-person development effort to the large multi-person/multi-platform software development effort including both new code development and code enhancements. Therefore, each software development task needs to be assessed for its requirements, and an appropriate set of activities and guidelines from this document should be specified. In support of this, each software development project will specify what activities and guidelines will be required in a Software Development Plan (SDP) or similar document that suits the nature of the specific project. The SDP may reduce the guidelines specified in this Work Instruction for smaller projects. In general, where customer or project coding guidelines already exist, those guidelines will be used. The SDP should identify any instances where the GDSS coding standard will be used to supplement the customer or project coding guidelines, if present. If the GDSS coding standard is used, either as the standard or as a supplement to another standard, the SDP should also identify any exceptions to and/or deviations from this GDSS Work Instruction.

1.3. Process Overview

The guidelines described within this Work Instruction support and expand GDSS's Software Life Cycle Development Process with regard to the Software Coding activities associated with developing software for embedded systems as shown in Figure 1.

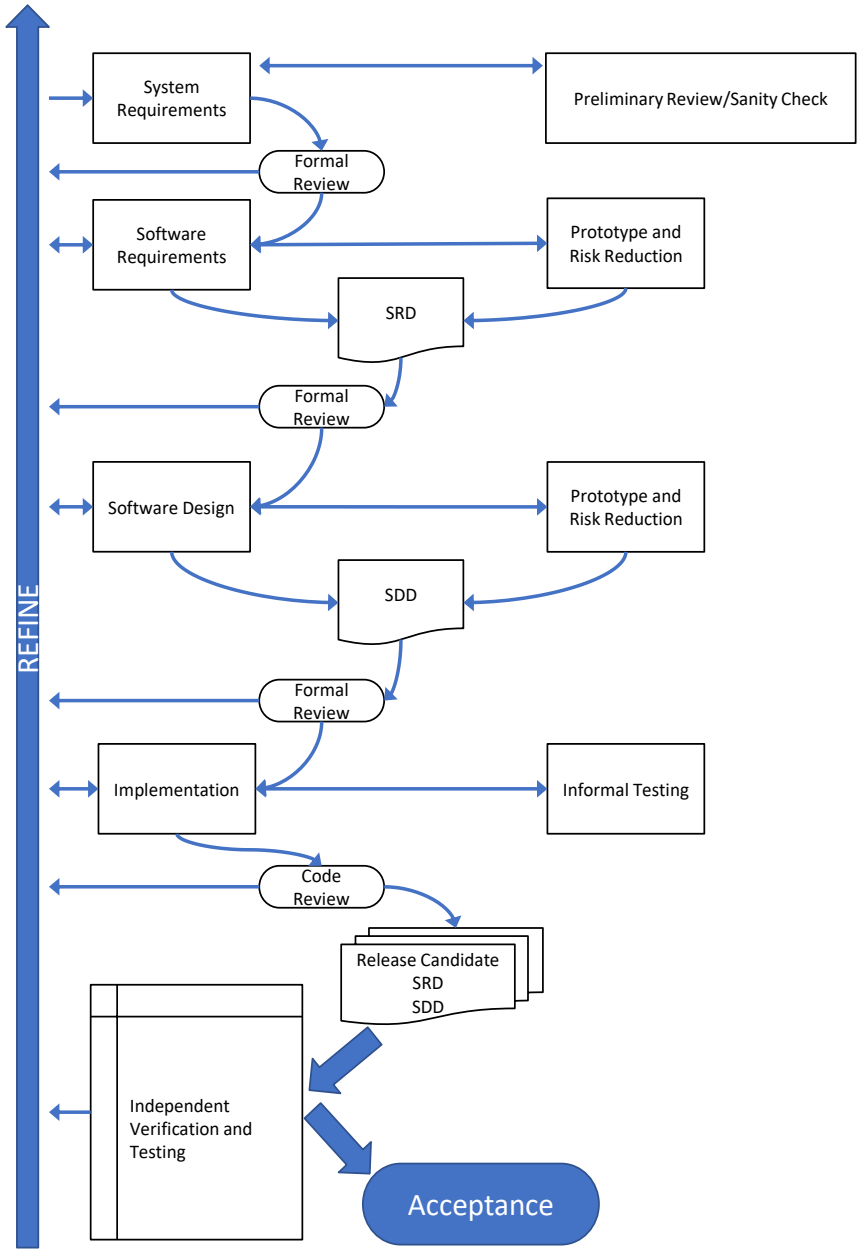

Figure 1 - GDSS Software Life Cycle Diagram

1.4. Exceptions

Embedded software source code generated prior to the establishment of this Work Instruction does not have to conform to the guidelines specified within this document. However, any time existing code is modified, the changes should conform to the style in effect for that file.

