Springer Vieweg

ATZ extra

SOFTWARE-DEFINED VEHICLE

Implemented Better With Rust as Programming Language

itk ENGINEERING

© Shutterstock | whiteMocca

Rust Integration Based on Interoperability in Legacy Software

Classic programming languages are reaching their limits due to increasing complexity, fast development cycles and growing quality requirements. Rust has the potential to become the language of choice for state-of-the-art software development. ITK Engineering describes how built-in functions for software quality enable secure coding and minimize the effort for debugging and testing.

WRITTEN BY

Christopher Schwager
is Realtime Architectures Senior Expert at ITK Engineering GmbH in Rülzheim (Germany).

Digital features play a pivotal role across all industries. Apps and digital solutions are firmly entrenched in consumers' lives. Going forward, people are going to expect greater connectivity, automation, and customization. These expectations will be met with software (SW) solutions which will have major implications for the customer experience and the specifications for the underlying hardware. The proliferation of SW-defined elements is driving demand

for continuous improvement in SW development. Data will have to be exchanged constantly with the cloud to continuously optimize SW within the confines of the given hardware constraints. Vendors must accelerate their development cycles to continue differentiating products with new features and upgrades throughout their lifecycles.

SW is thus sure to grow even more complex. It will take more time to validate SW via extensive testing and debugging to prevent and redress flaws – especially as aspects such as functional safety figure ever more prominently in the equation. Part of the problem is that the classic C and C++ programming languages used to develop embedded SW lack basic safety guarantees, for instance, for memory and thread safety. These deficiencies are among the main causes of SW crashes. Many tools and methods have been developed to ensure quality through extensive validation. In a SW-defined world, the increasing costs associated with this approach will no longer be acceptable [1]. New solutions are needed to ensure efficiency and quality during programming. One could be a programming language such as Rust.

RUST AS MODERN PROGRAMMING LANGUAGE

Rust [2] has been gaining traction in many industries in recent years. Leading tech companies such as Google [3] and Amazon [4] have adopted this new language. This multi-paradigm programming language was designed to enable functional safety as well as security, parallel programming, and ease of learning, while serving performance-critical applications well. Rust's innovative memory management sets new standards, especially when it comes to functional safety. The concept of variable ownership guarantees memory safety at compile time. There is no need to apply other principles such as garbage collection at runtime. This significantly improves SW performance. Every variable and every value have an owner. Variables must be borrowed when other functions seek to manipulate the value. This goes to ensure that the value cannot be simultaneously changed by anyone else. The value is dropped when the owner goes out of scope. And the strict type system rules out implicit conversions between variables. Another aspect is the use of coding guidelines such as MISRA. A comparison has shown that Rust renders 80 % of the MISRA guidelines irrelevant. This results in greater efficiency and lower costs. What's more, Rust can match or exceed established languages in terms of performance, efficiency, and control [5]. Simple yet comprehensive and consistent, the toolchain treats developers to a unique experience. The bottom line: Because of its good performance and functional safety as well as security aspects, Rust lends itself to any kind of app – anything goes, from real-time embedded apps to web apps. It does indeed have the potential to replace established programming languages.

THE CHALLENGES OF INTEGRATING

Switching languages entirely is seldom a viable option. The obstacle is the legacy SW, which took a lot of effort to set up and then evolve over the years. In most cases, there is little point in redoing all this in another programming language. It would take far too much time and the costs would be prohibitive. On top of that, special tools were used to qualify and validate the SW. Another aspect relates to established SW platforms and technology stacks, which include operating systems such as Linux and QNX, as well as frameworks such as the Robot Operating System (ROS) and AUTOSAR. All these solutions have one thing in common: They were developed in C or C++ and provide application programming interfaces (APIs) in these programming languages. In order to take advantage of Rust, there has to be a migration path or a way to bridge the gap between the two worlds. The term for the latter is interoperability. This is not just about basic compatibility. It is also about the way SW components integrate with one another and how interfaces are implemented.

FOREIGN FUNCTION INTERFACE AS GATEWAY TO INTEROPERABILITY

The foreign function interface (FFI) is a mechanism that enables SW to use functions or services written in another programming language. It serves as a link between the calling conventions and semantics of the two programming languages. The application binary interface (ABI) plays a role at the machine code level by defining conventions such as memory layout and bit and byte encoding at that level. Ultimately, the two levels have to dovetail, **FIGURE 1**. Rust provides an FFI that allows function calls to and from C, **FIGURE 2**. This interface can also serve to interact with other programming languages such as C++. There are two main use cases – integrating C into Rust and vice versa. What they have in common is that the FFI has to be defined in Rust [6].

