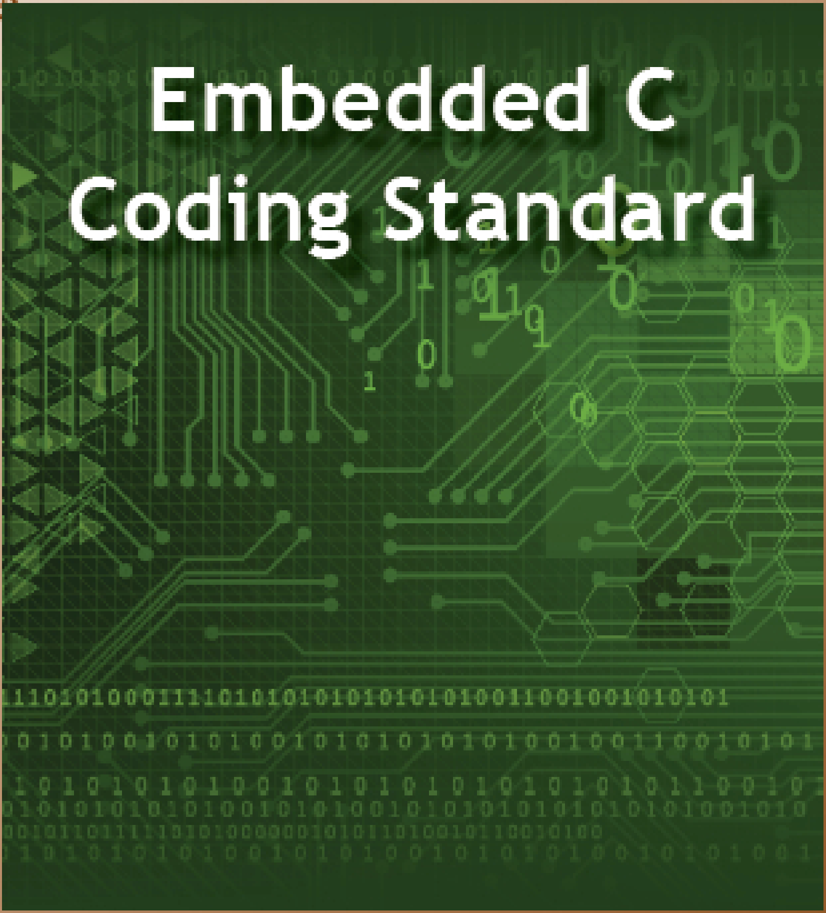
Zero Bugs...Period™



Embedded C Coding Standard

Netrino

Embedded C Coding Standard

Table of Contents

Introduction

Purpose of the Standard

The primary reason for adopting this coding standard is to reduce the number of bugs present in new embedded software (a.k.a., firmware) and in code later added or modified by maintainers. For this reason, specific rules in this document that describe techniques to eliminate or reduce the number of bugs in a program are tagged with the Zero Bugs...Period™ logo below.

Of course, a coding standard cannot by itself eliminate all of the bugs from a complex embedded system. This coding standard is one part of the Zero Bugs… Period™ firmware design methodology. Zero Bugs… Period emphasizes the critical role of developer training, software and system *architecture*, lightweight effective *process*, and the *culture* of a team or organization in keeping bugs out.

Other important reasons for adopting this coding standard include increasing the readability and portability of software, so that firmware may be maintained and reused at lower cost. A coding standard benefits a team of developers and larger organization by reducing the time required by individuals to understand or review the work of peers.

Guiding Principles

This coding standard was developed in accordance with the following guiding principles, which served to focus the authors' attention and eliminate conflict over items that are sometimes viewed by programmers as personal stylistic preferences:

1. Individual programmers do not own the software they write. All software development is work for hire for an employer or a client and, thus, the end product should be constructed in a workmanlike manner.

2. It is cheaper and easier to prevent a bug from creeping into code than it is to find and kill it after it has entered. A key strategy in this fight is to write code in which the compiler, linker, or a static analysis tool can detect bugs automatically—i.e., before the code is allowed to execute.

3. For better or worse (well, mostly worse), the ISO "standard[1]" C programming language allows for a significant amount of variability in the decisions made by compiler implementers. These many so-called "implementation-defined," "unspecified," and "undefined" behaviors, along with "locale-specific options", mean that programs compiled from identical C source code may behave very differently at run-time. Such gray areas in the language standard greatly reduce the portability of C programs that are not carefully crafted.
4. This coding standard prioritizes code reliability and portability above execution efficiency or programmer convenience.
5. There are many sources of bugs in software programs. The original programmer creates some bugs. Other bugs result from misunderstandings by those who later maintain, extend, port, and/or reuse the code.

1 [C90] and [C99]

- The number and severity of bugs introduced by the original programmer can be reduced through disciplined conformance with certain coding practices, such as the placement of constants on the left side of an equivalence (==) test.
- The number and severity of bugs introduced by maintenance programmers can also be influenced by the original programmer. For example, appropriate use of portable fixed-width integer types (e.g., int32_t) ensures that no future port of the code will encounter an unexpected overflow.
- The number and severity of bugs introduced by maintenance programmers can also be reduced through the disciplined use of consistent commenting and stylistic practices, so that everyone in an organization can more easily understand the meaning and proper use of variables, functions, and modules.

6. MISRA's Guidelines for the Use of the C Language[2] are more restrictive than this coding standard—but worthy of study. Deviation from any MISRA-C required or advisory rule should be carefully considered. The authors of the MISRA-C guidelines are knowledgeable of the risks of the use of C in safety-critical systems. Our few known differences of opinion with [MISRA04] are identified in the footnotes to this standard. Followers of Netrino's coding standard may wish to adopt the other rules of MISRA-C in addition to the rules found here.
7. To be effective, coding standards must be enforceable. Wherever two or more competing rules would be similarly able to prevent bugs but only one of those rules can be enforced automatically, the more enforceable rule is recommended.

