MISRA C in Safety-Critical Systems: How COTS Compiler Technologies Enforce Best-Practice Programming

The software running today's automobiles needs to be every bit as reliable as a defense system. Military designers can take advantage of MISRA C, a more structured version of C designed for safety-critical applications.

For Reprint Orders Call (949)226-2000 / ©2002 The RTC Group

Stephan Paternotte, Product Marketing Manager, Altium Limited

The C programming language is today's de facto standard for embedded systems high-level language programming. Unfortunately, the C language suffers from drawbacks in consistency, and its intrinsic qualities make it somewhat unsuitable for programming highly robust applications.

Through enhanced code checking and strict enforcement of best-practice programming rules, standard C compiler tools are available to help programmers produce code that conforms to the guidelines defined by the Motor Industry Software Reliability Association (MISRA). These guidelines benefit programmers of embedded products in safety-critical areas such as automotive, industrial control, communications, medical and aerospace.

However, the MISRA guidelines are also valuable to any embedded applications programmer concerned with the general quality and robustness of their work. While MISRA C compliance is not required of any COTS application, companies developing safety-related embedded military systems gain a significant edge by enforcing best-practice programming standards.

Using MISRA C Guidelines for Safety-Critical Systems

With its excellent support for high-speed, low-level I/O operations, as well as boasting compact compiler-generated code, C has generated a lot of appeal with developers of embedded applications.

Backed by a solid international standard, C is broadly supported from a wide range of code-generating (mathematical) modeling tools. Ever-increasing complexities of embedded applications drive programmers from assembly language to the higher-level languages like C. This undeniable trend is noticeable across the board of embedded systems developments—from straightforward control code, to highly sophisticated, time-critical DSP algorithms.

Unrivaled in the embedded systems market, each year automotive microcontroller-systems program source files grow in size and complexity up to a factor of ten. Considering the statistical average of 0.25% errors in any kind of software, it is fully understandable that the automotive industries are concerned with reducing the fault rate in their embedded software.

The kinds of safety-critical software issues that the automotive industry is concerned with are relevant to other industries as well. The typical components of safety-critical software include: structured software development processes, code reviews, consistent use of types and methods, boundary checking, reproducible behavior and real-time responsiveness.

Cons of C

Although there are myriad good reasons for C turning out to be the most prominent higher-level programming language—such as its support for high-speed, low-level I/O operations, and compact compiler-generated code—programming in C does not guarantee problem-free code. Of course, it's possible to write C code in a structured manner that reduces the chance of producing errors. But C can also be written in a very condensed, hard-to-comprehend manner, which dramatically increases the likelihood of introducing errors.

Additionally, a standard C compiler does not necessarily detect many small typing errors. Consider the operators &&, &, ||, |, +=, =, and ==, and think of the ease with which a small typo can still lead to perfectly valid C code.

Even if the developers are fairly knowledgeable about the language, not all C programmers are fully aware of the side effects of all possible constructs. For instance, implicit or explicit casts can easily lead to truncation errors creating confusion for fellow programmers.

The Ideal C Compiler for Safety-Critical Applications

Companies developing safety-related embedded systems gain a significant edge by enforcing best-practice programming standards. With a MISRA C–enabled compiler such as the one included in Altium's TASKING software development tools, programmers can write code that contains fewer error-prone C constructs and consequently develop more robust embedded systems for safety-critical applications.

- TASKING software development tools are the only code-checking facilities in standard C compiler tools based on the MISRA C organization's Guidelines for the Use of the C Language in Vehicle-Based Software
- Well suited for safety-related or mission-critical applications or where general quality and robustness of code are of importance
- Selectable restrictions to ISO/IEC 9899:1990 standard C programming language
- Supported on many TASKING compiler tools, including C166, ST10, TriCore, M16C, XA and 8051

