Designing Reliable Code using MISRA C

By Greg Davis
Engineering Manager, Compiler Development
Green Hills Software, Inc.

Class #346

Introduction

No software engineering process can guarantee reliable code, but following the right coding guidelines can dramatically reduce the number of errors in your code.

Many embedded systems live in a world where a system failure can be catastrophic. Their systems are so critical that if they fail, people may lose their lives. Also, unlike a PC that may be rebooted, many systems are so intertwined with their environment that they cannot be so easily restarted even if the failure was not catastrophic.

There are many factors that determine the reliability of an embedded system. A well-conceived design is crucial to the success of a project. Also, a team needs to pay attention to its development process. There are many different models of how software development ought to be done, and it is prudent to choose one that makes sense. Finally, the choice of operating system can mean the difference between a project that works well in the lab and one that works reliably for years in the real world.

Even the most well thought-out design is vulnerable to bugs when the implementation falls short of the design. This paper focuses on how one can use a set of coding guidelines, called MISRA C, to help root out bugs introduced during the coding stage.

MISRA C

MISRA stands for Motor Industry Software Reliability Association. The MISRA consortium publishes a document called Guidelines For the Use of the C Language In Vehicle Based Software that defines MISRA C. More information on MISRA can be found at their web site at http://www.misra.org.uk.

According to the MISRA C document, its purpose is not to promote the use of C in embedded systems. Rather, the guidelines accept that C is being used for an increasing number of projects. The MISRA C guidelines discuss general problems in software engineering and note that C does not have as much error checking as other languages do. Thus the guidelines hope to make C safer to use, although they do not endorse MISRA C or C over other languages.

MISRA C is a subset of the C language. In particular, it is based on the ISO/IEC 9899:1990 C standard, which is identical to the ANSI X3.159-1989 standard, often called C ’89. Thus every MISRA C program is a valid C program. The MISRA C subset is defined by 127 rules that constrain the C language. We will use the terms “MISRA” or

"MISRA C" loosely in the remainder of the document to refer to either the base document (Guidelines For the Use of the C Language In Vehicle Based Software) or the language subset.

We will start by reviewing MISRA C: what is consists of, the types of rules, and some examples of the rules themselves. After that, we will explore about ways that MISRA C can be used to improve the reliability of your system, even if you are not designing software for automobiles.

At the time of writing a second edition of MISRA C was under development, but was not available to the general public. This paper will discuss the publicly available first edition.

What is MISRA C?

MISRA C is written for vehicle-based software, and it is intended to be used within a rigorous software development process. The standard briefly discusses issues of software engineering, such as proper training, coding styles, tool selection, testing methodology, and verification procedures.

MISRA C also talks about the ways to ensure compliance with all of the rules. Some of the rules can be verified by a static checking tool or a compiler. Many of the rules are straightforward, but others may not be or may require whole-program analysis to verify. Management needs to determine whether any of his tools can automatically verify that a given rule is being followed. If not, this rule must be checked my some kind of manual code review process. Where it is necessary to deviate from the rules, project management must give some form of consent by following a documented deviation procedure. Other non-mandatory "advisory" rules do not need to be followed so strictly, but cannot just be ignored altogether.

The MISRA rules are not meant to define a precise language. In fact, most of the rules are stated informally. Furthermore, it is not always clear if a static checking tool should warn too much or too little when enforcing some of the rules. The project management must decide how far to go in cases like this. Perhaps a less strict form of checking that warns too little will be used throughout most of the development, until later when a stricter checking tool will be applied. At that point, somebody could manually determine which instances of the diagnostic are potential problems.

Most of the rules have some amount of supporting text that justifies the rules or perhaps gives an example of how the rule could be violated. Many of the rules reference a source, such as parts of the C standard that state that such behavior is undefined or unspecified.

Before exploring how one could use MISRA C, let's familiarize ourselves with the concepts and some examples of the rules of MISRA.

Taxonomy of the Rules

The MISRA C rules are classified according to the C construct that they restrict. However, most of the rules fall into a couple of groups.

The first group of rules consists of rules that intend to make the language more portable. For example, the language does not specify the exact size of the built in data types or how conversions between pointer and integer are handled. So, an example of a rule is one that says:

Rule 13 (advisory):

> *The basic types of **char**, **int**, **short**, **long**, **float**, and **double** should not be used, but specific-length equivalents should be **typedef**'d for the specific compiler, and these types names used in the code.*

This rule effectively tries to avoid portability problems caused by the implementation-defined sizes of the basic types. We will return to this rule in the next section.