1.5. Document Conventions

1.5.1. Wording Conventions

The use of ***"SHALL"*** and ***"SHOULD"*** within the guidelines specified within this Work Instruction observes the following rules:

- The word ***shall*** in the text indicates a firm commitment to achieve the stated guideline and to validate compliance. Departure from such a guideline is not permissible.
- The word ***should*** in the text expresses a recommendation or advice guideline. Such recommendations or advice are expected to be followed unless good reasons are stated within the project plan, or similar project documentation, for not doing so.

1.5.2. Symbolic Conventions

This document uses the following symbolic conventions:

- Words or symbols in bold represent the use of an C language keyword within standard text.

Example:
#DEFINE, **return**

- Words or symbols in text boxes represent sample source code.

Example:

```
void foo()
{
    something();
}
```

1.5.3. Coding Standard Identification

This document will contain a Coding Standard Identification (CSI) tag for traceability and will be located next to the heading of the corresponding guideline.

Example:
3.4.5. One Statement Per Line [CS_Rule_003]

This CSI will provide traceability to information on how each guideline will be validated (e.g., code read inspection, static analysis tool), and will reside in the spreadsheet C CodingStandards Compliance Matrix.xlsx.

2. References

The following documents, of the exact issue shown, provide supporting information to the processes associated with GDSS's Software Life Cycle.

2.1. Reference Documents

Table 1: Document Reference

Document #	Title
999-1501-000	General Digital Software Services Quality Manual
999-1505-000	Software Abbreviations and Definitions
N/A	MISRA C_2012 Guidelines for the use of the C language in critical systems
N/A	CodingStandards Compliance Matrix.xlsx

2.2. Abbreviations and Definitions

Definitions for the acronyms and terminology used within this Work Instruction can be found in Document *Software Abbreviations and Definitions.*

2.3. Control and Maintenance

The control and maintenance of this Work Instruction will follow the process defined in Document *General Digital Software Services Quality Manual.*

3. Software Guidelines

GDSS embedded coding style is very simple, clean, and straightforward. While specific algorithms may be complex and nuanced, the source code itself should be written in a style that is easily understood by anyone with a very basic understanding of the programming language. The purpose of these guidelines is to formalize a coding style which achieves the following goals:

- Generally, conform to industry norms
- Logical separation of modules for comprehension and namespace cleanliness
- Clarity: obfuscation is bad, especially in safety critical and/or embedded applications
- IV&V code reads, and testing should be simple and straightforward
- Simplicity: if a standard is difficult to follow, it will be difficult to enforce

3.1. General Guidelines

This section defines general guidelines applicable to the development of software source code in C language.

3.1.1. External Standards [CS_Rule_001]

If the software is to be certified to a specific industry standard, the specific industry standard shall take precedence over GDSS standards if there is a conflict.

3.1.2. Consistency [CS_Rule_002]

Code within a project should be consistent. It is preferred that GDSS coding standards are followed as closely as possible. Where legacy code exists that does not follow GDSS standards, code modifications should conform to the existing style and/or appropriate sections of the legacy code should be updated. The decision to conform to vs update legacy coding standards shall be made on a case-by-case basis, considering factors such as technical risk, schedule risk, and the overall practical benefits of updating.

3.1.3. Compiler Specific Behavior and Compiler Options [CS_Rule_003]

Code should conform to a published C standard. Software should avoid compiler specific behavior where possible. Code should not depend on non-trivial compiler options for proper behavior.

Some trivial compiler options:

- library/header paths
- optimization options

Some non-trivial options:

- defining a macro that controls conditional compilation

Example:
cc myProgram.c -d USE_RELEASE_ALGORITHM=1

In general, if something can be controlled within source code solely by using features of the published language spec, it should be controlled with those features.

3.1.4. File Size [CS_Rule_004]

A single file should not exceed 1,000 lines, all-inclusive (i.e., including comments, code, & blank lines).

3.1.5. One Statement Per Line [CS_Rule_005]

Code should be limited to one statement per line.

