Electronic Design

WILLIAM WONG | Technology Editor

Improving Code Quality in the New Year

Are you going to reduce bugs and improve security and code quality in 2018?

I don't intentionally put bugs into my code. Do you? Probably not, but having coded for decades I know that those nasty critters sneak in whether we like it or not.

I spent a good deal of time coding in assembler. It was easy to make errors because a programmer needed to do most of the checking. Moving to a high-level language improved the development process, but C traded off some common bugs for others—like using a single equal sign (=) instead of a double equal sign (==), causing an assignment when I intended to do a comparison in an IF statement. Sometimes the code would work because the value being assigned matched the result of what the comparison should have resulted in.

Programmers often dismiss these kinds of bugs as trivial, which they are, but they are ones that can result in problems that are hard to find and fix—ones that cause major headaches when the code is in the field. It is well known that the cost of fixing a bug grows exponentially the longer it is around. Fixing something that has been in field for any length of time is significantly costlier that finding it the when it is introduced.

There are many tools and methodologies that can improve the development process, like static analysis tools or code reviews. The difference between the two is that the former uses a computer while the latter involves human beings, although machine learning is starting to tackle code reviews. Still, while many will probably not be offended by having another person review their code, they may scoff at having the computer do it. Unfortunately, the computer is a lot better at tracking down those nitpicking bugs like in the above assignment/comparison example. Our Electronic Design Embedded Survey indicated that many (but not most) are using static analysis tools.

Now it is true that many C/C++ compilers can check and warn about such problems, but only if the developers take advantage of this and enable those options. Likewise, most static analysis programs are much more sophisticated. They can find more complicated problems, from identifying dead code to detecting semantic errors the compiler cannot.

Languages have been devised that address some common

errors like memory management. Java is one example of a language that has garbage collection support as part of its specification. One can argue about the usefulness of this feature in embedded applications, but it is only one way to approach the problem.

Another language that looks to address memory management is Rust, which is new but growing in popularity. It uses the manually managed storage approach common in C and C++ while placing restrictions on the use of pointers and allocation procedures. Unfortunately, there are no commercial Rust compilers at this point, which makes its use moot for many embedded applications.

SPARK, a provable subset of Ada, takes the idea of static analysis to the extreme. The problem with applying static analysis to C and C++, versus SPARK, is that the languages were not designed to allow developers to indicate the intent of their code. Likewise, many of the techniques available are deficient from a static analysis point of view, making it difficult at best for a static analysis tool to determine a programmer's intent.

Ada 2012 included the concept of contracts. This allows a developer to indicate what kinds of arguments can be accepted and what kinds of actions/results will occur. Contracts can set

up to emit code for this type of checking, but with SPARK these contracts can be used to analyze the code. The analysis can determine whether the code will actually do what the developer desires. The advantage of this more detailed definition and analysis is that the contract code is not required since rest of the code will perform as expected. For example, accessing data outside of the range of an array would not be possible since the indices used to access the data will be within range.

Most programmers will be unlikely to switch programming languages at this point, but picking up SPARK and Ada is a relatively easy chore for C and C++ programmers. There are many advantages to switching besides contracts, but starting to use static analysis tools with C and C++ will be a major step forward for most companies, as reducing bugs is important for safety and security too.

Static analysis tools are also useful in enforcing coding standards. They do add time to the development process, and they typically increase the tool costs, but they payback for both can be significant. While they will not eliminate all bugs, they will elevate the kinds of bugs that will occur. They will also allow developers to track down and fix those bugs instead of more trivial ones that the tools will flag.

There are a number of vendors that provide static analysis tools, including Adacore, Rogue Wave Software/Klockwork, Grammatech, LDRA, Parasoft, Programming Research, and Synopsys. There are also some open-source tools, including cppcheck and the Eclipse Codan (CODe Analysis) project. Most of these tools will support the majority of MISRA C/C++ rules.