Another source of portability problems are undefined behaviors. A program with an undefined behavior might behave logically, or it could abort unexpectedly. For example, using one compiler, a divide by 0 might always return 0. However, another compiler may generate code that will cause hardware to throw an exception in this case. Many of the MISRA C rules are there to forbid behaviors that produce undefined results because a program that depends on undefined behaviors behaving predictably may not run at all if recompiled with another compiler.

Unlike this first group of rules that guard against portability problems, the second group of rules intends to avoid errors due to programmer confusion. While such rules don't make the code any more portable, they can make the code a lot easier to understand and much less error prone. Here's an example:

Rule 19 (required):

> *Octal constants (other than zero) shall not be used.*

By definition, every compiler should do octal constants the same way, but as I will explain later, octal constants almost always cause confusion and are rarely useful.

A few other rules are geared toward making code safe for the embedded world. These rules are more controversial, but adherence to them can avoid problems that many programmers would rather sweep under the carpet.

Examples of the Rules

We will start by reviewing the rules mentioned above.

- *Octal constants (other than zero) shall not be used.* (Rule 19/required)

  To see why this rule is helpful, consider:

  line_a |= 256;
  line_b |= 128;
  line_c |= 064;

  The first statement sets bit 8 of the variable line_a. The second statement sets bit 7 of line_b. You might think that the third statement sets bit 6 of line_c. It doesn't. It sets bits 2, 4, and 5. The reason is that in C any numeric constant that begins with 0 is interpreted as an octal constant. Octal 64 is the same as decimal 52, or 0x34.

  Unlike hexadecimal constants that begin with 0x, octal constants look like decimal numbers. Also, since octal only has 8 digits, it never has extra digits that would give it away as non-decimal, the way that hexadecimal has a, b, c, d, e, and f.

  Once upon a time, octal constants were useful for machines with odd-word sizes. These days, they create more problems than they're worth. MISRA C prevents programmer error by forcing people to write constants in either decimal or hexadecimal.

- *The basic types of **char**, **int**, **short**, **long**, **float**, and **double** should not be used, but specific-length equivalents should be **typedef**'d for the specific compiler, and these type names used in the code.* (Rule 13/advisory)

  This is a portability requirement. Code that works correctly with one compiler or target might do something completely different on another. For example:

```
int j;
for (j = 0; j < 32; j++) {
    if (arr[j] > j*1024) {
        arr[j] = 0;
    }
}
```

  On a target where an int is a 16-bit quantity, j*1024 will overflow and become a negative number when j == 32. MISRA C suggests defining a type in a header file that is always 32-bits. For example one could define a header file called misra.h that does this. It could define an 32 bit type as follows:

```
#include <limits.h>
#if (INT_MAX == 0x7fffffff)
typedef int SI_32;
typedef unsigned int UI_32;
#elif (LONG_MAX == 0x7fffffff)
typedef long SI_32;
typedef unsigned long UI_32;
#else
#error No 32-bit type
#endif
```

Then the original code could be written as:

```
SI_32 j;
for (j = 0; j < 32; j++) {
    if (arr[j] > j*1024) {
        arr[j] = 0;
    }
}
```

Strict adherence to this rule will not eliminate all portability problems based on the sizes of various types[1], but it will eliminate most of them.

The potential drawback to such a rule is that programmers understand the concept of an "int", but badly-named types may disguise what the type represents. Consider a "generic_pointer" type. Is this a void * or some integral type that is large enough to hold the value of a pointer without losing data? Problems like this can be avoided by sticking to a common naming convention. Although there will be a slight learning curve for these names, it will pay off over time.

Another problem is that using a type like UI_16 may be less efficient than using an "int" on a 32-bit machine. While it would be unsafe to use an int in place of a UI_16 if the code depends on the value of the variable being truncated after each assignment, in many cases the code does not depend on this. In some cases, an

[1] The "integral promotion" rule states that before chars and shorts are operated on, they are cast up to an integer if an integer can represent all the values of the original type. Otherwise, they are cast up to an unsigned integer. The following code will behave differently on a target with a 16-bit integer (where it will return 0) than it will on a target with a 32-bit integer (where it will return 65536).

```
UI_32 a()
{
   UI_16 x = 65535;
   UI_16 y = 1;
   return x+y;
}
```

optimizing compiler can remove the extra truncations; in the rest, the extra cycles can be considered the price of safety.

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

  Consider the following code:

File1.c	File2.c

F_64 maxtemp;
F_64 GetMaxTemp(void)
{
  return maxtemp;
}
void SetMaxTemp(F_64 x)
{
  maxtemp = x;
}
	void IncrementMaxTemp(void)
	{
	  SetMaxTemp(GetMaxTemp() + 1);
	}

  This code may look OK, but it will not work as expected with most compilers. C has some rather dangerous rules that assume that type of a function when the function has not been declared. In File2.c, GetMaxTemp is called, but never declared A conforming ANSI/ISO C compiler will assume that GetMaxTemp() returns an int. In reality, GetMaxTemp will return a double. Depending on the architecture and compiler different things will happen, but this code will rarely work the right way.

  MISRA C avoids this problem by forcing the user to declare functions before they are used.

  C already requires variables to be declared before they are used. So, assuming one uses a conforming compiler (there is another rule for this), the rule could had been written:

  All function identifiers shall be declared before use.

  The MISRA spec explains this and other apparent redundancies:

  > *There are a few cases within this document where a rule is given which refers to a language feature which is banned or advised against elsewhere in the document. This is intentional. It may be that the user chooses to use that feature, either by raising a deviation against a required rule, or by choosing not to follow an advisory rule. In this case the second rule, constraining the user of that feature, becomes relevant.*

- *All automatic variables shall have been assigned a value before being used.* (Rule 30/required)

In C, automatic variables have an undefined value before they are written to. Unlike in Java, they are not implicitly given a value like 0. This sounds like good programming practice, so few people would disagree with this rule in most cases. But, how about the following case:

```
extern void error(void);
UI_32 foo(UI_8 arr[4])
{
    UI_32 acc, j;
    for (j = 0; j < 4; j++)
        acc = (acc << 8) + arr[j];
    if (acc == 1)
        error();
    return acc;
}
```

This function seems to pack 4 characters into an 32-bit integer, check for an error condition, and then return the int. While acc starts with an undefined value, one could argue that the undefined value is shifted out 8 bits at a time until the whole word is defined.

If this explanation makes sense to you, you might think that the function should always work. But according to the C spec, the function is still undefined. One can think of the C specification as a contract between the programmer and the compiler. If the programmer follows the rules of the specification, the compiler should produce the expected code. Any undefined behavior in the program effectively voids this contract, so that the compiler may do some unexpected things.

This example may seem academic, but it is not. It turns out that the most straightforward implementation of a modern constant propagation algorithm will "optimize" the code so that error() is never called.

- *The right hand operand of a && or || operator shall not contain side effects.* (Rule 33/required)

  A side-effect is defined as an expression that accesses a volatile object, modifies any object, writes to a file, or calls off to a function that does any of these things, possibly through its own function calls.

  The nomenclature "side-effect" may sound ominous and undesirable, but after some reflection, it becomes clear that a program cannot do much of anything useful without side-effects.

  As an example of where this rule is helpful is as follows:

```
file_handle *ptr;
success = packet_waiting(ptr) &&
          process_packet(ptr);
```

This may work fine in a lot of cases. But, even if it is safe, it can easily become a hazard later. For example, a programmer might decide to put some cleanup code in process_packet. Or process_packet might evolve so that it closes the file represented by ptr.

A safer way to write this would be:

```
file_handle *ptr;
success_1 = packet_waiting(ptr);
success_2 = process_packet(ptr);
success = success_1 && success+2;
```

This guarantees that process_packet is always called.

This rule is not a portability or safety issue, per se, because the behavior of the || and && operators are well-defined in all cases. But, the rule is intended to eliminate a common source of programming errors.

- *All non-null statements shall have a side-effect.* (Rule 53/required)

The C standard defines a null statement to be a statement that consists of just a semicolon. For example:

```
if (success)
    /* this is a NULL statement */
    ;
else
    error();
```

The effect of this rule is to eliminate statements that don't do anything. In many cases, these statements are programming errors. Here is an example:

```
status == get_packet();
```

In fact, the programmer probably meant to write:

```
status = get_packet();
```

The useful concept of a side-effect brings up an interesting point. It is generally impossible for a code checker to verify whether a function call contains side effects. The first difficulty is that the function being called may not be contained

in the current module, so a traditional checker that doesn't do intermodule analysis can not see the function to determine whether it does anything. The second difficulty is that the code checker cannot always determine at compile-time whether a function that contains statements with side effects will actually execute any of them. This problem is insurmountable, as it can be reduced to the undecidable halting problem.

Perhaps there is a more pertinent question. Is it really undesirable to ensure that all function calls have side-effects? For instance, should one really avoid calling the following function when the buffer is empty?

```
void flush_buffer (buffer *b)
{
    if (b->bytes != 0)
        flush_buffer_nonempty(b);
}
```

A restriction like this goes against the principles of data abstraction[2], but this should not be taken as an indication that MISRA C is useless to programmers. We will discuss these issues more in the next section when we consider how one might apply MISRA C to an embedded development project.

