MISRA-C Guidelines for Safety Critical Software

In 1998, the UK's Motor Industry Software Reliability Association established a set of 127 guidelines for the use of C in safety- critical systems. Here's a look at the rules, what they mean, and how they can work for you.

The reasons for the popularity of C in the embedded realm are clear: easy access to hardware, low memory requirements, and efficient run-time performance being foremost among them. Equally well known are the problems with C: highly limited run-time checking, a syntax that is prone to stupid mistakes that are technically legal, and a myriad of areas where the ISO C standard explicitly states that the behavior is either implementation defined or undefined.

In the hands of a truly experienced programmer, most of the limitations of C can be avoided. Of course, that leaves the problem of inexperienced programmers. The problem is similar to the dilemma faced by parents when their offspring learn to drive. The solution that most parents adopt is to give their child a large, slow lumbering vehicle with excellent brakes; few hand over the keys to a high performance sports car.

ISO C is like a high performance vehicle–with very questionable brakes. Despite this, organizations have little choice other than to let their beginners loose with the language. The results are predictable and well documented.

So, what can be done? The UK-based Motor Industry Software Reliability Association (MISRA) realized that in many areas of an automobile design, safety is of paramount importance. They also recognized that C was the de facto language for coding such systems and that these systems are, therefore, vulnerable to C's limitations. Rather than mandating the use of a safer language such as Ada, the organization looked at ways to make C safer.

The result was a set of "Guidelines for the Use of the C Language in Vehicle-Based Software," or "MISRA C," as they are more informally known. The guidelines, which were first published in 1998, comprise a 70-page document that describes a workable subset of C that avoids many of its well-known problems. (Unfortunately, you have to buy the document from MISRA. It is available at www.misra.org.uk for about $50.)

The MISRA C document contains eight chapters and two appendices and was obviously written by experienced embedded programmers. Chapters 1 through 6 contain important information about the rationale for MISRA C and how to interpret the rules. These chapters should be read prior to diving into

the actual rules, which are found in Chapter 7.

Safety guidelines

In all, MISRA C has 127 rules. Of these, 93 are required and the remaining 34 are advisory. The distinction between these two types of rules is important. C code that claims conformance to MISRA C must comply with all 93 required rules. Conforming code should adhere to the advisory rules as much as is practical. In other words, either you buy into the spirit of MISRA C or you don't.

With this in mind, it's time to discuss the rules themselves. The example rules are quoted verbatim, in part to illustrate the tone and style of the guidelines. Let's start at the beginning:

Rule 1 (required): All code shall conform to ISO 9899 standard C, with no extensions permitted.

When I first read this, I laughed out loud. Because the C standard was not originally written with embedded systems in mind, it is impossible to write a non-trivial embedded program without resorting to a variety of compiler extensions. For example, ISO C provides no way to specify that a function is an interrupt service routine. (MISRA frowns upon assembly language, so you can't use that as a way to skirt the issues.)

Fortunately, in reading the notes associated with Rule 1, one finds the following comment:

It is recognized that it may be necessary to raise deviations (as described in section 5.3.2) to permit certain language extensions, for example to support hardware specific features.

The deviations mentioned in the comment are, arguably, the strongest feature of MISRA C. In putting together the rules, the authors recognized that they could not possibly foresee all the circumstances associated with all embedded systems. Thus, they put in place a mechanism by which an organization can formally deviate from a given rule. Note that this does not mean that the programmer is free to violate the rules on a whim. Rather it means that all violations must be carefully considered, justified, and documented.

MISRA also recommends that deviations be localized as much as possible–such as, all I/O operations be performed in one file. Notwithstanding MISRA C, this is good programming practice anyway.

Most of the other rules are common sense and are succinctly stated. For instance:

Rule 20 (required): All object and function identifiers shall be declared before use.

Any programmer who has a problem with this rule probably needs to find a new line of work.

Some of the rules are more stylistic in their intent and, as such, are more likely to offend individual sensibilities. This is particularly true of the advisory rules, such as:

Rule 49 (advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

This rule goes against my own preferred style of coding. However, I see where the authors are going and wouldn't exactly be compromising deeply held beliefs by conforming to their advice.

While Rule 49 isn't going to change much in the way most people write code, Rule 104 could have a bigger impact.

Rule 104 (required): Non-constant pointers to functions shall not be used.

The use of pointers to functions is a favored technique of many experienced embedded programmers.

The bottom line: regardless of your experience level, if you choose to go down the MISRA C path, expect to have to change the way you code in both big and small ways.

Compliance checking

In writing these rules, the authors appear to have tried very hard to make compliance checking via static analysis possible for as many rules as possible. Static analysis is a fancy term meaning that a computer program could be written to check for violations of the rules.

The most obvious program to do the compliance checking is a compiler. In fact, several embedded compiler vendors, including Green Hills and Tasking (now Altium), do offer compilers with MISRA compliance switches. Does this mean that you have to use a subset of the compilers out there in order to get compliance checking? Fortunately, no! PC-Lint (from Gimpel) now offers MISRA compliance checking as well.

Unfortunately, though, not all of the rules can be enforced automatically. For the other 23 rules, the only available enforcement mechanism is a code review. So, if nothing else, adherence to MISRA C guarantees that your organization will have to conduct code reviews.

MISRA recommends the use of a compliance matrix to track how your organization intends to check its conformance to each rule.

Final thoughts

Many organizations that develop software, particularly in the U.S., have not even heard of MISRA C–let alone require its use. However, this doesn't mean that as an individual developer you shouldn't consider

adopting these guidelines on your own. After all, MISRA is a tool to help you write safer, more portable code.

For those of you who can't handle the effort of formally adopting MISRA C, you should, at the very least, examine the rules. If you see rules that contradict your normal coding style, then I suggest you look long and hard at your practices. Chances are that the MISRA folks are older and wiser than you.

MISRA C does nothing for checking the validity of your algorithms. It doesn't enforce a particular style, and it cannot stop you from writing completely idiotic code. What it will do-if you let it-is protect you from most of the darker corners of the C language.