Integrating C into Rust starts with the C-API of the code to be integrated. The interface, both data types and function signatures, must be rendered in an external block in Rust. Rust assumes that external functions are unsafe, so calls must be packed into an unsafe block. This is why implementing an additional safe interface around the raw C interface is a good idea. This interface restores the lost guarantees. The compiler cannot verify that the declarations are correct. When integrating Rust into C, individual public functions can be prepared for use in C with the keyword extern "C" and the attribute *#[no_mangle]*. Compatible C headers must also be created. If the data types are compatible, there are no other special demands to bear in mind.

DATA TYPE INTEROPERABILITY

Scalar data types such as integers – for example, the 32-bit unsigned integer type u32 – and float/double floating point numbers are binary-compatible between Rust and C. The reason why floating point numbers are compatible is because both languages use an IEEE-754 standard-compliant representation. However, there are differences in Boolean and character data types. In C, zero represents false and any non-zero number means true. The latterly introduced C99 standard established an explicit data type (bool/_Bool) and corresponding values for true = 1 and false = 0. This is also the definition in Rust [7]. However, many C libraries use their own type definition based on an integer and corresponding macros. The data type length and Boolean values definitions may differ, so this can trigger undefined behavior at the interface.

The difference is even greater when it comes to character representation.

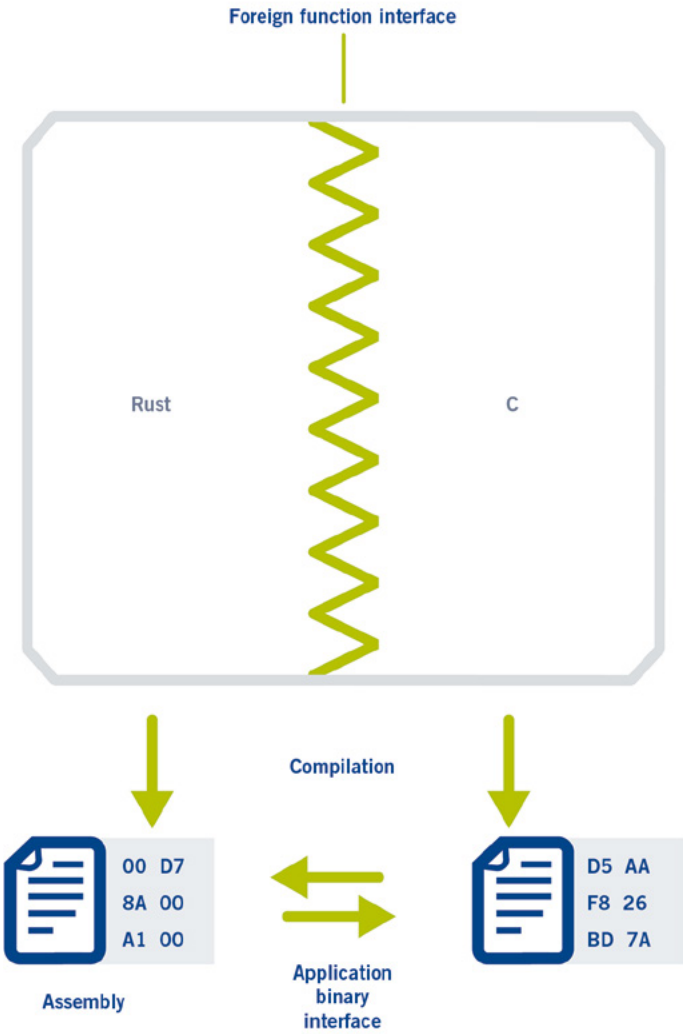


FIGURE 1 Interoperability at the FFI and ABI levels (© ITK Engineering)

Rust represents characters (char) as four-byte Unicode scalar values. In C, a character, typically an ASCII character, of the basic character set is encoded as an integer in one byte. This means that full interoperability based on the C char is not possible because only the basic character set is interchangeable between the two. Type conversions have to be done on the Rust side and invalid characters have to be filtered and processed accordingly on the C side.

Composite data types also present some obstacles. Take, for example, arrays: The length of the array plays a special metadata role in Rust alongside the actual data elements. This allows Rust to determine if the access is within the array boundaries. In C, an array name is a pointer to the first element of the array. An equivalent to the check performed in Rust is possible only in rare, exceptional cases. If the length is dynamic, it has to be specified as an additional argument alongside the start address. This means that in Rust, only a slice can be created when reconstructing from a C pointer and its length. Unlike an array, a slice has the great drawback that the length is not fixed at compile time. Therefore, a check for out-of-bounds access can only take place at runtime. An unsafe block has to be used to create the slice [9], which can result in undefined behavior.