The final two rules that I will survey are perhaps the most controversial.

- *Dynamic heap memory allocation shall not be used.* (Rule 118/required)
- *Functions shall not call themselves, either directly or indirectly.* (Rule 70/required)

One problem with dynamic memory is that it needs to be used carefully in order to avoid memory leaks that could cause a system to run out of memory. Also, since implementations of malloc() may vary, heap fragmentation may not be the same between different toolchains.

Likewise, recursion needs to be used carefully or otherwise a system could easily exceed the amount of available stack space.

Applying MISRA C

MISRA C, in its entirety, is not for everyone. MISRA C was designed for the automotive market where reliability is of the utmost importance, but manufacturers in other markets,

[2] It is also possible that this is an oversight in the MISRA C specification.

such as game machines, may be able to tolerate less reliability in order to cram more features into the product.

As discussed earlier, some of the rules in the standard are advisory. One need not always follow them, although they are not supposed to just be ignored. Even the mandatory rules do not need to be observed everywhere. But, a manufacturer wishing to claim that his product is MISRA C compliant must have a list of where it was necessary to deviate from the rules, along with other documentation mentioned in the standard.

A looser approach might suffice in many cases where total compliance is not necessary. For example, let's consider dynamic memory allocation. Some projects might only use dynamic memory in rare circumstances. It might be wise for an embedded development team to look through their uses of dynamic memory to verify that their use of dynamic memory is truly safe.

Consider the following example:

```
#include <stdlib.h>
typedef unsigned int UI_32;

extern UI_32 receive_sample(void);
extern UI_32 checksum_data(UI_32
length, UI_32 *data);
void send_reply(UI_32 reply);
extern void panic(void);

/* This thread loops endlessly, receiving a
 * packet of variable length and reply
 * with the checksum of the packet.
 */
void thread(void)
{
   while (1) {

        /* Get the length of the next packet
         */
        UI_32 length = receive_sample();

        if (length != 0) {
           UI_32 count, reply;

            /* Allocate memory for the next
             * packet */
            UI_32 *data = (UI_32 *)
            malloc(sizeof(UI_32) * length);

            for (count = 0; count < length;
                  count++) {
                 data[count] =
                    receive_sample();
            }

            reply = checksum_data(length,
                      data);

            send_reply(reply);

        }
    }
}
```

There are a couple of programming errors in the example:

1. The code does not check that malloc returns a non_NULL pointer.
2. The memory allocated is never freed.

This kind of analysis might lead to other insights. For example, there is often an upper bound on the size of most inputs. If this is true, in this case, then the programmer could have just as well used malloc in a case where a static or automatic array of fixed size would have been better. Even if these sorts of transformations are not possible, it can still be instructive to look at the places where memory allocation is used. This requirement will tend to discourage unneessary uses of malloc.

Of course, a development team could use most of MISRA C, but totally disregard rules that do not seem practical for their application given the amount of development time that they have. For example, a team could follow all of the required MISRA rules, except for the rule that prohibits dynamic memory allocation. They could also decide to follow many of the useful advisory rules, such as rule 13 (which says to use length-specific types instead of the built-in types). Later on, perhaps after completing the next milestone, the team could reconsider any rules that they chose to disregard in the last pass.

In *The Embedded Muse #52*, Jack Ganssle reviewed MISRA C and offered the following major observations[3]:

- A number of rules make a lot of sense
- Some of the rules are overly restrictive and may be hard to abide by
- The rules are valuable, but they are not complete by themselves. They should be incorporated into your existing coding standards.
- If the rules were published online, they'd be more accessible. Since they're currently only available in hard-copy form, few will ever take the time to use MISRA.

These observations are reasonable; however, while not everyone may use MISRA, those who do will find it well worth their time.

Another thing that makes MISRA C particularly attractive is that a number of embedded tools vendors are already checking the rules this in their compilers and code checkers. This off the shelf support makes MISRA C easier than other alternatives that specify rules but have little infrastructure to back them up.

Conclusion

MISRA C is a valuable tool for programming teams trying to write highly reliable code. The rules are well thought out and provide many insights into likely errors and constructs that cause non-portable behavior. Almost anyone who writes C code will find MISRA's

[3] Ganssle, Jack. *The Embedded Muse 52*. http://www.ganssle.com/tem/tem52.pdf

coding guidelines useful. Consistent use of MISRA C is likely to increase software reliability.