GRAMMATECH

HOW TO AVOID COMMON PITFALLS IN MISRA COMPLIANCE

TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

BACKGROUND

In embedded development, C remains an extremely popular choice of language. Although other languages, such as Ada, C++, and Java are used in some circumstances, and model-driven development is becoming more popular in specific domains, about 50% of the code running on embedded systems is still hand-written C.

C is a great language in many respects. It is easy to use for interfacing with hardware devices. It is readily available for almost every processor. It is flexible enough to allow an author to write very tight and efficient code.

Unfortunately, C is also an extremely hazardous language. Its very flexibility means that it is easy for a programmer to make mistakes. Because the standard of what constitutes a valid C program is very liberal, compilers are bad at detecting many different kinds of errors. Further, it is riddled with ambiguities, so code that works perfectly well with one compiler may fail when a different compiler is used because each compiler has a different valid interpretation of the standard.

These challenges make C programs quite susceptible to serious memory-access defects such as buffer overruns, null pointer exceptions, and many others. Other classes of errors, such as resource leaks, use of uninitialized memory, and use-after-free errors are also endemic and abundant in C programs. And when concurrency is used, defects such as data races and deadlocks are easy to introduce but difficult to detect in development.

WHY ARE ORGANIZATIONS MOVING TO MISRA C:2012 FOR EMBEDDED APPLICATIONS?

In the face of these challenges, developers must be extremely careful about using C. Thorough testing, of course, is paramount, and the use of advanced static
analysis tools such as CodeSonar has taken off in recent years because they have proven to be effective at finding some of these defects.

An additional way of reducing the risk of C is to restrict the use of the language by prohibiting some of the more unsafe practices used in programming with it. There are several coding standards in popular use that attempt to do this, but the most mature and widely accepted of these is MISRA C.

MISRA C is a standard developed by the Motor Industry Software Reliability Association, and aims to foster safety, reliability, and portability of programs written in ISO C for embedded systems. Since the introduction of the first edition in 1998 and a subsequent revision in 2004, its use has grown steadily and is now used widely in domains beyond automotive including aerospace, medical devices, industrial control, and others.

The latest edition is MISRA C:2012, which extended coverage to C99 (while maintaining applicability to C90) and eliminated many of the inconsistencies of previous editions. It is widely considered to be a great improvement over the previous standard. Because of this, it is very hard to recommend that organizations continue to use any of the previous versions.

Three Critical Factors
Every Development Team Should Understand About MISRA C:2012

1. Use the latest standard.

This latest edition brings substantial improvements from the previous standard, and organizations that have not yet switched to this version are exposing their products and customers to considerable risk.

2. Pay special attention to Rule 1.3 and Directive 4.1.

Rule 1.3 *There shall be no occurrence of undefined or critical unspecified behavior.*

Directive 4.1 *Run-time failures shall be minimized.*

Many of the most serious bugs in C arise from undefined behavior. These bugs include:

- Buffer overruns and underruns
- Invalid pointer direction
- Double close
- Data races
- Division by zero
- Use of uninitialized memory

3. Adopt an automated static analysis tool.

MISRA C recommends the use of an automated static analysis tool to find violations of the standard.

- Avoid lightweight tools, which can find superficial syntactic violations but are generally not capable of finding the deeper defects. A clean report from a lightweight tool can give you a false sense of security while missing serious defects.
- Look for a tool that can find both violations of the syntactic rules as well as bugs such as the ones listed above, which have deep semantic knowledge of the entire program.

All subsequent references to rule numbers in this paper are with respect to the MISRA C:2012 standard.

The guidelines are separated into 143 rules that are intended to be statically checkable, and 16 "directives" that address development policy and process. The rules are mostly prohibitions on using certain code constructs or practices, and range from the superficial to the deep.

MISRA C:2012 AND ADVANCED STATIC ANALYSIS: THE EMBEDDED GAME-CHANGER

One of the most important aspects of MISRA C is its support of automated static analysis tools to find violations of the standard. Because tool support is so important, it is helpful to understand the kinds of properties that static analysis tools can detect. Some tools can only reason about superficial syntactic properties of the code, whereas the more advanced tools have deep semantic knowledge of the entire program.