3.1.6. Files, Function, Variable Naming [CS_Rule_006]

Files, function declarations and variable definitions should use common sense when being named. Use meaningful names that indicate the intent of the class/module/function/variable. If the intent changes during software development so that the original name is misleading, the class/module/function/variable should be renamed.

3.1.7. Undefined Behavior [CS_Rule_007]

Code shall not depend on undefined behavior for proper functionality.

Example:

```
x = ++x - z; //NOT OK
```

3.2. Style Guide

3.2.1. File/Directory names [CS_Rule_008]

All file names shall be lower case. Names shall only contain alphanumeric characters and the underscore '_' character. The underscore character shall be used as a separator. Names shall not begin or end with an underscore. There shall not be multiple underscore characters adjacent to each other. Names shall not contain spaces.

Example:

```
foo.c
file_name.c
longer_file_name.c
_foo.c                  (unacceptable, begins with an underscore)
file__name.c            (unacceptable, multiple underscores adjacent to each other)
file_#2.c               (unacceptable, # is not alphanumeric)
```

Exceptions shall be made for names which cannot follow this naming convention, either for technical or project reasons. Exceptions shall also be made for files which traditionally have a different specific naming convention and conforming to GDSS conventions would result in confusion or increase potential for errors.

Example:
Makefile (acceptable, make files are traditionally capitalized)

3.2.2. Directory Structure [CS_Rule_009]

For the purposes of this section, a module is defined as a set of functions, definitions and declarations that logically belong together.

- project_root/
  Shall contain files such as readme.txt, makefiles, and build instructions. readme.txt should include, at a minimum, the module name that serves as the entry point for the program. project_root should not contain any source code.
- project_root/include
  Shall contain any include files which must be shared between modules.
- project_root/module_name
  Shall contain all source code for a given module, except for public header files stored in project_root/include.

3.2.3. Special File Names [CS_Rule_010]

- project_root/module_name/module_name.c
  Should contain all public global data definitions for a module.
- project_root/include/module_name.h
  Should contain all public declarations for a module. For example, externs, public function prototypes, public structures and macros shall be declared in this file.
- project_root/module_name/module_name_p.c
  Should contain private global data definitions for a module.
- project_root/module_name/module_name_p.h
  Should contain private data declarations for a module.

3.2.4. Line Length [CS_Rule_011]

Code should not extend past 80 columns except where longer lines significantly increases readability.

3.2.5. Braces

Brace positions should be in Stroustrup style, which is illustrated below. And, as the example shows, even single line bodies of **if**, **else**, **do**, **while** should be enclosed by braces.

Example:

```
if (x > 7) {
    x = y - 3;
}
else {
    x = z - 3;
} //OK
if (x > 7)
    DoSomething(); //NOT OK, missing braces around single statement
```

3.2.6. Indentation

3.2.6.1. Indents

Indents shall be 4 or 8 spaces and shall be consistent within a project. Indentation level shall be specified as part of the software development plan.

3.2.6.1. Tabs

Tabs shall not be used. Exceptions shall be made where tabs must be used for technical reasons (example: make files).

3.2.6.3. switch Statements

switch statement **case** labels shall not be indented. Code within a **switch** statement shall be indented.

Example:

```
switch (x) {
case UP:
    do_something();
    break;

case DOWN:
    do_something_different();
    break;

default:
    //empty default case
    break;
}
```

3.2.6.4. Multi-line condition expressions

When conditions span multiple lines, the additional lines should be indented one additional level, and the relational operator should begin the additional lines.

Example:

```
if (something_is_true()
        && something_else_is_false()) {
    i = 0;
}
```

3.2.7. do…while

do…**while** loops shall be formatted as follows:

Example:

```
do {
    …
} while (expr);
```

3.2.8. return statements

return statements should not use parentheses when returning simple values or variables. **return** statements should use parentheses when returning an expression.

Example:

```
return 1;

return ret_val;

return(x != 0);
```

3.2.9. Spacing

There shall be spaces between/around:

- relational/mathematical/assignment/binary operators
- after commas
- keywords
- braces
- conditionals

There shall not be spaces around:

- unary operators
- structure access operators
- parentheses
- semicolons

Example:

```
x + y = z

if (this && that) {
}

do_something(parm1, parm2);

int a[10] = { 10, 20, 30, 40 };

a = error ? NULL : get_name();

*a = temp;

a.foo = temp;

a->foo = temp;

a++;
```

3.2.10. General C Naming Convention

- Shall only use alphanumeric characters and underscores. Dashes or other symbols shall not be used.
- Names shall not begin or end with an underscore.
- Multiple underscores shall not be adjacent to each other.
- Macros and enumerations shall be all upper case, with underscores separating words.
- All other names shall be all lower case, with underscores separating words.

