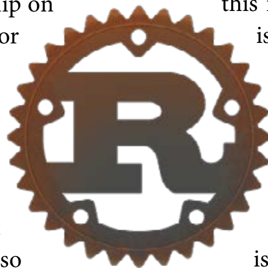
WILLIAM WONG | Technology Editor

Reflections on Rust

Technology Editor Bill Wong takes a crack at Rust, a relatively new programming language designed for safe and secure application development.

Many of you knzow that I advocate embedded programmers to try out new languages and techniques outside of their normal realm, which tends to be C and C++ programming (see "C Programmers, Time To Try Ada" on electronicdesign.com). In fact, C is still the favored language for embedded applications mostly because it supports every chip on the market. Unfortunately, C is a very good tool for shooting yourself in the foot.

The rise of the Internet of Things (IoT) has elevated the safety and security discussion that had been relegated to specific arenas like military, avionics, medical, and transportation. Even in these areas there were only some that required higher levels of safety and security, or so we thought. These days it is not uncommon for safety and security to be at the top of the list for concerns or features in a project. Some results from our *Electronic Design* 2017 Embedded Revolution survey highlight this concern.



This brings us back to the ways of addressing safety and security issues and how to address them. Coding standards, static analysis tools, and good design methodologies are just a few things that can help in this area, but having a programming language that helps rather than hinders is another item that can have a major impact on the resulting programs. Ada/SPARK *(see "Ada 2012: The Joy of Contracts" on electronicdesign.com)* and Java *(see "Java For Critical Jobs" on electronicdesign.com)* are two examples where safety and security were part of the design criteria for the languages.

The up-and-coming language to address this space is Rust. Rust has a number of features that lend itself to more bug-free code, including guaranteed memory safety using an inherent reference counting systems and threads without data races, just to name two major features.

I started learning Rust recently, and still have a long way to go in understanding the advantages and disadvantages, but I thought it would be useful to give some feedback to those looking for alternatives to C and C++.

First, Rust is a significant deviation from C and even C++, although it supports most of their semantics. Second, Rust looks somewhat like C and C++ because of its use of curly brackets and general flow control syntax, although this is true only in a cursory sense. In actuality, Rust is quite different from C and C++ in both syntax and semantics. Finally, Rust is new, evolving, and community supported. The community is robust and the compiler version is currently 1.16.0.

Now for a few details on Rust: It is built on LLVM, so its code generation and optimization is built on a tried-and-tested platform. Rust has an ownership and borrowing system for memory that I will talk about later, but it does not have a built-in garbage collector like Java. The Rust compiler is designed to generate high-performance application code like C and C++ and to be a fast compiler.

