Bug-Killing Coding Standard Rules for Embedded C

Mon, 2009-03-30 12:44 - webmaster

by Michael Barr (http://www.barrgroup.com/Embedded-Systems/How-To/Bug-Killing-Standards-for-Embedded-C)

A C coding standard can help keep bugs out of embedded software by leveraging common language features and development tools.

A coding standard defines a set of rules for programmers to follow in a given language. This makes a coding standard similar in purpose to the English standard known as Strunk & White (*The Elements of Style*). The original version of this popular book of grammar and other rules for written English was once described as a "forty-three-page summation of the case for cleanliness, accuracy, and brevity in the use of English."

Embedded software is also better when rules for "cleanliness, accuracy, and brevity" are followed. The adoption of a coding standard by a team or a company has many benefits. For example, a coding standard increases the readability and portability of software, so that firmware may be maintained and reused at lower cost. A coding standard also benefits a team of software developers and their entire organization by reducing the time required by individual team members to understand or review the work of peers.

In my view, though, one of the biggest potential benefits of a coding standard has been too long overlooked: a coding standard can help keep bugs out. It's cheaper and easier to prevent a bug from creeping into code than it is to find and kill it after it has entered. Thus, a key strategy for keeping the cost of firmware development down is to write code in which the compiler, linker, or a static-analysis tool can keep bugs out automatically—in other words, before the code is allowed to execute.

Of course, there are many sources of bugs in software programs. The original programmer creates some of the bugs, a few lurking in the dark shadows only to emerge months or years later. Additional bugs result from misunderstandings by those who later maintain, extend, port, and/or reuse the code.

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 each equivalence (==) test.

The original programmer can also influence the number and severity of bugs introduced by maintenance programmers. For example, appropriate use of portable fixed-width integer types (such as **int32_t**) ensures that no future port of the code to a new compiler or target processor 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.

Available C standards

Over the years, I have reviewed the details of many coding standards, including several specifically aimed at the use of C for firmware development. I have also studied MISRA's 2004 "Guidelines for the Use of the C Language in Safety-Critical Systems." The authors of "MISRA-C" are knowledgeable of the risks in safety-critical system design and their guidelines narrow the C language to a safer subset. (For more information about the MISRA-C Guidelines, see Nigel Jones's article "MISRA-C Guidelines for Safety Critical Software".)

Unfortunately, although firmware coding standards and MISRA-C sometimes overlap, coding standards too often focus primarily on stylistic preferences—missing their chance to help reduce bugs. MISRA-C, by contrast, offers great advice for eliminating bugs but very little guidance on practical day-to-day issues of style.

It was thus out of necessity that my team of engineers developed the *Barr Group Embedded C Coding Standard*. This coding standard was created from the ground up to help keep bugs out of firmware. In addition, we applied the following guiding principles, which served to eliminate conflict over items that are sometimes viewed by individual team members as personal stylistic preferences:

- 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.
- For better or worse (well, mostly worse), the ANSI/ISO "standard" 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" (see Appendix G of the standard), mean that programs compiled from identical C source code may behave very

differently at run time. These gray areas in the language greatly reduce the portability of C programs that are not carefully crafted.

- The reliability and portability of code are more important than either execution efficiency or programmer convenience.
- The MISRA-C guidelines were carefully crafted and are worthy of study. On the small number of issues where we have a difference of opinion with MISRA-C, we make this clear. Of course, followers of Barr Group's coding standard may wish to adopt the other rules of MISRA-C in addition.
- To be effective in keeping out bugs, 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.

Ten bug-killing rules for C

Here are some examples of coding rules you can follow to reduce or eliminate certain types of firmware bugs.

Rule #1 - Braces

Braces ({ }) shall always surround the blocks of code (also known as compound statements) following **if**, **else**, **switch**, **while**, **do**, and **for** keywords. Single statements and empty statements following these keywords shall also always be surrounded by braces.

```
// Don't do this ...
if (timer.done)
   // A single statement needs braces!
   timer.control = TIMER_RESTART;

// Do this ...
while (!timer.done)
{
   // Even an empty statement should be surrounded by braces.
}
```

Reasoning: Considerable risk is associated with the presence of empty statements and single statements that are not surrounded by braces. Code constructs of this type are often associated with bugs when nearby code is changed or commented out. This type of bug is entirely eliminated by the consistent use of braces.

Rule #2 - Keyword "const"

The **const** keyword shall be used whenever possible, including:

- To declare variables that should not be changed after initialization,
- To define call-by-reference function parameters that should not be modified (for example, **char const * p_data**),
- To define fields in structs and unions that cannot be modified (such as in a struct overlay for memory-mapped I/O peripheral registers), and
- As a strongly typed alternative to **#define** for numerical constants.

Reasoning: The upside of using **const** as much as possible is compiler-enforced protection from unintended writes to data that should be read-only.

Rule #3 - Keyword "static"

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.

Reasoning: C's **static** keyword has several meanings. At the module-level, global variables and functions declared **static** are protected from inadvertent access from other modules. Heavy-handed use of **static** in this way thus decreases coupling and furthers encapsulation.

Rule #4 - Keyword "volatile"

The **volatile** keyword shall be used whenever appropriate, including:

- To declare a global variable accessible (by current use or scope) by any interrupt service routine,
- To declare a global variable accessible (by current use or scope) by two or more tasks, and
- To declare a pointer to a memory-mapped I/O peripheral register set (for example,**timer_t volatile * const p_timer**).[1]

Reasoning: Proper use of **volatile** eliminates a whole class of difficult-to-detect bugs by preventing the compiler from making optimizations that would eliminate requested reads or writes to variables or registers that may be changed at any time by a parallel-running entity.[2]

Rule #5 - Comments

Comments shall neither be nested nor used to disable a block of code, even temporarily. To temporarily disable a block of code, use the preprocessor's conditional compilation feature (for example, #**if 0 ... #endif**).

```
// Don't do this ...
/*
 a = a + 1;
 /* comment */
 b = b + 1;
*/

// Do this ...
#if 0
 a = a + 1;
 /* comment */
 b = b + 1;
#endif
```

Reasoning: Nested comments and commented-out code both run the risk of allowing unexpected snippets of code to be compiled into the final executable.

Rule #6 - Fixed-width data types

Whenever the width, in bits or bytes, of an integer value matters in the program, a fixed-width data type 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 Table 1.

Standard for signed and unsigned fixed-width integer types.

Integer width	Signed type	Unsigned type
8 bits / 1 byte	int8_t	uint8_t
16 bits / 2 bytes	int16_t	uint16_t
32 bits / 4 bytes	int32_t	uint32_t
64 bits / 8 bytes	int64_t	uint64_t

Table 1

Reasoning: The ISO C standard allows implementation-defined widths for **char**, **short**, **int**,**long**, and **long long** types, which leads to portability problems. Though the 1999 standard did not change this underlying issue, it did introduce the uniform type names

shown in the table, which are defined in the new header file **<stdint.h>**. These are the names to use even if you have to create the typedefs by hand.

Rule #7 - Bit-wise operators

> None of the bit-wise operators (in other words, **&**, |, ~, ^, <<, and >>) shall be used to manipulate signed integer data.

```
// Don't do this ...
int8_t  signed_data = -4;
signed_data >>= 1;  // not necessarily -2
```

Reasoning: The C standard does not specify the underlying format of signed data (for example, 2's complement) and leaves the effect of some bit-wise operators to be defined by the compiler author.

Rule #8 - Signed and unsigned integers

> 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.

```
// Don't do this ...
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 because -9 + 6 becomes
   // (0x100 - 9) + 6 = 253.
}
```

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.

Rule #9 - Parameterized macros vs. inline functions

> Parameterized macros shall not be used if an inline function can be written to accomplish the same task.[3]

```
// Don't do this ...
#define MAX(A, B)   ((A) > (B) ? (A) : (B))
// ... if you can do this instead.
inline int max(int a, int b)
```

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 doesn't 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.

Rule #10 - Comma operator

> The comma (,) operator shall not be used within variable declarations.

```
// Don't do this ...
char * x, y; // did you want y to be a pointer or not?
```

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.

Add the above rules to your coding standard to keep bugs out of your firmware. And read the follow-on article More Bug-Killing Coding Standard Rules for Embedded C. Then follow my blog at embeddedgurus.net/barr-code to learn more techniques for bug-proofing embedded systems.

Endnotes

1. Complex variable declarations like this can be difficult to comprehend. However, the practice shown here of making the declaration read "right-to-left" simplifies the translation into English. Here, "**p_timer** is a constant pointer to a volatile **timer_t**" register set. That is, the address of the timer registers is fixed while the contents of those registers may change at any time. [back]

2. Anecdotal evidence suggests that programmers unfamiliar with the **volatile** keyword think their compiler's optimization feature is more broken than helpful and disable optimization. I suspect, based on experience consulting with numerous companies, that the vast majority of embedded systems contain bugs waiting to happen due to a shortage of **volatile**keywords. These kinds of bugs often exhibit themselves as "glitches" or only after changes are made to a "proven" code base. [back]

3. The C++ keyword **inline** was added to the C standard in the 1999 ISO update. [back]