As a rule, raw pointers are ABI compliant. Note that the Rust compiler cannot provide any guarantees, for example, as to memory safety. Thus, the use of raw pointers requires an unsafe block. One could instead use options and/or references, but that would mean that the caller is responsible for the quality of the data. User-defined data types such as structures harbor fewer potential sources of error. They can be converted to a C representation using the attribute #[repr(C)]. This resolves issues such as bit/byte padding, data alignment, and packing. **FIGURE 3** summarizes data type interoperabilities.

FFI DEFINITION

Writing a compatible interface between Rust and C manually would appear to be

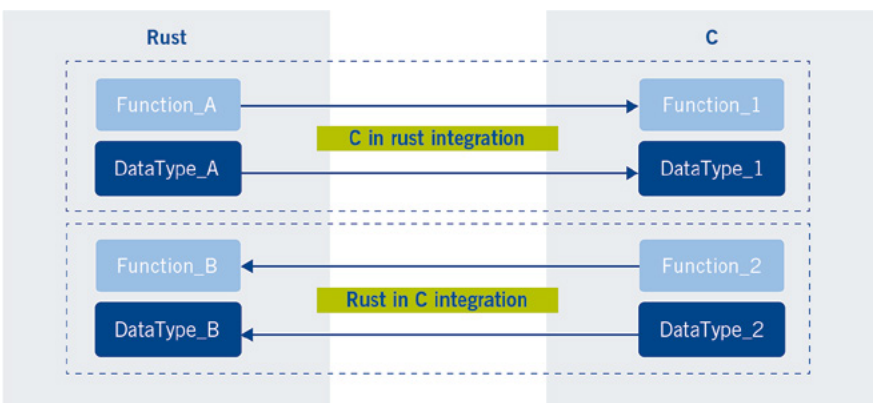
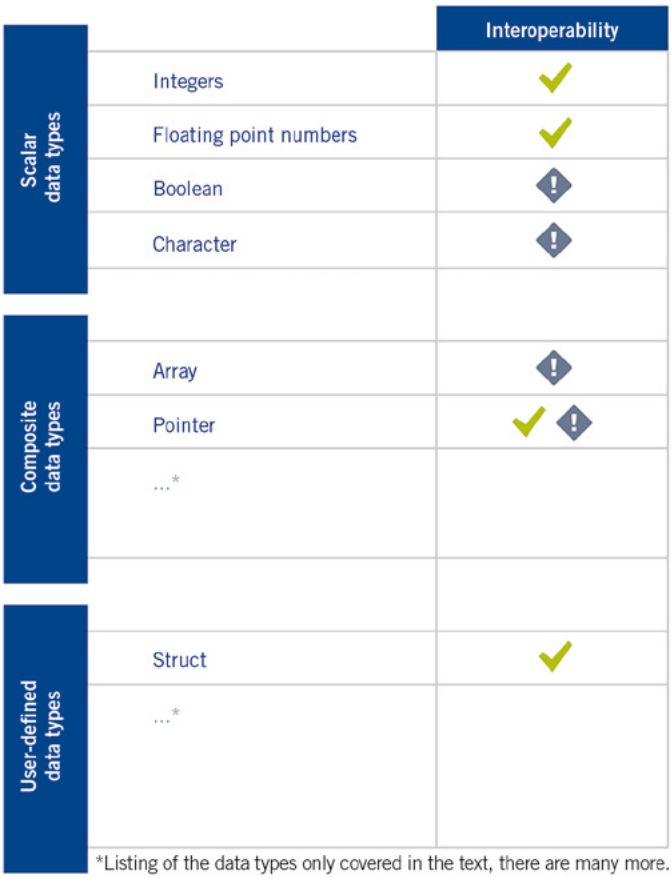


FIGURE 2 Various use cases for interoperability with C (© ITK Engineering)

FIGURE 3 Various data types are interoperable between the C and Rust programming languages (© ITK Engineering)



an onerous chore. However, the technical implementation is straightforward. There are FFI generators available such as bindgen [10] and cbindgen that can automatically generate the required Rust and C files. Experiencewise, this is common practice and a good option, unless this involves a complex interface with many dependencies. If C code is called from Rust, a safe wrapper can then be added manually. It should have properties that ensure the call is safe for all inputs and outputs despite the unsafe call to the C interface – including type conversion and has to take into account lifetimes and ownership, which includes, for example, a mechanism to release memory once it has been transferred fully from C to Rust – even in the event of a panic error.

