by Anders Lindgren, IAR Systems

MISRA C—Some key rules to make embedded systems safer

No one can deny that embedded systems are becoming more and more common. However, as the number of systems grows, and as we put more and more responsibility on the systems, some important questions must be asked. The most important question is: How can we ensure that embedded systems are safe?

The C programming language provides a lot of power to its users, almost as much power as when writing in assembler. In addition, the language contains several areas that the standard explicitly states are undefined, as well as several items that the implementations are free to define.

Writing a safe program in C not only means that the program runs as the programmer expected it to. It must also run correctly when ported to a different environment. More importantly, when other people read the source code the meaning must be crystal clear. One reason is to simplify inspection. Another reason is that there is an increased risk that problems are introduced when the code is developed further.

One approach that the Motor Industry Software Reliability Association (MISRA) has taken is to define a subset of the C programming language. If an application only uses this subset then a lot of the pitfalls in the C programming language are avoided.

The subset is known as "MISRA C" and is defined in a booklet named "Guides For The Use Of The C Language In Vehicle Based Software". The guidelines consists of 127 rules that are classified as either required or advisory.

IAR Embedded Workbench, for selected products (see list below), has got built-in support for checking compliance with most of the MISRA C rules. Some rules, however, can't be checked automatically.

For applications written without the rules in mind, it could be a difficult task to conform to them. For example, MISRA C rule 118 forbids the use of dynamically allocated memory, rule 101 specifies that the use of pointer arithmetics is prohibited, and rule 102 says that no more than 2 levels of pointer indirection should be used.

On the other hand, several of the rules are very straight-forward and sound. Even if you don't have the ambition to make your application fully MISRA C compliant, supporting a few of the rules will at least make it a little bit safer.

In this article I present a number of rules that I could recommend to all projects, even if they are not intended to be fully MISRA C compliant. The rules that are selected are rules that a compiler normally doesn't check.

Using MISRA C in IAR Embedded Workbench

The MISRA C interface to IAR Embedded Workbench is straight forward. You can select which rules you would like to be checked, either by selecting the rules from a list in the "Options" menu in the

"General options" section under the "MISRA C" tab. Alternatively, if the command line version of the tools are used, with the `--misrac` and `--misrac_verbose` command line options.

Most of the rules are checked by the compiler itself. However, some of the rules requires access to the entire applications, in those cases the linker will perform the check. You can inspect the list and map files, respectively, for a list of the rules that were enabled and checked. Please note that some rules simply can't be checked automatically.

Rule 13 (advisory)

This rule states that an application should not directly use basic types such as `char, int, float` etc. Instead specific-length equivalents should be `typedef`ed for the specific compiler, and these type names should be used in the code. The reason is that different compilers could use different underlying representation for the basic types. The most common case is the type `int` that could be seen as 16 bit wide by one compiler and 32 bit wide by another.

My personal experience with specific-length types is that it is that I take greater care when selecting the appropriate size and signedness. Instead of just picking an `int` I now more carefully select, say, `uint16_t` whenever I need to represent something that never will be negative. One effect of such code is that it is easier to read and understand as the reader doesn't have to consider a situation where expressions could be negative.

The C standard, as of the 1999 version, contains a header file `stdint.h` that provides typenames on the form `int8_t` and `uint32_t`. IAR Embedded Workbench supports this header file in the DLib standard library.

Rule 23 (advisory): All declarations at file scope should be static where possible

As you probably know, by declaring a variable or a function `static` it cannot be seen or used directly by other modules in the application.

This rule:

- Prevents you from unintentionally exposing internal help functions and file-local variables
- Forces you to really design the interface of new modules
- Makes the interface more clear, something that is rely important as applications grow older and the person who wrote the code may not be around any more.

When using IAR Embedded Workbench this rule will be checked by the linker since it has got access to the entire application.

Rule 33 (required): The right hand side of a "&&" or "||" operator shall not contain side effects

There is nothing in the C language that prevents you from writing code that looks like the following:

```
if ((x == y) || (*p++ == z))
{
/* do something */
}
```

In this example, the right hand side of the || operator is only evaluated (and its side-effects executed) if the expression on the left-hand side is false—that is, if `x` and `y` are not equal. In this example, the side-effect is to increase the pointer `p`.

However, even if this behavior is specified by the standard it is still easy to get it wrong when writing the code. And even if you manage to get it right, everyone that will ever read or maintain the code must also understand the rules and your intentions.

As a side note, the code above could be rewritten into the much more straight-forward sequence instead.

```
int doSomething = 0;
if (x == y)
{
doSomething = 1;
}
else if (*p++ == z)
{
doSomething = 1;
}
if (doSomething)
{
/* do something */
}
```

Rule 59 (required): The statement forming the body of an "if", "else if", "else", "while", "do ... while", or "for" statement shall always be enclosed in braces

Basically, this says that from now you must clean up your act, you can't write sloppy things like the `else` clause in following example.

```
if (x == 0)
{
y = 10;
z = 0;
}
else
y = 20;
```

The idea of this rule is to avoid a classical mistake. In the example below the line `z = 1;` was added. It looks as though it's part of the `else` clause but it's not! In fact, it is placed after the `if` statement altogether, which means that the assignment will always take place.

If the original `else` clause would have contained the braces from the beginning this problem would never have occurred.

```
if (x == 0)
{
y = 10;
z = 0;
}
else
y = 20;
z = 1;
```

References

"Guides For The Use Of The C Language In Vehicle Based Software" -- Motor Industry Software Reliability Association, April 1998.