For Reprint Orders Call (949)226-2000 / ©2002 The RTC Group

MISRA C section	Example guideline
Comments	Prohibits nested comments, and advises against commenting out sections of code.
Identifiers	Defines a limit of max 31-character significance, and discourages the use of identical identifiers.
Types	Describes that the basic types *char*, *int*, *short*, *long*, *float*, and *double* should be replaced with typedefs indicating the specific length (e.g., SI_16 for a 16-bit signed integer) and the type *char* shall always be declared as either *unsigned char* or *signed char*.
Constants	Prohibits the use of octal constants. By definition, ANSI/ISO C compilers interpret decimal integer constants with leading zeros as octal.
Conversions	Prohibits the use of implict-type conversions as well as redundant explicit casts.
Expressions	Describes that the value of an expression should be the same under any permissible order of evaluation, and floating-point variables are not to be tested for exact equality or inequality.
Control Flow	Prohibits the use of *goto*, *break* and *continue*. Also defines constraints on the use of the *if*, *else*, *switch*, and *case* constructs.
Functions	Defines many required rules on the declaration and use of functions.
Pointers and Arrays	Prohibits the use of non-constant pointers to functions, and completely discourages the use of pointer arithmetic.
Structures and Unions	Requires that all structure/union members are named and referred to by name only.

Table 1

Here is a small sampling of the 127 rules that are defined in the MISRA C guidelines. For a complete reference, read MISRA's Guidelines for the Use of the C Language in Vehicle-Based Software (www.misra.org.uk).

As well, some features of C behave differently from what programmers might expect. Other definitions of C features have pretty much been left open to the discretion of the compiler implementation.

One of the main reasons that C compilers do such a great job of generating compact, efficient code lies in the limited runtime checking. The C programming language does not offer provisions that prevent arithmetic exceptions—such as divide by zero, overflow, validity of addresses or pointers, or surpassing array boundaries—from causing a runtime software failure. Although this saves a lot of code space and runtime performance, it also leaves open a magnificent trap for less prudent programmers.

Fully aware of these types of defects, programmers in the safety-related embedded applications area, who have a strong interest in writing robust and consistent code, rightfully have concerns with C.

MISRA C Concept

For all the good reasons, including its broad availability across microprocessor/DSP architectures, ANSI/ISO C is well accepted as the standard for higher-level language programming of embedded systems. But as discussed, C is also considered not particularly suitable for programming safety-related applications.

For many years, companies developing safety-critical embedded applications have found ways to overcome the drawbacks of C—primarily by means of written guidelines with the intent of reducing the probability of coding errors by restricting the use of error-prone C constructs.

Based on experience and expertise in automotive applications, as well as information from a number of public-domain sources, in 1998 MISRA defined a total of 127 programming rules that are applicable when developing safety-related applications in C. In *Guidelines for the Use of the C Language in Vehicle-Based Software*, also known as the MISRA C guidelines, MISRA defines a core list of required rules and a supplementary set of advisory rules.

Although the MISRA C guidelines outline a wide range of programming techniques, they can also be interpreted as a subset of the ISO C programming language that is intended to be more suitable for the development of safety-critical applications. Table 1 highlights some of the "dos and don'ts" that are reflected in the MISRA C guidelines. For the complete list of 127 rules, read MISRA's *Guidelines for the Use of the C Language in Vehicle-Based Software* (www.misra.org.uk).

The MISRA C guidelines are organized under a broad range of topics in the C language. For example, in the section "Constants," rule 18 advises that numeric

```
I = ((I==2) && (J<<=1))
```

```
if ((x = y)  != 0)
```

Figure 1

The code examples here show intentional assignment within a Boolean expression. Or was it a typo? MISRA C rule 35 prohibits the use of assignment operators within Boolean expressions and again emphasizes that one should not unnecessarily combine actions into one line of code.

For Reprint Orders Call (949)226-2000 / ©2002 The RTC Group

```
MY_UCHAR uc;
MY_SHORT si;
...
uc = si
```

Figure 2

Because MISRA C rule 43 says not to use implicit conversions (e.g., multiplying a char by an int), the compiler is able to alert the programmer to this potentially dangerous problem of incompatible data types. The compiler forces the programmer to make explicit casts, telling the compiler what type it should assume when, for example, multiplying char by int.

constants should be appropriately suffixed to indicate type. To avoid mixing data types and related forthcoming implicit casts in an expression, lines like:

```
gauge = gauge + 3;
```

should be written as:

```
gauge = gauge + 3u;
```

in which case all elements in the expression are of the *unsigned int* type. In combination with "Conversions" rule 43, which prohibits implicit conversions with possible information loss, rule 18 enforces consistent use of data types throughout the whole application.