In the opposite direction – that is, when calling Rust from C – the external Rust interface has to be defined manually. The goal here is to do without unsafe blocks while still defining a fault-tolerant interface: use *Option<&T>* or *Option<Box<T>>* rather than raw pointers to make it null pointer-tolerant and take into account lifetimes and ownership.

CONCLUSION

The software-defined world calls for new approaches that can cope with rising costs and growing complexity while ensuring efficiency and quality early on, during programming. Rust, as a safe and secure programming language, has this potential. However, switching languages altogether is rarely possible or practical. This is why programmers need ways to combine Rust with mainstream languages such as C. The compatible ABI is essential, but there are some issues to overcome regarding FFI definition. If these are resolved, the two languages will indeed be interoperable.

REFERENCES:

[1] Roland Berger (ed.): Computer on wheels Part 4, (July 2022). Online: https://content.roland-berger.com/hubfs/07_presse/Roland_Berger_Article_Computer_on_wheels_4_2022.pdf, access: March 21, 2024

[2] Rust Team (ed.): Rust. A language empowering everyone to build reliable and efficient SW. Online: https://www.rust-lang.org/, access: January 26, 2024

[3] Miller, S.; Lerche, C.: Sustainability with Rust. In: AWS (ed.): AWS Open Source Blog. Online: https://aws.amazon.com/de/blogs/open-source/sustainability-with-rust/, access: January 26, 2024

[4] Vander Stoep, J.; Hines, S.: Rust in the Android platform. Online: https://security.googleblog.com/2021/04/rust-in-android-platform.html, access: January 26, 2024

[5] Ng, V.: Rust vs C++, a Battle of Speed and Efficiency. In: Journal of Mathematical Techniques and Computational Mathematics 2 (2023), No. 6, pp. 216 -220

[6] Rust Community (ed.): The Rustonomicon. Foreign Function Interface. Online: https://doc.rust-lang.org/nomicon/ffi.html, access: January 26, 2024

[7] Beingessner, A: Notes on Type Layouts and ABIs in Rust. Online: https://faultlore.com/blah/rust-layouts-and-abis/#the-layoutsabis-of-builtins, access: January 26, 2024

[8] Rust Community (ed.): The Rust Reference. Online: https://doc.rust-lang.org/reference/types/textual.html, access: January 26, 2024

[9] Rust Community (ed.): The Rust Standard Library. Online: https://doc.rust-lang.org/std/slice/fn.from_raw_parts.html, access: January 26, 2024

[10] Rust Community (ed.): Rust-Bindgen. Online: https://github.com/rust-lang/rust-bindgen, access: January 26, 2024

IMPRINT
Special Edition 2024 in cooperation with ITK Engineering GmbH, Bergfeldstraße 2, 83607 Holzkirchen; Springer Fachmedien Wiesbaden GmbH, Postfach 1546, 65173 Wiesbaden, Amtsgericht Wiesbaden, HRB 9754, USt-IdNr. DE81148419

MANAGING DIRECTORS:
Stefanie Burgmaier I Andreas Funk I Joachim Krieger
PROJECT MANAGEMENT: Anja Trabusch

COVER PHOTO: © Shutterstock I whiteMocca

ITK Engineering

Stability, reliability and methodological expertise – this is what we have stood for since our founding in 1994. At all times, our customers have benefitted from our dedicated multi-industry know-how, especially in the fields of control systems design and model-based design. Customers can count on us – from conception through to deployment, we cover the entire development process.

Our areas of expertise include:

- Software development
- Hardware development
- Electrical & electronic systems
- System integration
- Software as a product
- Turnkey systems
- Customer specific development
- Technical consulting
- Seminars
- Quality assurance

The satisfaction of each of our partners and mutually respectful cooperation shape our corporate philosophy, in which four values are firmly anchored: Read more about this on the web.

V.1.0.0_e_2021

ITK Engineering GmbH
Headquarters: Ruelzheim
Im Speyerer Tal 6
76761 Ruelzheim, Germany
T: + 49 (0)7272 7703-0
F: + 49 (0)7272 7703-100
info@itk-engineering.com

www.itk-engineering.com
www.itk-career.com

Founded in 1994
Branch offices throughout Germany – ITK companies worldwide.

Follow us on: