Electronic Design

RAFAEL TAUBINGER, Technical Marketing Specialist,
IAR Systems

Take Control of Your RISC-V Codebase

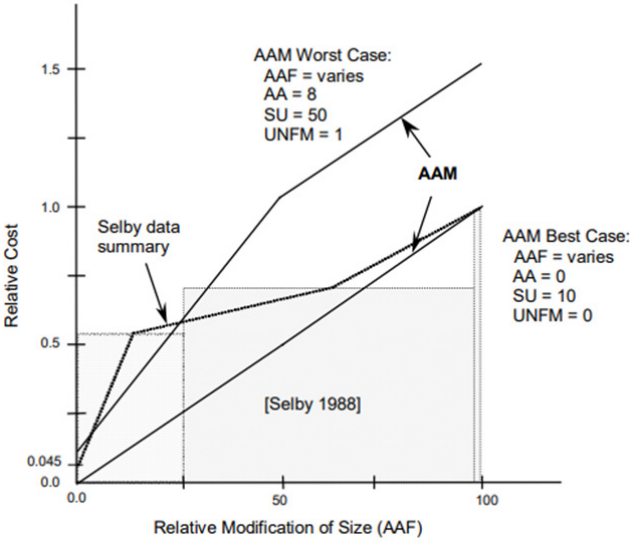
Delivering more complex software at an ever-increasing pace raises the risks of software errors, which can affect product quality as well as cause security issues. This becomes even more of a reality with the relatively new RISC-V codebase.

When we talk about take control of your RISC-V codebase, there are really two aspects to it. The first meaning is reusing your codebase for future projects. The second aspect is that poor code quality is actually a widespread problem—there's quite a bit of evidence to support the claim that bad coding practices lead directly to vulnerabilities.

Clearly, then, every developer and company must improve code quality so that the software stands the test of time. In other words, it needs to be defect-free, or as close to defect-free as possible.

Reusing Code

The Boehm's COCOMO[1] method estimates how the relative cost of writing the code is dramatically impacted by how much you modify the reused software *(Fig. 1)*. The x-axis is what percentage of modification you do to the code you intend to reuse, while the y-axis represents the percentage of what it would be if you wrote fresh code.



1. The Boehm's COCOMO nonlinear reuse effects method estimates how the relative cost of writing the code is dramatically affected by the amount of modification done to the reused software.

Note that for two of the three data samples of code, you didn't have to modify much of the supposedly reused code to suddenly jump to 50% of the effort of rewriting the code from scratch. The key point here is that if you really want to reuse code, it must be of very high quality and well designed to be cost-effective.

Focus on Code Quality

There are several reasons why code quality is a big issue. First, depending on the maturity of your development organization, you can spend up to 80% of your time in debugging.

If you could quickly isolate defects before they make it into a formal build, you'd have a lower defect injection rate, which means you can meet your organization's quality metrics much more quickly. But it also means that your code has fewer remaining bugs overall, making it a good candidate for reuse since using the code again has a lower chance of uncovering a previously undetected bug.

In addition, high-quality code is easier to maintain because of fewer defects and—if it follows good software engineering principles—it will be easier to extend. Therefore, reusing it really does give you faster follow-on projects. It's also easier to get safety certifications if your application requires it. Consequently, higher code quality means less "technical debt" to reusing it.

Available Coding Standards

Many coding standards are available, but only a few are widely used. MISRA C[2] is a software-development standard for the C programming language put together by the Motor Industry Software Reliability Association. Its aims are to

facilitate code safety, portability, and reliability in the context of embedded systems, specifically those systems programmed in ISO C.

The first edition of the MISRA C standard, "Guidelines for the use of the C language in vehicle-based software", was produced in 1998, and is officially known as MISRA C:1998. There was an update in 2004 and again in 2012 to add more rules. There's also a MISRA C++ 2008 standard based on C++ 2003.

Some good coding standard rules also can be found from the CWE - Common Weakness Enumeration[3] from MITRE. The list was started when the folks at mitre.org did a survey to find out what kinds of defects developers accidentally inject into their code. Surprisingly, developers of all stripes—web, app, desktop, or embedded—tend to make the same kinds of mistakes. Thus, was born the CWE, which is a list of these common pitfalls that developers should avoid.

For example, there are allocations without deallocations in C++ code (or even in C code). Another involves functions used without prototyping, which is an interesting point on good coding practices. If you don't prototype your function, you don't get rigorous type-checking at compile time.

However, you also can have less efficient code because the rules of the C language state that without a prototype, all arguments are promoted to integers. This can invoke casting and floating-point operations if your MCU doesn't have an FPU. That's why you should always prototype. But the main point to the CWE is that it identifies risky and bad coding behavior.

SEI CERT C and C++[4] also define common vulnerabilities that come from case studies, like checking floats for out-of-bounds conditions and making sure that you don't override a const. It also prescribes styling conventions to make code more readable and understandable.

Practical Examples of MISRA C 2012

MISRA C 2012 is widely used for securing code quality in embedded applications. Let's explore some rules and directives to better understand how the coding standards affect the source code.

In Directive 4.6, for example, you're not allowed to use a primitive data type. At first, this may seem like an odd thing to do, but when you understand the reason, it makes a lot of sense. Different compilers treat things like int differently, both in terms of its size and its signedness. This can make it tricky to review code as well. If you're a reviewer, it also makes you wonder if the original author of the code understood how the compiler interprets that code. If you don't use primitive types, you make the code invariant across compilers and architectures.

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

2. MISRA C 2012 rule13 states that the right-hand side of an AND or OR operator cannot contain side-effects.

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

3. MISRA C 2012 rule 14 states that the body of an if or while statement must be enclosed in curly braces.

Most of the time, developers will be using something like uint16_t, which tells the compiler the variable is an unsigned 16-bit quantity because the width and signedness are explicitly stated in the variable type. These are part of stdint.h.

Another interesting directive is rule 13. It says that the right-hand side of an AND or OR operator cannot contain side-effects. The code snippet in *Figure 2* might look fully correct, but it isn't.

The issue is that the right-hand side only gets executed if the expression on the left-side is false. Only then will the pointer p be post-incremented. The problem is that it's easy to get this behavior wrong when writing code. Moreover, everyone who ever reviews, tests, or maintains the code must understand the ramifications of how you wrote your code. It's clear that comments in this section of code could help, but the reality is that it's seldom well-documented.

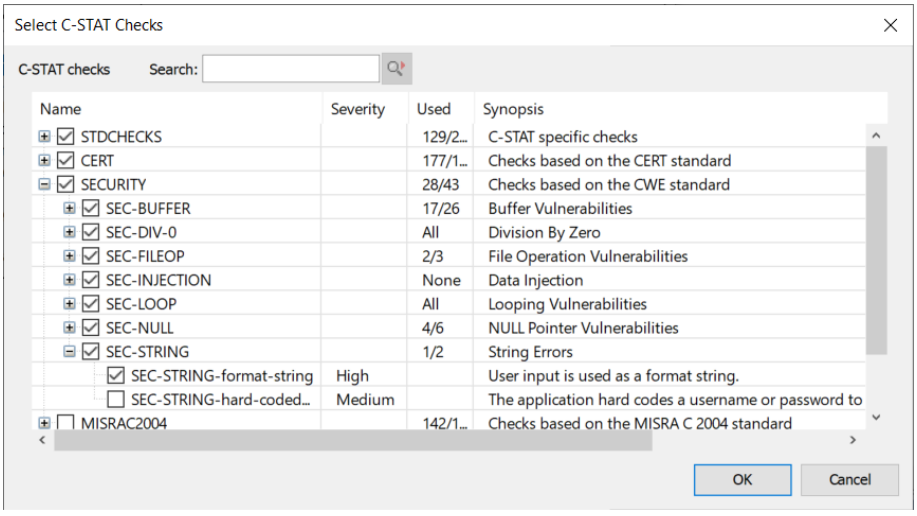
Rule 14 states that the body of an if or while statement must be enclosed in curly braces *(Fig. 3)*.

It's difficult to tell if the z=1 statement is intended to be part of the else block. This happens because it's indented at the same level as the previous statement. If it's intended to be that way, this is a bug because it clearly doesn't go in the code block the way it's written. This rule helps prevent this type of coding error. It's just a small sample of the 200+ rules available in MISRA C to make your code more reliable and portable, thus future-proofing your design.

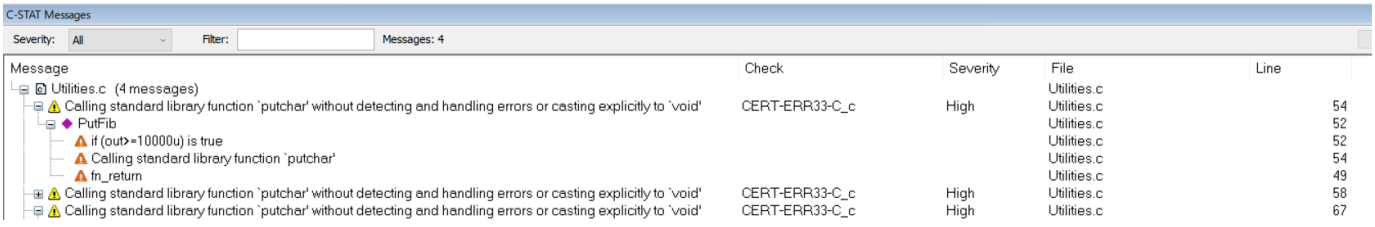
Fast Ways to Better Code

The quickest way to improve code quality is to use code-analysis tools. In fact, if you're doing a functional-safety certified application, you're required to use static analysis.

These types of tools help you find the most common sources of defects in your code. However, they also help you find problems that developers tend to not think or worry about when they're trying to write their code, especially when they're just putting up scaffold code to just



4. Standards and rule selection.



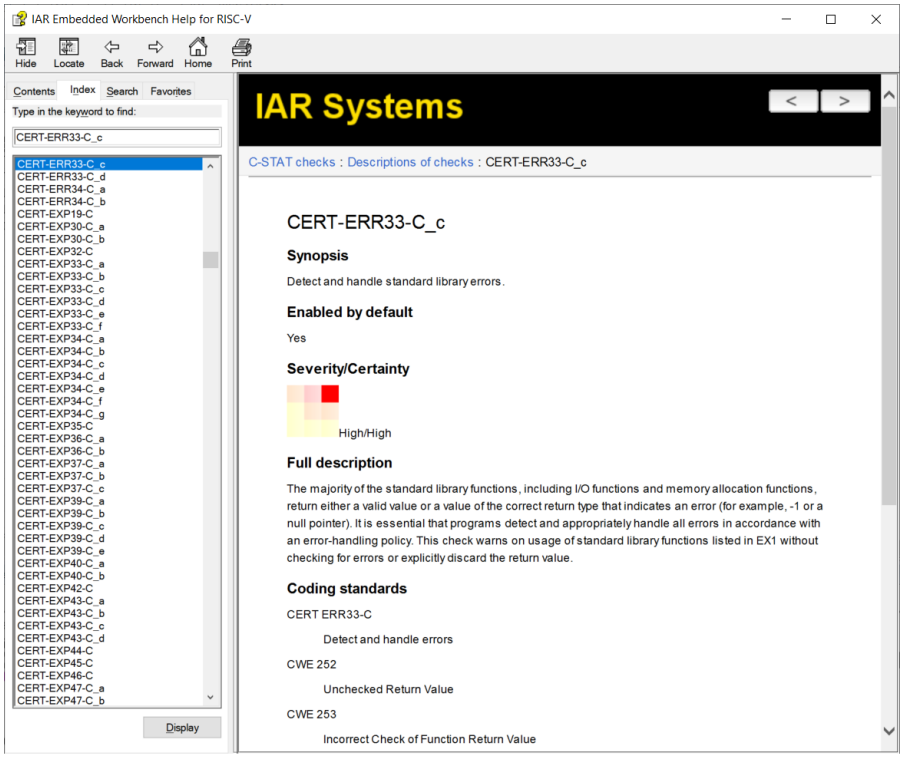
5. Code-analysis messages.

get something working. These types of tools really help you develop better code because they enforce coding standards.

Depending on the quality of your static-analysis solution, they can check for many other potential issues while you're still desk-checking your code. To see how it works, let's check the C-STAT static-analysis tool (which is built into IAR Embedded Workbench for RISC-V) in action.

C-STAT sources its rules from MISRA C 2004 and 2012 rulesets, MISRA C++ 2008 ruleset, the Common Weakness Enumeration (CWE) from MITRE, and from SEI CERT C. *Figure 4* shows the available rules that can be enabled or used to enforce compliance with the coding standards.

It's possible to drill down into categories and select only the rules that we feel are applicable to our project. In addition, it's possible to override these selections at the group, file, function, or even individual line level to give a



6. Context-sensitive help.

complete granularity over what's being checked.

Once the tools are configured, the project (or group or individual source file) can be analyzed (or files). After the analysis completion, it's possible to drill into each file to verify the issue that has been triggered:

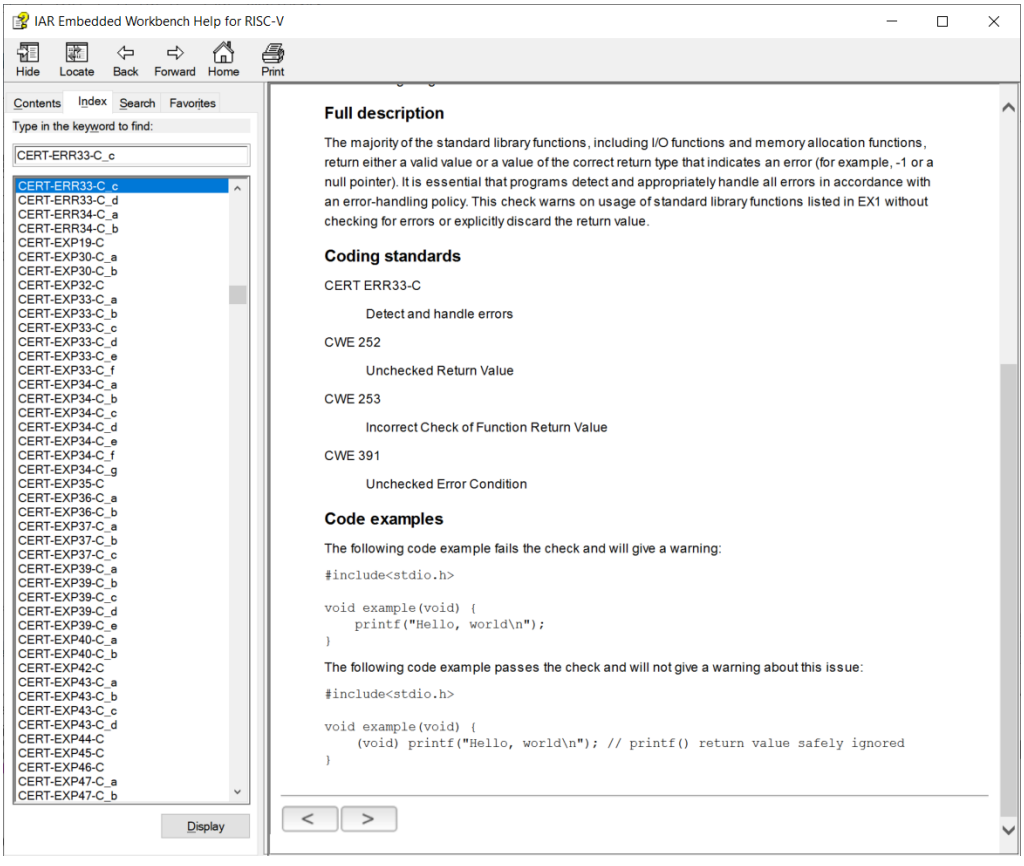
The detected issue CERT-ERR33-C_c in *Figure 5* is part of the CERT C rules. Most of the rules are self-explanatory, but more information can be found in the user guides or in the context-sensitive help (F1) like that displayed in *Figure 6*.

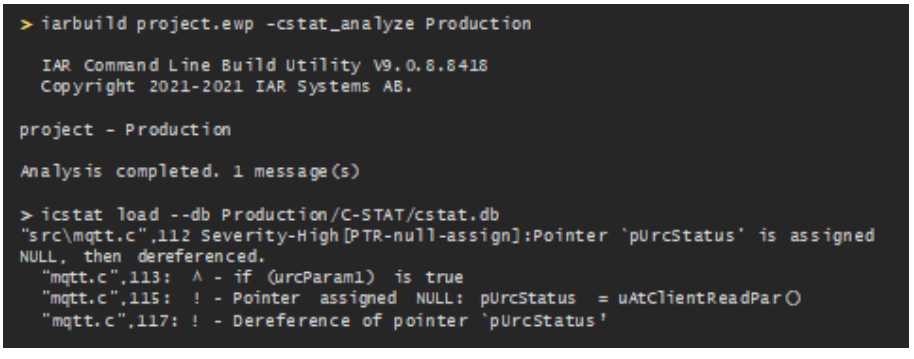
From the help window, it's possible to get a full description of the issue, how certain this is a bug versus the severity of what happens if that bug manifests itself, and all of the coding standards it violates. Most importantly—down at the bottom, like shown in *Figure 7*—it's possible to see 1-3 code examples that show a bad example and how to correct it so that it will pass the check and make the code more robust. This helps to quickly eradicate the defects that static analysis uncovered in the source code.

Automated Workflows

Ensuring code quality is important for developers working day-by-day at the desk. However, even more important is the code quality in modern and scalable build server topologies for CI/CD pipelines including virtual machines, containers (docker), and runners.

Code-analysis tools should scale well so that the automated





8. Automated workflows.

task of ensuring compliance with the programming standards can easily be achieved for bigger teams and teams spread out in different locations around the globe. *Figure 8* shows the use of the C-STAT static-analysis tool used from the command line in Linux - Ubuntu. For many automated workflows, the cross-platform support is a standard to improve efficiency for development teams.

Get Help from Code Analysis

One of the major theoretical benefits of static analysis is that it doesn't impact the performance of a system since it's not even running the system while performing the analysis. It's also independent of the quality of test suites. After all, finding a specific error in running code depends on executing a specific path through the program with a specific dataset. However, a static-analysis tool can, in theory, examine all possible paths through the code.

By introducing code-quality control early in the development cycle or while reusing code and future-proofing the source code, the impact of errors can be minimized. Providing static analysis at the fingertips of developers working with RISC-V devices with well-defined coding standards can help them find issues in the source code during development, where the cost of errors is smaller than in the released product.

References

1. https://en.wikipedia.org/wiki/COCOMO
2. https://www.misra.org.uk/misra-c/
3. https://cwe.mitre.org/
4. https://wiki.sei.cmu.edu/confluence/display/seccode/