In the absence of a needed rule or a conflict between rules, the spirit of the above principles should be applied to guide the decision.

[2] [MISRA98] and [MISRA04]

Enforcement Guidelines

Conformance with this coding standard is mandatory. Non-conforming code shall be made to meet these minimum standards. Non-conforming code shall be detected and removed via automated scans, formal code inspections, or informal discovery. All code that is submitted for a release of the software shall conform to these standards, unless a deviation has been permitted.

Deviation Procedure

At the project level, it is acceptable for another coding standard (such as the coding standard of a client or partner) to be adopted instead of this one. In that case, all members of the project team should follow the selected coding standard.

At the module level, it is only acceptable to deviate from this coding standard with the approval of the project manager. The reason for the deviation and the approver's name shall be stated as close as possible to the scope of the deviation. For example, a single deviation in a function should be documented in a block comment within or above that function, whichever is most helpful to the next reader.

Acknowledgements

Though my name is the only one on the front of this book, the development of Netrino's Embedded C Coding Standard was a collaborative effort, involving many at Netrino plus other members of the embedded software community. I am specifically grateful to Nigel Jones, Jack Ganssle, Jean Labrosse, and Miro Samek for publishing earlier firmware coding standards; to Netrino's Joe Perret and Salomon Singer for getting this project underway in a big way; to Andrew Girson for the numerous discussions leading to Zero Bugs…Period; to Elizabeth Gallauresi for handling various details; and to all of the many technical reviewers who provided important feedback, including Mike Ficco, JR Simma, Dan Smith, Michael Wilk, and several of the other folks already identified.

Copyright Notice

1 General Rules

1.1 Which C?

Rules:

a. All programs shall be written to comply with the latest available ISO Standard for the C Programming Language, which is currently [C99].[3]

b. Whenever a C++ compiler is used, appropriate compiler options shall be set to restrict the language to the latest standard C subset supported by the compiler.

c. The use of proprietary compiler language keyword extensions, #pragmas, 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.[4]

---

[3] This deviates from [MISRA04] Rule 1.1. Compilers compatible with [C99] offer many valuable improvements, including support for the fixed-width integer types, C++ style comments, the ability to declare automatic variables where needed, and inline functions. In the absence of a [C99]-compliant compiler, a [C90]-compliant compiler shall be used.

[4] This appears to deviate from [MISRA04] Rule 1.1, but is consistent with Rules 2.1 and 3.4, as well as MISRA-forseen deviations from Rule 1.1.

Reasoning: Even “standard” C varies by compiler, but we need as common of a platform as we can find to make possible the rules and enforcement mechanisms that follow. C++ is a different language and the use of C++ and C should not be mixed in the same design; C++ programmers should consult *Netrino's Embedded C++ Coding Standard*, which differs substantially from this document.

Exceptions: These rules may be ignored in the case that the compiler supports only an older version of the C standard.

Enforcement: These rules shall be enforced during code reviews.

1.2 Line Widths

Rules:

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

Reasoning: Code reviews and other examinations are from time-to-time conducted on printed pages, which must be free of distracting line wraps as well as missing (i.e., past the right margin) characters. Line width rules also ease on-screen side-by-side code differencing.

Exceptions: None.

Enforcement: Violations of this rule shall be detected by an automated scan during each build.

1.3 Braces

Rules:

a. 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.

b. 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.

Reasoning: There is considerable risk associated with the presence of empty statements and single statements that are not surrounded by braces. Code constructs like this are often associated with bugs when nearby code is changed or commented out. This risk is entirely eliminated by the consistent use of braces. The placement of the left brace on the following line allows for easy visual checking for the corresponding right brace.

Exceptions: None.

Enforcement: The appearance of a left brace after each if, else, switch, while, do, and for shall be enforced by an automated tool at build time. The same or another tool shall be used to enforce that all left braces are paired with right braces at the same level of indentation.

1.4 *Parentheses*

Rules:

a. 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.

b. Unless it is a single identifier or constant, each operand of the logical && and || operators shall be surrounded by parentheses.

Example:

```
if ((a < b) && (b < c))
{
    result = (3 * a) + b;
}
return result;
```

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

1.5 Common Abbreviations

Rules:

a. Abbreviations and acronyms should generally be avoided unless their meanings are widely and consistently understood in the engineering community. The table below contains a list of commonly used abbreviations and their meanings.

b. A table of additional project-specific abbreviations and acronyms shall be maintained in a version-controlled document.

Examples: See Appendix C.

Exceptions: Project-specific abbreviations that do not conflict with the common abbreviations in this standard may be added in the manner described above.

Enforcement: Consistent use of these abbreviations shall be enforced during code reviews.