Example:

```
//CAPITALIZED NAMES

//macros are all caps with underscores
#define THIS_IS_A_MACRO

//enums are all caps with underscores
enum {
    ONE,
    TWO,
    BUCKLE_MY_SHOE
};


//lowercase names

int tmp;

int serial_id;

struct sample_struct {
    int i;
};

void sample_func(int i, float adc_reading);
```

3.2.11. Public Symbol Naming

Public symbols are defined as any symbols intended for use and exposed outside of a module. All public symbols shall start with the name of the module that declared the symbol.

Example, using a module named *eprom* for demonstration purposes:

```
struct eprom_mem_block {
    void *start;
    unsigned size;
};

extern int eprom_is_erased;

void eprom_erase(struct eprom_mem_block *block);
```

3.2.12. Private Symbol Naming

No special naming convention is necessary in addition to the General C Naming Convention.

3.2.13. Comments

Comments should add useful information that cannot be obtained from the actual code. This may include details of why an implementation is used, warnings about potential unexpected behavior or possibly what algorithms are being implemented. Avoid trivial or overly detailed comments.

Example:

```
//NOT OK
//loop over all of the array elements
for (loopCounter=0; loopCounter<numEle; loopCounter++)
{
    //set a's element equal to b's element
    a[loopCounter] = b[loopCounter];
}

//OK
//Copy b for safe keeping. Algorithm will operate on tmp.
for (i = 0; i < numEle; i++)
{
    tmp[i] = b[i];
}
```

3.2.14. Comment style

- Block style comments should not be "intermingled" with code. If a section of code requires a block comment, place the block before the code in question.
- Block style start/end sequences shall appear alone on a line.
- Every line of a block comment shall begin with a character sequence which makes it obvious the line is a comment.
- There shall be at least one space between characters denoting a C style (/*…*/) comment and the comment itself.
- If a comment appears on the same line as code, it shall appear at the end of the line, shall not extend beyond one line, and shall not have dependencies on surrounding comments. In other words, if the entire line of code in question cannot be moved, in its entirety, without changing the semantics and clarity of the comment, the comment shall not appear on the same line as code.
- Code shall never be commented out. Once commented out code is allowed to infest a repository, it becomes impossible to know if code is intentionally commented out vs commented out code which was used for debugging but was mistakenly left commented out. Therefore, commented out code in the repository is always a mistake.

The following examples demonstrate the four, acceptable comment styles.

Example:

```
Style 1:

/*****************************************************************************
* Use this style for file and function headers only
*
*****************************************************************************/

Style 2:

/*
 * Use this style for block comments WITHIN code.
 * Only use this between major blocks of code to convey any useful
 * information about what's going to happen next.
 * Don't use this style within a small loop
 * or something of that nature. Put this BEFORE the loop, if necessary.
 */

        /*
         *
         * This style respects indent level of surrounding code
         *
         */

Style 3:

// preferred form of single line comments

Style 4:

/* acceptable if C++ style comments are not permitted. */


        // single line comments always
        /* respect the indent level of surrounding code */


EXAMPLES OF BAD COMMENTS:

/*
    Unnaceptable block comment style
    It's ugly and also difficult to tell where it begins and ends*/

/*And definitely don't ever do something like this

   for (i=0; i<j; i++) wait();


   because it's very difficult to spot what's a comment and what's not*/

    //comments on same line as code, but not isolated
    for (i = 0; i < SIZE; i++) { //loop over the entire array
        tmp[i] = name[i];        //store the current value of name in a
        eprom_read(name,i);      //tmp array, and then read in a new value
    }
```

3.3. Implementation Guide

3.3.1. Header File Inclusion

- Source code files should include only the required header files.
- Header file names may include relative path information but include directories should be set so that path information is unnecessary.
- Header file names shall never include absolute path information.

3.3.2. Control Structure Nesting Levels

Control structures should be limited to 3 levels deep.