Another, not too obvious, error in the "Constants" realm is caught by rule 19, which states, "Octal constants (other than zero) shall not be used." Not many programmers realize that when they assign a constant with a value with leading zero, it is interpreted by the compiler as octal. So, initializing an array with values that are—for the sake of alignment—fixed-length values, is, to say the least, tricky.

```
CalTable[1] = 109;  /* set to decimal 109 */
CalTable[2] = 100;  /* set to decimal 100 */
CalTable[3] = 071;  /* set to decimal 57 */
CalTable[4] = 052;  /* set to decimal 42 */
```

Except for the special-case zero, octal constants are better not used at all, in which case the compiler tools can easily offer a helping hand in the enforcement.

The "Operators" section in the MISRA C guidelines covers a wide range of rules related to C operators such as &, &&, ||, =, and so forth and their role in

```
if (EF == 0)
```

Figure 3

Given that floating-point variables are always greater or less than value you compare it to, testing them for exact equality or inequality, as shown in this code, will always returns a value of False. But because MISRA C rule 50 states that floating-point variables shall not be tested for exact equality or inequality, the compiler was able to catch the error and stop compilation until the programmer could fix the problem.

the expression evaluation. The background of these rules lies in the fact that many possible typing errors, whose side effects may or may not affect the application's runtime behavior, generally lead to perfectly valid C code.

For example, rule 35 prohibits the use of assignment operators within Boolean expressions and again emphasizes that one should not unnecessarily combine actions into one line of code.

```
i = ((i==2) && (j=1));
```

The code example here shows an intentional assignment within a Boolean expression. Or was it a typo? The fact remains that the possibly intended assignment could easily have been drawn outside the Boolean expression. This would have made the code less confusing, easier to understand, and better to maintain. By disallowing these kinds of combined expressions the compiler helps prevent hard-to-find typos that have possibly grave consequences for the runtime behavior of the safety-critical application.

Considering the broad range of safety-related applications, it is not hard to imagine what could go wrong with the side effects of an inadvertent assignment in a Boolean expression, an implicit type cast, commented section of code, typedef reuse, uninitiated automatics, stray function pointers, and so forth. Disastrous results caused by simple code errors might include: an infant ventilator that switches to increased frequency or Positive End Expiratory Pressure (PEEP), an airbag that fails to fire, an automotive ABS that fails to activate in an emergency, or a missile that inadvertently targets a friendly aircraft.

Typically, conformance to the MISRA C guidelines can be checked by manually reviewing the code, by using external static code analysis tools, or by invoking integrated restrictions in the embedded software development tools. Clearly, this last option is the most efficient method for time-crunched programmers.

MISRA C Code Checking in Compiler Technologies

To meet the needs of the automotive industry, as well as other safety-critical industries such as aerospace, industrial control, communications and medical, it's extremely useful for C compiler tools