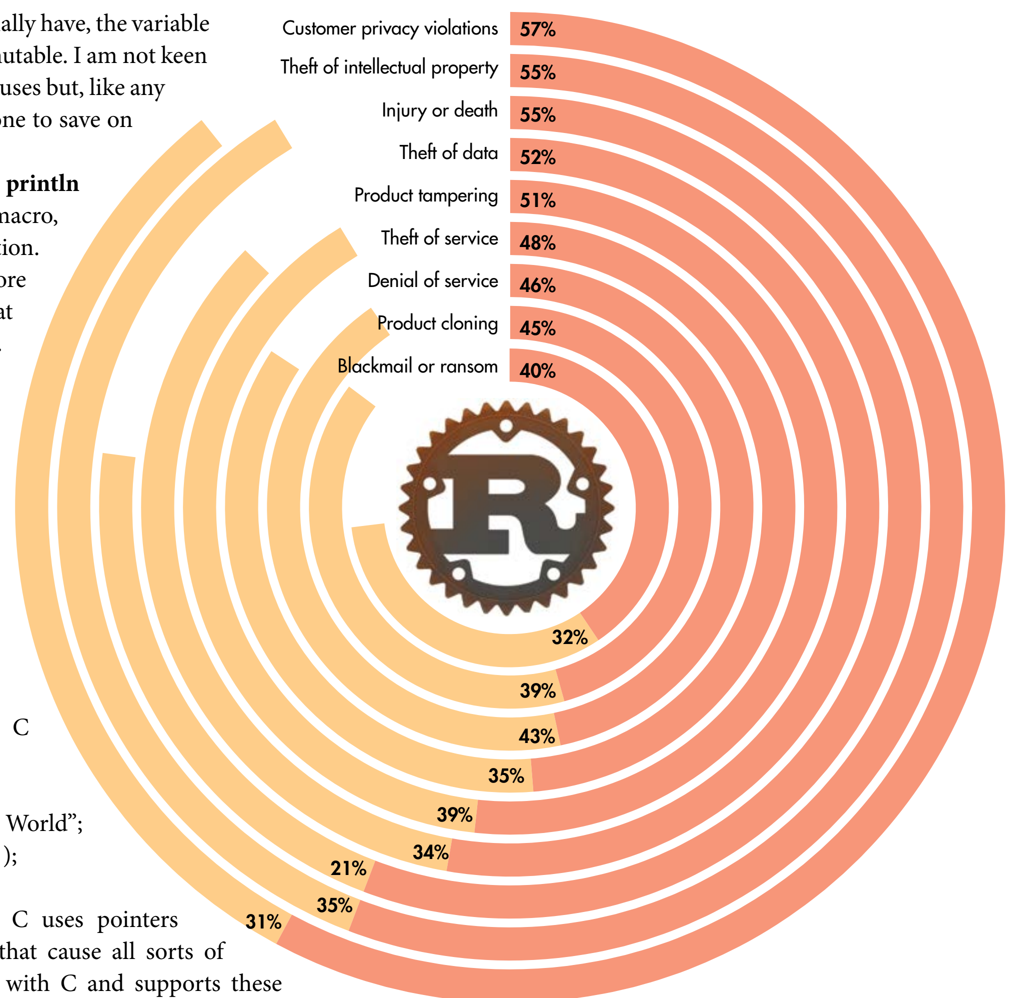
1. Hello Rust

There are other resources that will give you a good introduction to Rust, but here is my take on it, starting with a variation of Hello World.

```
fn main () {
        let hello = "Hello World";
        println!( "{}.", hello );
}
```

This looks a little like C, but **fn** and **let** give it away. The first indicates a function followed by the function name. The **main** function is the same as in C, the first thing called after initialization. The **let** statement is an assignment, but the data type in this example is inferred instead of explicitly defined, as with C or C++. In this case it is a string constant.

What's different with Rust is that our **hello** variable is immutable. It is assigned a value initially, but it will never



Security for just IoT applications has caused concern in a number of areas (from Electronic Design 2017 Embedded Revolution survey).

be changed. While it's possible to have mutable variables like C and C++ normally have, the variable name is prefixed by **mut**, for mutable. I am not keen on the abbreviations that Rust uses but, like any good cryptic language, it is done to save on typing.

The exclamation point after **println** indicates that the former is a macro, and this is a macro invocation. Macros are significantly more powerful than C macros that provide basic substitution. We won't get into the details of this particular macro, but the two arguments we are using include a formatting string, "{}.", and a value from the variable **hello**. The value, Hello World, replaces the curly brackets, and the following would be printed on the console.

Hello World.

This is a comparable C program.

```
void main () {
        char * hello = "Hello World";
        printf( "%s.\n", hello );
}
```

The big difference is that C uses pointers and zero terminated strings that cause all sorts of problems. Rust can interface with C and supports these constructs, but the defaults in Rust are the opposite, where exceptions must be explicitly noted as in Ada/SPARK. The idea is to prevent the programmer from doing things they should not but allowing that to occur it if the areas are clearly annotated.

2. Thanks for the Memory

As noted, Rust does pointers, but there is a lot more to the story. Essentially, Rust tracks the lifetime of pointers and references like SPARK can, so compile time checks can be applied to make sure the programmer is doing what they intend. Using references to objects that could have disappeared is not a good thing, and this can be check in most contexts.

Rust has a number of concepts, including object and related pointer lifetimes, as well as the idea of borrowers and ownership. In general, there is an owner of an object and references can be borrowed. An object cannot be released if there are borrowed references to it. All references have a lifetime, but they can often be inferred based on Rust language rules. There are also explicit lifetime parameters that can be used in various areas in the code, such as in function signatures.

As an aside, I will mention that Rust's use of semicolons may not be what you think. They are not statement terminators, as with C or Ada. In Rust, everything is an expression; a semicolon indicates that the value of the expression will be ignored and the result will be **nil**. The following shows both the lack of a semicolon and an indication of the lifetime of the function's result:

```
fn hello_world () -> & 'static str {
        "Hello World"
}
```

A call to **hello_world** could be used as the assigned value of **hello** in our prior Rust example. The ampersand, &, is a reference as with C++ and the **'static** modifier indicates the lifetime of the result. An error would be flagged if the result did not meet these criteria. Likewise, the compiler knows

what it is dealing with in terms of an object's lifetime where the function will be called.

Rust and Ada/SPARK have a lot to offer embedded developers, and could easily replace C and C++ in most applications. So far, learning Rust for me has been on par with taking up C++ and Ada/SPARK. It ain't easy, but the payoff is significant. Likewise, the availability of training can be a significant boost in getting started.

I have a number of languages under my belt, which has proved to be a benefit: The concepts are familiar, although the syntax and some of the semantics are different. The challenge for any programmer will be in understanding both the features and how to apply them on a regular basis.

The need for safe and secure programming languages should not be overlooked. Failures in the field—whether accidental or due to attacks—may jeopardize lives, property, and the companies that create the products. In the past, many have tried to get away limiting liability with an end-user license agreements (EULAs) that delivers software as-is. Having software that does not turn into a liability may be a better approach.