To understand why this distinction is important, it is useful to look at some rules. For example, rule 15.1 states, "The *goto* statement should not be used." The presence of a goto statement is clearly a simple syntactic property of the code, and as such, violations are easily found. A tool only has to be able to parse the syntax of the compilation unit to be able to find goto statements.

Rule 5.2 states, "Identifiers declared in the same *scope* and name space shall be distinct." This is also a fairly simple syntactic rule, but a tool that detects a violation must have a symbol table of the compilation unit so that it can reason about the identifiers and their scopes. It is fair to say that this rule is a little harder to enforce than the one that forbids gotos, but not by much.

On the other hand, there are some rules that require a very sophisticated analysis if violations are to be found. MISRA C:2012 labels each rule with an assessment of how amenable the rule is to static analysis. Rules are labeled *single translation unit* if a tool can find the violation by looking at only the compilation unit, or labeled *system* if the analyzer must look at all compilation units that contribute to the program in order to flag a violation.

Rules that are marked *single translation unit* are fairly easy to enforce, and in fact many compilers now have a mode where they can report such violations as warnings. An analyzer capable of finding violations of rules labeled *system* is said to be whole program.

A more important aspect of the rule is its decidability. A rule that is labeled *decidable* means that it is possible for a static analysis tool to find all such violations with no false positives; most of the superficial syntactical rules are marked as such.

In contrast, a rule that is labeled *undecidable* means that it is in general provably impossible for a static analysis tool to find all violations without any false positives. This is not to say that static analysis is not recommended for such rules — it just means that tools may fail to find some violations and may also report some false positives.

One such example is rule 2.2, "There shall be no *dead code*." Dead code is defined as any operation whose result does not affect the behavior of the program. It is easy to see how this is a hard property to detect — an analysis tool must be able to understand the semantics of all possible executions of the program and to be able to tell what portions of that code have no effect. Although there may be some instances that are easily detectable, finding all instances with no false positives is infeasible.

Although static analysis tools cannot detect all violations of undecidable rules, it is critically important that tools be used to detect as many violations as possible because that is where the most critical bugs are likely to hide. There are two clauses in the standard that are particularly relevant here — one rule and one directive.

Rule 1.3 "There shall be no occurrence of undefined or critical unspecified behavior."

Directive 4.1 "Run-time failures shall be minimized."

These are arguably the two most important clauses in the entire standard. Between them, they target the Achilles heel of C programs. Undefined behavior is explicitly discussed in the ISO standard for C (Annex J in the C99 document), and covers a broad range of aspects of the language. It often comes as a surprise to C programmers to learn that according to the standard, if a C program invokes undefined behavior, it is perfectly legal for that program to do anything at all. This is sometime facetiously referred to as the "catch fire" semantics, because it gives the compiler liberty to set your computer on fire.

Of course, since most compiler writers are not pyromaniacs, compilers usually try to do the most sensible thing in the face of undefined behavior. If the undefined behavior is detectable by the compiler, then the sensible thing is to have the compiler emit a compilation error. However, if the undefined behavior is not detectable by the compiler, then a compiler writer has essentially no choice but to assume it cannot happen.

Undefined behavior is not a rarely-encountered niche; the C99 standard lists 191 different varieties, and it turns out that even some apparently benign things are classified as undefined behavior. Consequently, it can be hard for even the most careful programmer to avoid undefined behavior.

Unspecified behavior is less hazardous, but has its own pitfalls. In this case, the standard specifies a set of legal behaviors, but leaves it to the compiler writer to choose which to use. This gives the compiler writer latitude to choose the interpretation that has the best performance, but it means that code can have different semantics when compiled by different compilers, or even when the same compiler is used in a different way.

Undefined behavior is almost always something that a programmer should be concerned about. Many of the most serious bugs are those that arise because of undefined behavior, for example, buffer overruns and underruns, invalid pointer indirection, use after free, double close, data races, division by zero, and use of uninitialized memory.