1.6 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);
}

unsigned int  y;
y = adc_read();
z = abs((int) y);  // A risky cast.
```

Reasoning: Casting is dangerous. In the example above, unsigned y can take a larger range of positive values than a signed integer. In that case, the absolute value will be incorrect as well. The above cast was likely used to quiet an important warning about possible loss of precision.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

1.7 Keywords to Avoid

Rules:

a. The *auto* keyword shall not be used.

b. The *register* keyword shall not be used.

c. The *goto* keyword shall not be used.

d. The *continue* keyword shall not be used.

e. The *break* keyword shall not be used outside of a *switch* statement.

Reasoning: The auto keyword is an unnecessary historical feature of the language. Some other features of the C language serve a purpose, but create more headaches than value. For example, the register keyword presumes the programmer is smarter than the compiler. Keywords goto, continue, and break often lead to spaghetti code.

Exceptions: None.

Enforcement: The presence of these keywords in new or modified source code shall be detected and reported via an automated tool at each build.

1.8 *Keywords to Frequent*

Rules:

a. 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.

b. The *const* keyword shall be used whenever appropriate. Examples include:

    i. To declare variables that should not be changed after initialization,

    ii. To define call-by-reference function parameters that should not be modified (e.g., char const * param),

    iii. To define fields in structs and unions that should not be modified (e.g., in a struct overlay for memory-mapped I/O peripheral registers), and

    iv. As a strongly typed alternative to #define for numerical constants.

c. The *volatile* keyword shall be used whenever appropriate. Examples include:

    i. To declare a global variable accessible (by current use or scope) by any interrupt service routine,

ii. To declare a global variable accessible (by current use or scope) by two or more tasks,

iii. 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.

Reasoning: C's static keyword has several meanings. At the module-level, global variables and functions declared static are protected from external use. Heavy-handed use of static in this way thus decreases coupling between modules. The const and volatile keywords are even more important. The upside of using const as much as possible is compiler-enforced protection from unintended writes to data that should be read-only. Proper use of volatile eliminates a whole class of difficult-to-detect bugs by preventing compiler optimizations that would eliminate requested reads or writes to variables or registers.[5]

Exceptions: None.

[5] Anecdotal evidence suggests that programmers unfamiliar with the volatile keyword think their compiler's optimization feature is more broken than helpful and disable optimization. We believe that the vast majority of embedded systems contain bugs waiting to happen due to missing volatile keywords. Such bugs typically manifest themselves as "glitches" or only after changes are made to a "proven" code base.

Enforcement: Appropriate use of these important keywords shall be enforced during code reviews.

2 Comments

2.1 *Acceptable Formats*

Rules:

a. Single-line comments in the C++ style (i.e., preceded by //) are a useful and acceptable alternative to traditional C style comments (i.e., /* … */).[6]

b. Comments shall never be nested.

c. Comments shall never be used to disable a block of code, even temporarily.

   i. To temporarily disable a block of code, use the preprocessor's conditional compilation feature (e.g., #if 0 … #endif). No block of temporarily disabled code shall remain in the source code of a release candidate.

[6] This is a deviation from [MISRA04] Rule 2.2, which we feel will not affect the number or severity of firmware bugs. The C++ "single-line" style makes comments easier to align and maintain. In addition this deviation is consistent with our choice of the [C99] language, which officially added single-line comments to the C language.

ii. Any line or block of code that exists specifically to increase the level of debugging output information shall be surrounded by #ifndef NDEBUG … #endif.[7] In this way, useful debug code may be maintained in production code, as the ability to gather additional information is often desirable long after development is done.

Reasoning: Nested comments and commented-out code both run the risk of allowing unexpected snippets of code to be compiled into the final executable. This can happen, for example, in the case of sequences such as /* code-out /* comment */ code-in */.

Exceptions: None.

Enforcement: The use of only acceptable comment formats can be only partially enforced by the compiler or static analysis. The avoidance of commented-out code, for example, must be enforced during code reviews.

[7] Our choice of negative-logic NDEBUG is deliberate, as that constant is associated with disabling the assert() macro. In both cases, the programmer acts to disable the verbose code. It's also good to have just one of these #defines to keep track of.

2.2 *Location and Content*

Rules:

a. All comments shall be written in clear and complete sentences, with proper spelling and grammar and appropriate punctuation.

b. The most useful comments generally precede a block of code that performs one step of a larger algorithm.[8] A blank line shall follow each such code block. The comments in front of the block should be at the same indentation level.

c. Avoid explaining the obvious. Assume the reader knows the C programming language. For example, end-of-line comments should only be used in exceptional circumstances, 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. Avoid writing unhelpful and redundant comments such as "shift left 2 bits".

[8] It is a good practice to write the comment blocks first, as you should not begin the coding until you can explain the logic, algorithm, or sequence of steps in words.

d. The number and length of individual comment blocks shall be proportional to the complexity of the code they describe.

e. Whenever an algorithm or technical detail has come from a published source, the comment shall include a sufficient reference to the original source (via book title, website URL, or other details) to allow a reader of the code to find the cited reference material.

f. 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.

g. All assumptions shall be spelled out in comments.[9]

h. Each module and function shall be commented in a manner suitable for automatic documentation generation via Doxygen (www.doxygen.org).

[9] Of course, a set of design-by-contract tests or assertions is even better than comments.

i. Use the following capitalized comment markers to highlight important issues:

    i. "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.

    ii. "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.

    iii. "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.
```

Reasoning: Following these rules results in good comments. And good comments result in good code. Unfortunately, it is easy for source code and documentation to drift over time. The best way to prevent this is to keep the documentation as close to the code as possible. Doxygen is a useful tool to regenerate documentation describing the modules, functions, and parameters of a project as that code is changed.

Exceptions: Individual projects may standardize the use of Doxygen features of beyond those in the template files.

Enforcement: The quality of comments shall be evaluated during code reviews. Code reviewers should be on the lookout both that the comments accurately describe the code and that they are clear, concise, and valuable. Rebuilds of Doxygen-generated documentation files, for example in HTML or PDF, shall be automated and made part of the software build process.

3 White Space

3.1 Spaces

Rules:

a. Each of the keywords *if, else, while, for, switch,* and *return* shall always be followed by one space.

b. Each of the assignment operators =, +=, -=, *=, /=, %=, &=, |=, ^=, ~=, and != shall always be preceded and followed by one space.

c. Each of the binary operators +, -, *, /, %, <, <=, >, >=, ==, !=, <<, >>, &, |, ^, &&, and || shall always be preceded and followed by one space.

d. Each of the unary operators +, -, ++, --, !, and ~, shall always be written without a space on the operand side and with one space on the other side.

e. The pointer operators * and & shall be written with white space on each side within declarations but otherwise without a space on the operand side.

f. The ? and : characters that comprise the ternary operator shall each always be preceded and followed by one space.

g. The structure pointer and structure member operators (-> and ., respectively) shall always be without surrounding spaces.

h. The left and right brackets of the array subscript operator ([and]) shall always be without surrounding spaces.

i. Expressions within parentheses shall always have no spaces adjacent to the left and right parenthesis characters.

j. 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.

k. Each comma separating function parameters shall always be followed by one space.

l. Each semicolon separating the elements of a *for* statement shall always be followed by one space.

m. Each semicolon shall follow the statement it terminates without a preceding space.

Reasoning: The placement of white space is as important as the placement of the text of a program. Good use of white space reduces eyestrain and increases the ability of the author and reviewers of the code to spot potential bugs.

Exceptions: None.

Enforcement: These rules shall be enforced by an automated tool such as a code beautifier.

3.2 Alignment

Rules:

a. The names of variables within a series of declarations shall have their first characters aligned.

b. The names of *struct* and *union* members shall have their first characters aligned.

c. The assignment operators within a block of adjacent assignment statements shall be aligned.

d. The # in a preprocessor directive shall always be located in column 1, except when indenting within a #if or #ifdef sequence.

Reasoning: Visual alignment emphasizes similarity. A series of consecutive lines containing variable declarations is easily spotted and understood as a block. Blank lines and unrelated alignments should be used to visually distinguish unrelated blocks of code appearing nearby.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

3.3 Blank Lines

Rules:

a. No line of code shall contain more than one statement.

b. 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.

c. Each source file shall have a blank line at the end.[10]

Reasoning: Appropriate placement of white space provides visual separation and thus makes code easier to read and understand, just as the white space areas between paragraphs of this coding standard aid readability.

Exceptions: None

Enforcement: These rules shall be enforced during code reviews.

[10] This is for portability, as some compilers require the blank line.

3.4 Indentation

Rules:

a. Each indentation level within a module should consist of 4 spaces.

b. Within a *switch* statement, each *case* statement should be indented; the contents of the case block should be indented once more.

c. 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)
    {
        ...
    }
}
```

Reasoning: Fewer indentation spaces increase the risk of visual confusion while more spaces increases the likelihood of line wraps.

Exceptions: The indentation in legacy code modules that are indented differently shall not be changed unless it is anticipated that a significant amount of the code will be modified. In that case, the indentation across the entire module shall be changed in a distinct version control step. This is because a side effect of changing indentation is the loss of difference tracking capability in the version control system. It is thus valuable to separate the code changes from the indent changes.

Enforcement: An automated tool shall be provided to programmers to convert indentations of other sizes automatically. This tool shall modify all new or changed code prior to each build.

3.5 Tabs

Rules:

a. The tab character shall never appear within any module.

Reasoning: The width of the tab character varies by editor and programmer preference, making consistent visual layout a continual source of headaches during code reviews and maintenance.

Exceptions: Existing tabs in legacy code modules shall not be eliminated unless it is anticipated that a significant amount of the code will be modified. In that case, the tabs shall be eliminated from the entire module in a distinct version control step. This is because a side effect of eliminating tabs is the loss of difference tracking capability in the version control system. It is thus valuable to separate the code changes from the white space changes.

Enforcement: The absence of the tab character in new or modified code shall be confirmed via an automated scan at each build.

3.6 Linefeeds

Rules:

a. Whenever possible, all source code lines shall end only with the single character LF (0x0A).

Reasoning: The multi-character sequence CR-LF (0x0D 0x0A) is more likely to cause problems in a multi-platform environment than the single character LF. One such problem is associated with multi-line preprocessor macros on Unix platforms.

Exceptions: None.

Enforcement: Whenever possible, programmer's editors shall be configured to use LF. In addition, an automated tool shall scan all new or modified source code files during each build, replacing each CR-LF sequence with an LF.

4 Modules

4.1 Naming Conventions

Rules:

a. All module names shall consist entirely of lowercase letters, numbers, and underscores. No spaces shall appear within the file name.

b. All module names shall be unique in their first eight characters, with .h and .c used for the suffix for header and source files respectively.

c. No module name shall share the name of a standard library header file. For example, modules shall not be named "stdio" or "math".

d. Any module containing a main() function shall have the word "main" in its filename.

Reasoning: Multi-platform work environments (e.g., Linux and Windows) are the norm rather than the exception. To support the widest range, file names should meet the constraints of the least capable platforms. Additionally, mixed case names are error prone due to the possibility of similarly-named but differently-capitalized files becoming confused. The inclusion of "main" in a file name is an aid to the maintainer that has proven useful.[11]

Exceptions: None.

Enforcement: An automated tool shall confirm that all file names used in each build are consistent with these rules.

[11] We have encountered the case of a company with one project having over 200 files containing a function called main().

4.2 Header Files

Rules:

a. There shall always be precisely one header file for each source file and they shall always have the same root name.

b. Each header file shall contain a preprocessor guard against multiple inclusion, as shown in the example below.

c. The header file shall identify only the procedures, constants, and data types (via prototypes or macros, #define, and struct/union/enum typedefs, respectively) about which it is strictly necessary for other modules to know about.

   i. It is recommended that no variable be defined (via extern) in a header file.

   ii. No storage for any variable shall be declared in a header file.

d. No header file shall contain a #include statement.

Example:

```
#ifndef _ADC_H
#define _ADC_H
...
#endif /* _ADC_H */
```

Reasoning: The C language standard gives all variables and functions global scope by default. The downside of this is unnecessary (and dangerous) coupling between modules. To reduce inter-module coupling, keep as many procedures, constants, data types, and variables as possible hidden within a module's source file.

Exceptions: It is acceptable to deviate from the common root name rule for the core application module (e.g., if "foomain.c" contains main(), its header file may be "foo.h"). Under certain circumstances it may be necessary to share a global variable across modules. Whenever this is done, such a variable shall be named with the module's prefix, declared volatile, and always protected from race conditions at each location of access.

Enforcement: These header file rules shall be enforced during code reviews.

4.3 Source Files

Rules:

a. 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).

b. 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.

c. 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.

d. Absolute paths shall not be used in include file names.

e. Each source file shall be free of unused include files.

f. No source file shall #include another source file.

Reasoning: The purpose and internal layout of a source file module should be clear to all who maintain it. For example, the public functions are generally of most interest and thus appear ahead of the private functions they call. Of critical importance is that every function declaration be matched by the compiler against its prototype.

Exceptions: None.

Enforcement: Prior to each build, an automated tool shall scan source files to ensure they #include their own header file but not unused header files. Lint is an example of a tool that can be configured to perform the second check automatically.

4.4 File Templates

Rules:

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

Reasoning: Starting each new file from a template ensures consistency in file header comment blocks and ensures inclusion of appropriate copyright notices.

Exceptions: None.

Enforcement: The consistency of comment block formats shall be enforced during code reviews.

5 Data Types

5.1 Naming Conventions

Rules:

a. 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'.

b. All new structures, unions, and enumerations shall be named via a typedef.

Example:

```
typedef struct
{
    uint16_t  count;
    uint16_t  max_count;
    uint16_t  _unused;
    uint16_t  control;

} timer_t;
```

Reasoning: Type names and variable names are often appropriately similar. For example, a set of timer control registers in a peripheral calls out to be named 'timer'. To distinguish the structure definition that defines the register layout, it is valuable to create a new type with a distinct name, such as 'timer_t'. If necessary this same type could then be used to create a shadow copy of the timer registers, say called 'timer_shadow'.

Exceptions: It is not necessary to use typedef with anonymous structures and unions.

Enforcement: An automated tool shall scan new or modified source code prior to each build to ensure that the keywords *struct*, *union*, and *enum* are used only within typedef statements or in anonymous declarations.

5.2 *Fixed-Width Integers*

Rules:

a. 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.[12]

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

b. The keywords *short* and *long* shall not be used.

c. Use of the keyword *char* shall be restricted to the declaration of and operations concerning strings.

[12] If program execution speed is also a consideration, note that the [C99] standard defines a set of "fastest" fixed-width types. For example, *uint_fast8_t* is the fastest integer type that can hold at least 8 bits of unsigned data.

Reasoning: The [C90] standard allows implementation-defined widths for *char*, *short*, *int*, *long*, and *long long* types, which leads to portability problems. The [C99] standard did not resolve this, but introduced the uniform type names shown in the table, which are defined in the C99 header file <stdint.h>.

Exceptions: In the absence of a C99-compliant compiler, it is acceptable to define the set of fixed width types in the table above as typedefs based on *char*, *short*, *int*, *long*, and *long long*. If this is done, use compile-time checks (such as static assertions) to have the compiler flag incorrect type definitions. It is acceptable to use the native types when C Standard Library functions are used—just be careful.

Enforcement: At every build an automated tool shall scan for and flag the use of the keywords *short* and *long*, which are not to be used. Compliance with the other rules shall be checked during code reviews.

5.3 Signed Integers

Rules:

a. Bit-fields shall not be defined within signed integer types.

b. None of the bit-wise operators (i.e., &, |, ~, ^, <<, and >>) shall be used to manipulate signed integer data.

c. 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:

```
uint8_t   a = 6u;
int8_t    b = -9;

if (a + b < 4)
{
    // This correct path should be executed
    // if -9 + 6 were -3 < 4, as anticipated.
}
else
{
    // This incorrect path is actually executed,
    // as -9 + 6 becomes (0xFF – 9) + 6 = 252.
}
```

Reasoning: Several details of the manipulation of binary data within signed integer containers are implementation-defined behaviors of the C standard. Additionally, the results of mixing signed and unsigned data can lead to data-dependent bugs.

Exceptions: None.

Enforcement: Static analysis tools can be used to detect violations of these rules.

5.4 Floating Point

Rules:

a. Avoid the use of floating point constants and variables whenever possible. Fixed-point math may be an alternative.

b. 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 = 3.1415927f).

   iii. Ensure that the compiler supports double-precision, if your math depends on it.

   iv. Never test for equality or inequality of floating point values.

Example:

```
// Ensure the compiler supports double-precision.
#include <limits.h>
#if (DBL_DIG < 10)
    #error "Double precision is not available!"
#endif
```

Reasoning: There are a large number of risks of errors stemming from incorrect use of floating point arithmetic; these are outside the scope of this document.[13] By default, C promotes all floating-point constants to double precision, which may be inefficient or unsupported on the target platform. However, many microcontrollers do not have any hardware support for floating point math. The compiler may not warn of these incompatibilities, instead performing the requested numerical operations by linking in a large (typically a few kilobytes of code) and slow (numerous instruction cycles per operation) floating-point emulation library.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

[13] [Seacord] has a nice explanation of these issues and some suggested guidelines in Chapter 6.

5.5 Structures and Unions

Rules:

a. 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.

b. Appropriate care shall be taken to prevent the compiler from altering the intended order of the bits within bit-fields.[14]

Reasoning: There is a tremendous amount of implementation-defined behavior in the area of structures and unions. Bit-fields, in particular, suffer from severe portability problems, including the lack of a standard bit ordering and no official support for the fixed-width integer types they so often call out to be used with.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

[14] Options include static assertions or other compile-time checks as well as the use of preprocessor directives to select one of two competing struct definitions.

6 Procedures

6.1 *Naming Conventions*

Rules:

a. No procedure 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*, *class*, *true*, *false*, *public*, *private*, *friend*, *protected*, and many others.

b. No procedure shall have a name that overlaps a function in the C standard library. Examples of such names include strlen, atoi, and memset.

c. No procedure shall have a name that begins with an underscore.

d. No procedure name shall be longer than 31 characters.[15]

e. No function name shall contain any uppercase letters.

f. No macro name shall contain any lowercase letters.

[15] Rule 11 (required) of [MISRA98].

g. Underscores shall be used to separate words in procedure names.

h. 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()).

i. The names of all public functions shall be prefixed with their module name and an underscore (e.g., force_read()).

Reasoning: Good function names make reviewing and maintaining code easier (and thus cheaper). The data (variables) in programs are nouns. Functions manipulate data and are thus verbs. The use of module prefixes is in keeping with the important goal of encapsulation and helps avoid procedure name overlaps.

Exceptions: None.

Enforcement: Compliance with these naming rules shall be established in the detailed design phase and be enforced during code reviews.

6.2 *Functions*

Rules:

a. All reasonable effort shall be taken to keep the length of each function limited to one printed page, or about 50-100 lines.

b. 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.

c. All functions shall have just one exit point and it shall be at the bottom of the function. That is, the keyword *return* shall appear a maximum of once.[16]

d. A prototype shall be defined for each public function in the module header file.

e. All private functions shall be defined *static.*

f. Each parameter shall be explicitly declared and meaningfully named.

[16] In fact, [IEC61508] requires it.

Example:

```
int
state_change (int event)
{
    int result = ERROR;

    if (EVENT_A == event)
    {
         result = STATE_A;
         // Don't return here.
    }
    else
    {
        result = STATE_B;
    }

    return (result);
}
```

Reasoning: Code reviews take place at the function level. Each function should be visible on a single printed page, so that flipping back and forth (a distraction) is not necessary. Similarly, multiple exit points are distracting to reviewers and thus do more harm than good to readability.

Exceptions: None.

Enforcement: Compliance with these rules shall be checked during code reviews.

6.3 Function-Like Macros

Rules:

a. Parameterized macros shall not be used if an inline function can be written to accomplish the same task.[17]

b. 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.

   iii. Use each argument no more than once, to avoid unintended side effects.

Example:

```
// Don't do this ...
#define MAX(A, B)   ((A) > (B) ? (A) : (B))

// ... if you can do this instead.[18]
inline int max(int a, int b)
```

[17] [C99] formally added the C++ keyword *inline* to C.

[18] Note that individual functions will be needed to support each base type for comparison.

Reasoning: There are a lot of risks associated with the use of preprocessor #defines, and many of them relate to the creation of parameterized macros. The extensive use of parentheses (as shown in the example) is important, but does not eliminate the unintended double increment possibility of a call such as MAX(i++, j++). Other risks of macro misuse include comparison of signed and unsigned data or any test of floating-point data. Making matters worse, macros are invisible at run-time and thus impossible to step into within the debugger.[19]

Exceptions: In the case of necessary (and tested, and documented) efficiency, a local exception can be approved. That's when the other rules kick in.

Enforcement: The avoidance of and safe use of macros shall be enforced during code reviews.

[19] Of course, *inline* functions are also invisible at debug time.

6.4 Tasks

Rules:

a. All functions that represent tasks (a.k.a., threads) shall be given names ending with "_task" (or "_thread").

Example:

```
void
alarm_task (void * p_data)
{
    alarm_t  alarm = ALARM_NONE;
    int      err   = OS_NO_ERR;

    for (;;)
    {
        alarm = OSMboxPend(alarm_mbox, &err);
        // Process alarm here.
    }
}
```

Reasoning: Each task in a real-time operating system (RTOS) is like a mini-main(), typically running forever in an infinite loop. It is valuable to easily identify these important functions during code reviews and debugging sessions.

Exceptions: Alternatively, "_thread" may be used.

Enforcement: This naming convention shall be enforced during the detailed design phase and in code reviews.

6.5 Interrupt Service Routines

Rules:

a. 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".

b. All functions that implement ISRs shall be given names ending with "_isr".

c. 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 lack a prototype, be declared *static*, and be located at the end of the associated driver module.[20]

d. 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().[21]

[20] Be forewarned that a smart static analsys tool, such as lint, will likely complain about this unreachability.

[21] Although this doesn't prevent any bugs, it sure does help find bugs in the hardware and makes the system more robust.

Reasoning: An ISR is an extension of the hardware. By definition, it and the straight-line code are asynchronous to each other. If they share global data, that data must be protected with interrupt disables in the straight-line code. The ISR must not get hung up inside the operating system or waiting for a variable or register to change value.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

7 Variables

7.1 *Naming Conventions*

Rules:

a. 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*.

b. No variable shall have a name that overlaps with a variable name from the C standard library (e.g., errno).

c. No variable shall have a name that begins with an underscore.

d. No variable name shall be longer than 31 characters.

e. No variable name shall be shorter than 3 characters, including loop counters.[22]

f. No variable name shall contain any uppercase letters.

[22] This is because you can't do a meaningful global search for "i".

g. 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.

h. Underscores shall be used to separate words in variable names.

i. Each variable's name shall be descriptive of its purpose.

j. The names of all global variables shall begin with the letter 'g'. For example, g_zero_offset.

k. The names of all pointer variables shall begin with the letter 'p'. For example, *p_led_reg.

l. The names of all pointer-to-pointer variables shall begin with the letter pp. For example, gpp_vector_table.

m. The names of all integer variables containing "effectively Boolean" information (i.e., 0 vs. non-zero) shall begin with the letter 'b' and phrased as the question they answer.[23] For example, b_done_yet or gb_is_buffer_full.

[23] It is unsafe, in C, to define constants such as TRUE and FALSE where TRUE is equal to 1. However, it is safe to compare "effectively Boolean" integer values against 0. For example, the test "if (!gb_buffer_is_full)" is safe and consistent with Rule 13.2

Reasoning: The base rules are adopted to maximize code portability across compilers. Many C compilers recognize differences only in the first 31 characters in a variable's name and reserve names beginning with an underscore for internal names. The other rules are meant to highlight risks and ensure consistent proper use of variables.

Exceptions: None.

Enforcement: These variable-naming rules shall be enforced during code reviews.

(advisory) of [MISRA04]. Note too that [C99] adds <stdbool.h>, which defines a *bool* type and constants *true* and *false*.

7.2 *Initialization*

Rules:

a. All variables shall be initialized before use.

b. It is preferable to create variables as you need them, rather than all at the top of a function.[24]

Example:

```
for (int loop = 0; loop < MAX_LOOPS; loop++)
{
    ...
}
```

Reasoning: Too many programmers assume the C run-time will watch out for them. This is a very bad assumption, which can prove dangerous in a real-time system. It is easier to initialize some variables closer to their use, and this also aids readability of the code.[25]

Exceptions: None.

---

[24] Yet another handy feature allowed by [C99] but not in [C90].

[25] One study of back-and-forth eye movements during code reviews ([Uwano]) demonstrated the importance of placing variable declarations as close as possible to the code that uses them.

Enforcement: An automated tool shall scan all of the source code prior to each build, to warn about variables used prior to initialization. Lint is an example of a tool that can do this well.

8 Expressions and Statements

8.1 *Variable Declarations*

Rules:

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

Example:

```
char * x, y;   // Is y supposed to be a pointer?
```

Reasoning: The cost of placing each declaration on a line of its own is low. By contrast, the risk that either the compiler or a maintainer will misunderstand your intentions is high.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

8.2 *If-Else Statements*

Rules:

a. The shortest (measured in lines of code) of the *if* and *else if* clauses should be placed first.[26]

b. Nested *if-else* statements shall not be deeper than two levels. Use function calls or switch statements to reduce complexity and aid understanding.

c. Assignments shall not be made within an *if* or *else if* expression.

d. Any *if* statement with an *else if* clause shall end with an *else* clause.[27]

[26] Thanks to [Holub] for putting this "formatting for readability" idea into words.

[27] This is the equivalent of requiring a default case in every switch, as we do below.

Example:

```
if (NULL == p_object)
{
    result = ERR_NULL_PTR;
}
else if (p_object = malloc(sizeof(object_t))) // No!
{
    ...
}
else
{
    // Normal processing steps,
    // which require many lines of code.
    ...
}
```

Reasoning: Long clauses can distract the human eye from the decision-path logic. By putting the shorter clause earlier, the decision path becomes easier to follow. (And easier to follow is always good for reducing bugs.) Deeply nested *if* blocks are a sure sign of a complex and fragile state machine implementation; there is always a safer and more readable way to do the same thing.

Exceptions: For efficiency purposes, it may be desirable to reorder the sequence of *if-else* clauses to ensure the most frequent or most critical case is always found the fastest. Of course, *if-else* statements are typically not as efficient as tables of function pointers in terms of worst-case analysis.

Enforcement: These rules shall be enforced during code reviews, when reviewers feel it may aid readability.

8.3 *Switch Statements*

Rules:

a. The *break* for each case shall be indented to align with the associated *case*, rather than with the contents of the case code block.

b. All *switch* statements shall contain a *default* block.

Example:

```
switch (err)
{
   case ERR_A:
      ...
   break;

   case ERR_B:
      ...
   break;

   default:
      ...
   break;
}
```

Reasoning: Switch statements are powerful, but prone to errors such as missed break statements and unhandled cases. By aligning the *case* and *break* keywords it is possible to spot missing *breaks*.

Exceptions: None.

Enforcement: These rules can be enforced by an automated scan of all new or modified code during each build. Alternatively, they shall be enforced in code reviews.

8.4 Loops

Rules:

a. Magic numbers shall not be used as the initial value or in the endpoint test of a *while* or *for* loop.[28]

b. Except for a single loop counter initialization in the first clause of a *for* statement, assignments shall not be made in any loop's controlling expression.

c. Infinite loops shall be implemented via the controlling expression "for (;;)".[29]

d. 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:

```
// Don't use a magic number ...
for (int row = 0; row < 100; row++)
```

[28] Note that sizeof() is a theoretically handy way to dimension an array but that this method does not work when you pass a pointer to the array instead of the array itself. Thus the most portable method is a constant shared between the array declaration and the loop.

[29] We can make no compelling technical argument against while (1), but note that Kernighan & Ritchie have long recommended for (;;) and so many C programmers have become accustomed to structuring their infinite loops this way. In such a case, consistency is all that really matters.

```
{
    // ... when you mean a constant.
    for (int col = 0; col < MAX_COL; col++)
    {
        ...
    }
}
```

Reasoning: It is always important to synchronize the number of loop iterations to the size of the underlying data structure. Doing this through a single constant prevents a whole class of bugs that can result when changes in one part of the code, such as the dimension of an array, are not matched by changes in other areas of the code, such as a loop iterator that operates on the array. The use of named constants also makes the code easier to read and maintain.

Exceptions: It is acceptable to start or end a loop with an integer value of 0 (considered less magical by most), such as when iterating from or toward the base of an array.

Enforcement: These rules shall be enforced during code reviews.

8.5 Unconditional Jumps

Rules:

a. As stated earlier, the keywords *goto*, *continue*, and *break* shall not be used to create unconditional jumps.

Reasoning: Algorithms that utilize unconditional jumps to move the instruction pointer can be rewritten in a manner that is more readable and thus easier to maintain.

Exceptions: None.

Enforcement: These rules shall be enforced by an automated scan of all modified or new modules for inappropriate use of these tokens.

8.6 Equivalence Tests

Rules:

a. When evaluating the equality or inequality of a variable with a constant value, always place the constant value on the left side of the comparison operator, as shown in the *if-else* example above.

Reasoning: It is always desirable to detect possible typos and as many other bugs as possible at compile-time; run-time discovery may be dangerous to the user of the product and require significant effort to localize. By following this rule, the compiler can detect erroneous attempts to assign (i.e., = instead of ==) a new value to a constant.

Exceptions: None.

Enforcement: A static analysis tool shall be configured to raise an error or warning about all assignment statements where comparisons are ordinarily expected.

Bibliography

[Barr]	Barr, Michael and Anthony Massa. "Programming Embedded Systems with C and GNU Development Tools," 2nd Edition. O'Reilly, 2006.
[C90]	"ISO/IEC9899:1990, Programming Languages – C," ISO, 1990.
[C99]	"ISO/IEC9899:1999, Programming Languages – C," ISO, 1999.
[Dictionary]	Ganssle, Jack G. and Michael Barr. "Embedded Systems Dictionary." CMP Books, 2003.
[Ganssle]	Ganssle, Jack G. "A Firmware Development Standard: Version 1.4," The Ganssle Group, May 2007.
[Hatton]	Hatton, Les. "Safer C: Developing Software for High-Integrity and Safety-Critical Systems." McGraw-Hill, 1994.
[Holub]	Holub, Allen I. "Enough Rope to Shoot Yourself in the Foot: Rules for C and C++ Programming." McGraw-Hill, 1995.

[IEC61508] "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems," International Electromechanical Commission, 1998-2000.

[Koenig] Koenig, Andrew. "C Traps and Pitfalls." Addison-Wesley, 1988.

[MISRA98] "Guidelines for the Use of the C Language in Vehicle Based Software," The Motor Industry Software Reliability Association, April 1998.

[MISRA04] "MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems," The Motor Industry Software Reliability Association, October 2004.

[Prinz] Prinz, Peter and Ulla Kirch-Prinz. "C Pocket Reference." O'Reilly, 2003.

[Seacord] Seacord, Robert C. "The CERT Secure C Coding Standard." Addison-Wesley, 2009.

[Uwano] Uwano, H., Nakamura, M., Monden, A., and Matsumoto, K. "Analyzing Individual Performance of Source Code Review Using Reviewer's Eye Movement," *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, San Diego, March 27-29, 2006.

Appendix A: Header File Template