3.3.3. Function Entry/Exit points

- Functions shall not contain more than 1 entry point.
- Functions should not contain more than 1 exit point unless multiple exit points significantly increase readability and/or decreases complexity.

Example:

```
BOOLEAN foo(void) //NOT PREFERRED
{
    if (value > 1)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}


BOOLEAN foo(void) //OK, but can be better
{
    BOOLEAN ret_val = FALSE;

    if (value > 1)
    {
        ret_val = TRUE;
    }

    return ret_val;
}

BOOLEAN foo(void) //PREFERRED
{
    return (value > 1);
}
```

3.3.4. typedef usage

typedefs shall not be used to alias structs, unions and enums.

3.3.5. Avoid typedefs for pointers

Using a **typedef** to create pointer types shall be avoided except for opaque types. If opaque types are used, they shall have the semantics of a handle.

Example:

```
typedef some_struct* some_struct_ptr; //NOT OK, typedef obfuscates pointer
                                      //type

//the following is OK. eprom_handle is a completely opaque type.
//in public header file
typedef void* eprom_handle;

//in private implementation file
struct private_data {
    int i;
};

void some_function (eprom_handle handle)
{
    struct private_data *data = handle;
}
```

3.3.6. Standard Types

- It is acceptable to use **int** and **unsigned** types in limited cases when a generic number is needed of no specific size.
- **char** types should be used, and only used, for actual character/string data.
- All other types should be **typedef**'ed, such as **uint32**.
- If a specific size is required, the **typedef** should be used.

Example:

```
int i;
uint32 uint32_p1;
unsigned unsigned_p2;
char c1;
…
for (i = 0; i < size; i++) //OK, simple, generic int
…
out_port_a = unsigned_p2 & 0xFFFFFFFD; //NOT OK, assumes 32 bit int
out_port_a = uint32_p1 & 0xFFFFFFFD; //Use this instead
…
c1 = 'J'; //OK
…
//calculate something
i = i + c1; //NOT OK, don't use chars to do generic math unless doing
            //character conversions
```

3.3.7. Macros

The use of function-like macros should be avoided wherever possible. Prefer to use a real function instead. If function-like macros are used, they shall be extremely small and simple.

Example:

```
//OK. Function can not be used, and functionality is small and simple
#define SWAP(type, a, b) {\
    type swap_tmp = a;\
    a = b;\
    b = swap_tmp;\
}
```

3.3.8. Macro Expressions

Macro expressions intended to return a value shall be enclosed in parentheses.

Example:

```
#define CALC_PRODUCT(a, b) ((a) * (b)) //OK

#define CALC_PRODUCT(a, b) a * b //NOT OK
```

3.3.9. Multiple Statement Macros

Multiple statement macros shall be surrounded with curly braces.

3.3.10. switch Statements

Every **switch** statement shall include a **default** case. An empty **default** case shall include a comment indicating intention.

Each **case** shall terminate with a **break** or shall include a fall through comment.

3.3.11. Operator Precedence

Statements that include operators with different orders of precedence, or that are simply not obvious and clear, shall use parentheses to explicitly define design intent. Exceptions are made for standard and obvious cases where additional parentheses will only decrease readability.

Example:

```
i = x + y + z; //OK
i = x + (y * z); //OK
if (this || that & mask | bits)      //NOT OK
if (this || ((that & mask) | bits)) //ACCEPTABLE

tmpThat = that & mask;
tmpThat |= bits;
if (this || tmpThat) //PREFERRED for more complicated expressions

*s.member  = 5; //NOT OK – not obvious what the dereference applies to
*(s.member) = 5; //OK

(*s).member = 5; //ACCEPTABLE
s->member = 5; //PREFERRED

s->array[10] = 5;    //PREFERRED…common pattern that behaves as expected
(s->array)[10] = 5; //OK, but parentheses just add clutter
```

3.3.12. Unconditional Jumps

The use of unconditional jumps should be avoided. Use of unconditional jumps shall be limited to cases where it significantly improves code readability, reliability and/or simplicity. A common case where unconditional jumps are acceptable/preferred is to simplify deep nested error handling, or basic sanity checking of parameters.