None of these defects are specifically singled out as forbidden in the MISRA standard, but are instead covered under the umbrella of Rule 1.3 and Directive 4.1. Nonetheless, every such bug is a violation of the standard.

STATIC ANALYSIS TOOLS FOR MISRA COMPLIANCE: ALL TOOLS ARE NOT CREATED EQUAL

Although lightweight static analysis tools can detect some of the more obvious instances of these bugs, only the most advanced static-analysis tools are capable of finding the more subtle occurrences. When choosing a static analysis tool to enforce MISRA C compliance, the best choice is a tool that can find violations of the superficial syntactic rules as well as bugs such as the above.

To understand why advanced static analysis tools are capable of finding these bugs, it is useful to explain a little about how they work.

All static analysis tools work by creating a model of the program and then performing queries on that model to find anomalies. An advanced static analysis tool creates a model by parsing the code and then creating a set of representations that capture the important aspects of the semantics of the program.

These representations are very similar to those used by compilers, and include abstract syntax trees (ASTs), symbol tables, control-flow graphs (CFGs), type hierarchies, and the call graph. While superficial properties of the code can be computed by doing lightweight pattern matching on the AST or the CFG, finding deeper semantic bugs requires sophisticated algorithms that mimic a real execution of the program, but which instead of maintaining concrete values for variables, maintain a set of equations that model the abstract state of the program.

An advanced static analysis tool capable of finding serious defects is more than capable of finding violations of the syntactic rules as well. However it is important to note that the converse is not true — most of the lightweight static analysis tools available that can find the syntactic violations are generally not capable of finding the deeper defects. A clean bill of health from a lightweight tool can give a false sense of security when the serious defects are missed.

A static analysis tool, of course, is useless if nobody uses it, so a successful deployment is one that fits into the development process and that makes it easy for teams of engineers to analyze their code and collaborate on correcting any detected defects.

The most appropriate architecture is a client-server model. This allows engineers to spin up analyses on their own workstations, and have the results sent to a centralized permanent repository where they can be triaged, marked up, and assigned to engineers for remediation. The same architecture makes it easy to set up regularly-scheduled analyses or analyses automatically triggered by change commits.

For managers, the client-server model makes it possible to create reports. These can be used to monitor progress and quality, but more importantly, to demonstrate compliance with applicable standards.

WHAT SHOULD MY STATIC ANALYSIS TOOL PROVIDE?

The model is precise.

The tool can parse code exactly the same way the compiler parses it. All compilers are different, and analysis tools that don't take this into account can provide false results.

It does a whole-program analysis.

The tool can track how information flows between procedures and across compilation unit boundaries.

The analysis is flow, context, and path-sensitive.

The tool can be precise about finding and reporting defects.

Infeasible path elimination.

The tool uses this to cut down on the number of false-positive results reported. The best tools use advanced techniques such as SMT solvers.

MISRA C:2012 native checkers.

The tool uses native MISRA C:2012 checkers to assure compliance to the standard. Use of partnerships or compliance only to previous versions of the standards will not provide adequate performance.

CONCLUSION:
EFFECTIVE MISRA COMPLIANCE MEANS DETECTING BOTH SIMPLE & COMPLEX VIOLATIONS

Modern embedded software development organizations must be equipped to identify not only the violations of superficial syntactic rules, but also serious bugs arising from undefined behavior, as proscribed by the MISRA C:2012 standard.

Although lightweight static analysis tools can detect some of the more obvious instances of both, only the most advanced static-analysis tools are capable of finding the more subtle occurrences.

Our experience deploying static analysis at hundreds of organizations involved in all kinds of software development has demonstrated that there are additional properties of a tool that are critical if it is to be used effectively.

First, it must integrate with the build system.

Second, the analysis must be fast enough to not get in the way. This is important so that programmers get feedback on their changes quickly.

Third, the algorithms must be *incremental* so that when small changes are made to the code, the analysis does not have to recompute everything from scratch.

And finally, the algorithms must be *scalable* to large programs because it's not uncommon for embedded systems to consist of millions of lines of code.