For Reprint Orders Call (949)226-2000 / ©2002 The RTC Group

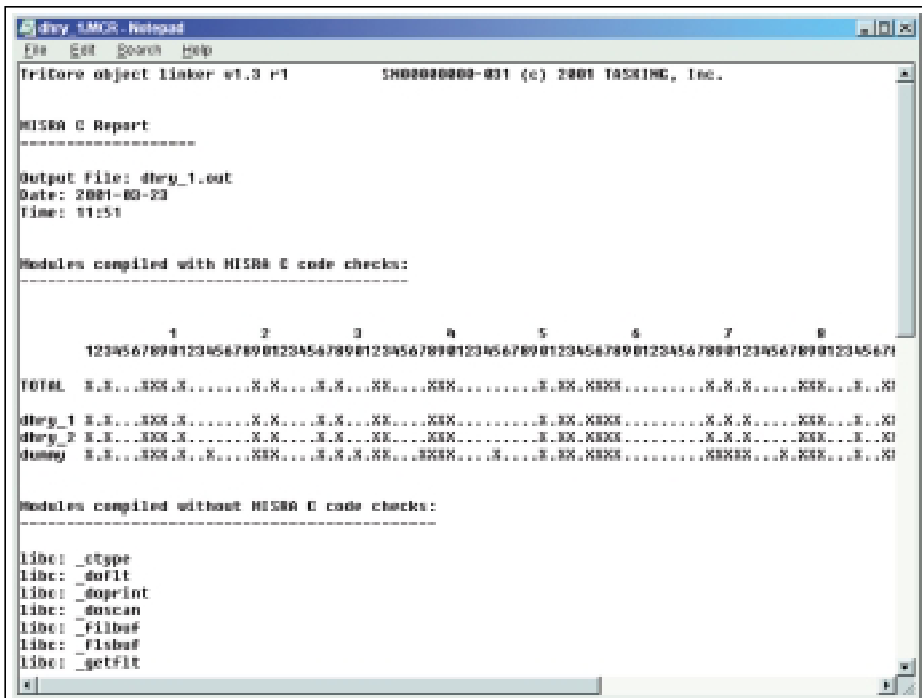

Figure 4

MISRA C provides automatic documentation, proving what checks have been done and which rules are being applied to which modules—an especially useful feature for safety-critical applications. In this screenshot, "X" marks which rules are being checked on that module. This easily configurable feature provides developers with major flexibility, enabling them to choose which rules are and aren't used and to tune the compiler to the specs of the application.

to include enhanced code-checking facilities that conform to the MISRA C guidelines. The ideal compiler would thus be part of an embedded software development tool suite with integrated facilities that enforce best-practice programming.

In general, embedded software development tools consist of a C/C++ compiler, macro-assembler, linker/locator tools, a professional editor and a debugger. A typical integrated development environment (IDE) should integrate all these tools and offer an easy-to-learn, easy-to-use interface for all of the necessary tools settings and configuration of the target.

To develop safety-critical applications, the C compiler should be integrated into the IDE and include front-end technology that has been enhanced with additional code checks that conform to the MISRA C rules. Thus, during compilation of the code, violations of the enabled MISRA C rules can be indicated with error messages, like those shown in Figures 1–3, and the build process can be halted. Such error messages clearly identify where and how the programmer went wrong and guide the user toward compliance with best-practice programming standards.

Given that not all users or applications are ready for full-blown MISRA C code checking, the compiler technology needs to be able to activate the rules that are applicable to a project one by one, allowing the developer to more clearly trace the code errors. And, of course, for applications that must comply with MISRA C guidelines, the compiler should also include a predefined configuration for conformance with MISRA C's "required" rules. MISRA C basically classifies rules as "required" or "advisory." For the purpose of compliance with the MISRA C guidelines, the application of the 93 "required" rules is regarded mandatory.

Alternatively, it is possible to read C settings from an external configuration file. This working mode is particularly useful in situations where the development team needs to adhere to a specific set of MISRA C rules. This approach enables a company's management to define specific templates that conform to the company's quality-assurance program, greatly reducing the amount of documentation and testing that is required for each and every application.

In addition, by adopting a MISRA C integrated development environment, developers ensure compliance with the MISRA C rules throughout the entire project since the ideal linker/locator technologies will generate MISRA C reports, like the one in Figure 4, that document the different modules in the project with the respective MISRA C settings at the time of compilation. Filed in the company's quality-assurance system, this report can provide proof that company rules for best-practice programming have been applied in the particular project.

COTS Strategic Edge

Companies developing safety-related embedded systems gain a significant edge by enforcing best-practice programming standards with MISRA C. With a MISRA C-enabled compiler (see sidebar: "The Ideal C Compiler For Safety-Critical Applications", page 22) programmers can write code that contains fewer error-prone C constructs and consequently develop more robust embedded systems—a benefit that extends beyond automotive into military, aerospace, medical, communications, industrial automation and computing industries.

Altium (formerly TASKING)
San Diego, CA.
(858) 521-4281.
[www.tasking.com].

MISRA
44 24 7635 5430.
[www.misra.org.uk].

Safety-related systems web sites:

- The World Wide Web Virtual Library: Safety-Critical Systems
- SCCS Index
- CASS, Conformity Assessment of Safety (related) Systems
- Advances in Safety Critical Systems
- International Organization for Standardization (ISO), see ISO/TR 15497.

For Reprint Orders Call (949)226-2000 / ©2002 The RTC Group