Example:

```
//This is prefereable when otherwise handling badError makes the code significantly
more complex
//and error prone

if (something) {
    …
    …
    if (somethingElse) {
        …
        …
        for (someCondition) {
            …
            …
            //UT OH!
            goto badError;
        }
    }
}

badError: GracefulShutdown();
```

3.3.13. Prefer constants to #define

Constants should be used in preference to **#define.**

Example:

```
#define MAX_LENGTH ((int)10) //this is acceptable
const int max_length = 10; //this is preferred
```

3.3.14. Variable Declarations

There should be only one variable declaration per line except for trivial declarations with no initialization.

Example:

```
char rx1;
char rx2;  //OK
int i, j, k; //OK i,j,k will be used as loop counters

char rx1, rx2; //NOT OK
int i, j, k=5;   //NOT OK. Initializations must occur on a separate line
```

Local variables should generally be declared at the beginning of a function. Exceptions are made where declaration within the code body significantly improves readability and semantics.

Variable names should be descriptive where appropriate but favor short and concise names where longer names do not improve readability.

Example:

```
int loopCounter; //instead of this
int i;           //use this

int tempAccumulator; //instead of this
int tmp;             //use this
```

3.3.15. Global Variables and externs

- Global variables shall only be defined in .c or assembly language files.
- externs shall be defined in a .h files.
- externs shall never appear in a .c file.
- The .c file where a global variable is defined must include the .h file where it is declared. This will force the compiler to check the declaration and definition for consistency.

3.3.16. Variable Description

Non-trivial variable declarations should include a brief one-line description of the variable. Trivial declarations should not include a description.

Example:

```
int bpMon; //blood pressure level – OK
int i;     //loop counter – NOT OK, don't clutter up code with trivial comments
```

3.3.17. Variable Initialization

The value of all auto variables shall be set before being read. If a variable may be set to a meaningful value at initialization time, it should be initialized. If a meaningful value is not available (loop counters, for example), it should not be initialized. Initialization to a meaningless value obfuscates the intent and semantics of the variable.

Example:

```
int foo (void)
{
    int ret_val = SUCCESS; //OK. asssume ret_val is SUCCESS unless an error
                           //occurs
    int i = 0; //NOT OK if i is a loop counter. Init obfuscates the real
               //purpose.
    …
}
```

3.3.18. Function Return Values and Parameters

Function return values and parameter lists shall always be explicitly declared.

Example:

```
int foo(void);  // OK

foo(); //NOT OK
```

3.3.19. Ternary Operator

Ternary Operators should be a neat and clean association of a value assignment with a conditional and should avoid implementing or calling functions with side effects.

Example:

```
clipped = test_number > MAX_LIMIT ? TRUE : FALSE;
```

3.3.20. Pointer Arithmetic

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Example:

```
int a[10];
int *p = a;
…
*(p+1) = 0; //OK assuming p+1 is still in array a

struct s
{
    int i;
    int k;
    int a[10];
};

struct s *p = &some_struct;
…
*((int*)p+1) = 0; //NOT OK. Must not treat i and k as a contiguous array

*((p->a)+1) = 0; //ugly and should be avoided, but OK.
                 //Pointer arithmetic results in pointer within the same
                 //array
```

3.3.21. Unbounded String Functions

The use of unbounded standard C/C++ string handling functions should be avoided. A bounded string handling function should be used instead (e.g., use **strncpy()** instead of **strcpy()**).

3.3.22. Recursion

- Unbounded recursion shall never be allowed.
- Bounded recursion is allowable, but should still be avoided, favoring a simpler, verifiable, implementation.

3.3.23. Assignment Statements

An assignment statement should never be placed inside another statement. An acceptable exception is loop variable management.

Example:

```
if ((x = y) != 7) //NOT OK

for(x = BOTTOM; x < TOP; x++) //OK
```

3.3.24. Parameter Names in Function Prototypes

Function prototypes shall include the parameter name as well as the type.

3.3.25. Parameter Limitation

The maximum number of parameters passed to a function should be limited to five.

3.3.26. Variable Length Parameter Lists

Variable length parameter lists should be avoided.

3.3.27. Code Simplicity

- Numeric and logical expressions in code should be kept as simple as possible to reduce the complexity of the expression.
- In such cases where code is too complex, readability should be improved by simplifying the code rather than by using comments alone to explain the complexity.

3.3.28. Cyclomatic Complexity

Any given code file should have a maximum of 20 possible linearly independent paths of execution (a cyclomatic complexity of 20). Exceptions shall be documented if a code file has a cyclomatic complexity of greater than 20.