```
/** @file module.h
*
* @brief A description of the module’s purpose.
*
* @par
* COPYRIGHT NOTICE: (c) 2009 Netrino, LLC.
* All rights reserved.
*/

#ifndef _MODULE_H
#define _MODULE_H

#ifdef __cplusplus
extern “C” {
#endif

int8_t max(int8_t a, int8_t b);

#ifdef __cplusplus
}
#endif

#endif /* _MODULE_H */
```

Appendix B: Source File Template

```
/** @file module.c
*
* @brief A description of the module's purpose.
*
* @par
* COPYRIGHT NOTICE: (c) 2009 Netrino, LLC.
* All rights reserved.
*/

#include "module.h"

/*!
* @brief A description of the function's purpose.
* @param[in] a Description of a.
* @param[in] b Description of b.
* @return int8_t
*/
int8_t
max (int8_t /*a*/, int8_t /*b*/)
{
    return ((a > b) ? a : b);
}
```

Appendix C: Standard Abbreviations

Abbreviation	Meaning
adc	analog-to-digital converter
avg	average
b_	boolean (i.e., 0 or non-zero)
buf	buffer
cfg	configuration
cmp	compare
curr	current
dac	digital-to-analog converter
ee	(serial) EEPROM
err	error
g_	global
gpio	general-purpose I/O pins
h_	handle (to)

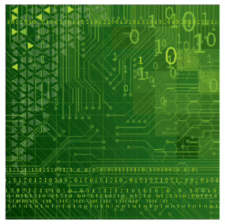
init	initialize
io	input/output
isr	interrupt service routine
lcd	liquid crystal display
led	light-emitting diode
max	maximum
mbox	mailbox
mgr	manager
min	minimum
msec	millisecond[30]
msg	message
next	next (item in a list)
nsec	nanosecond

[30] Note that second(s) shall not be abbreviated, nor minute, hour, day, week, month, or year. Among other things, this rule eliminates conflict between minute and minimum (for "min").

num	number (of)
p_	pointer (to)
pp_	pointer to a pointer (to)
prev	previous (item in a list)
prio	priority
pwm	pulse width modulation
q	queue
reg	register
rx	receive
sem	semaphore
str	string (null terminated)
sync	synchronize
temp	temperature
tmp	temporary
tx	transmit

usec	microsecond

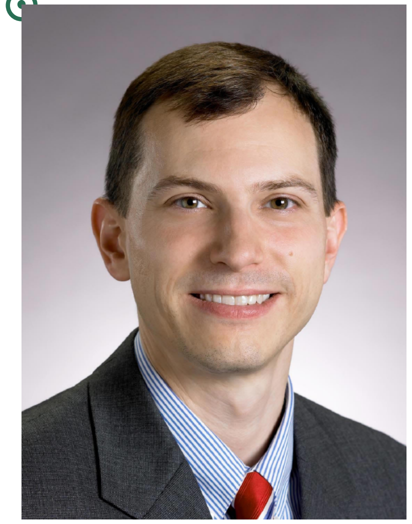
Embedded C Coding Standard



The primary reason for adopting this coding standard is to reduce the number of bugs present in new embedded software (a.k.a., firmware) and in code later added or modified by maintainers. Of course, a coding standard cannot by itself eliminate all of the bugs from a complex embedded system. This coding standard is rather a part of Netrino's Zero Bugs...Period™ firmware design methodology.

The Zero Bugs...Period (ZBP) methodology is a lightweight system of best practices to keep bugs out of embedded software. ZBP solves the recurring problems of missed schedules and buggy firmware. Adherents reduce bugs and put their schedules back on track by applying ZBP's overlapping layers of development, architecture, process, and culture improvements. ZBP emphasizes the importance of software and system architecture as well as programmer skills training and suggests key elements of process, such as design reviews, code reviews, and version control.

Other important reasons for adopting this coding standard include increasing the readability and portability of software, so that firmware may be maintained and reused at lower cost. A coding standard benefits a team of developers and larger organization by reducing the time required by individuals to understand or review the work of peers.



Michael Barr

Michael Barr is an internationally recognized expert on the design of embedded computer systems. In that role, he has provided expert witness testimony in federal court, appeared on PBS' *American Business Review*, and been quoted in various newspapers.

He is also the author of two books and more than forty articles on related subjects and is the creator of Netrino's Zero Bugs...Period design methodology. For three and a half years Michael served as editor-in-chief of Embedded Systems Programming.

In addition, Michael has been a member of the advisory board of the Embedded Systems Conference. Software he wrote continues to power millions of products. Michael holds B.S. and M.S. degrees in electrical engineering and has lectured in the Department of Electrical and Computer Engineering at the University of Maryland, from which he also earned an MBA.

Zero Bugs...Period™

Embedded C Coding Standard

Netrino
www.netrino.com