



Rust Is Ready for Safety-Critical Firmware, but Certification Practice Is Not

Yonas Carvalho^{1*}, Tiago Grant², Madina Nurpeisov³

¹Department of Electrical Electronics, Tagus Institute of Electrical Electronics, Colombia

²Department of Electrical Electronics, Chao Phraya Academy of Electrical Electronics, Barbados

³Department of Electrical Electronics, Amazon School of Electrical Electronics, Finland

Article Info

Volume 1, Issue 1, 2026

doi: [10.62715/ijesd.2026.v1.i1.pp26-31](https://doi.org/10.62715/ijesd.2026.v1.i1.pp26-31)

Received: 19 January 2026

Accepted: 30 March 2026

Published: 20 May 2026

Abstract

Firmware for vehicles, aircraft and industrial plant is still written almost entirely in C, under coding guidelines that exist because the language permits whole classes of error. Rust removes several of those classes at compile time: safe Rust has no dangling pointers, no buffer overruns that go undetected, and no data races. This article argues that the language and its tooling have matured to the point where they can carry safety-critical firmware, and that the remaining obstacle lies in certification practice. The functional safety standards are worded neutrally, yet the working apparatus built around them, the coding guidelines, the coverage criteria, the tool qualification routines and the experience of assessors, was shaped by C and has only begun to adjust. The article sets out the technical case for readiness, including formal soundness results and the arrival of a qualified compiler. It then examines where practice lags: the lack of a mature and widely accepted coding guideline, unsettled rules for unsafe code, coverage measurement for a language with pattern matching and generics, a fast release cadence, and the treatment of third-party libraries. Objections from both sides are considered. The article closes with the steps that standards bodies, tool vendors and project teams would need to take for Rust to be adopted on its merits.

Keywords: Rust, Functional safety, Firmware, Software certification, Memory safety, Tool qualification

© 2026 Yonas Carvalho et al. This is an open access article under the CC BY license (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

1. Introduction

The dominant language of embedded firmware was designed in the early 1970s for writing an operating system on a minicomputer. C gives the programmer direct control of memory and asks in return that the programmer never make a mistake with it. The safety-critical industries have managed that bargain with process: restricted subsets such as MISRA C (MISRA, 2013), static analysis, review and extensive testing. The approach has worked well enough to put software in charge of brakes and flight controls. It has not removed memory errors, which large vendors and security agencies continue to identify as the leading class of serious software vulnerability (Miller, 2019; NSA, 2022).

* Corresponding author: Yonas Carvalho, Department of Electrical Electronics, Tagus Institute of Electrical Electronics, Colombia. E-mail: yonas.carvalho@tagus.edu

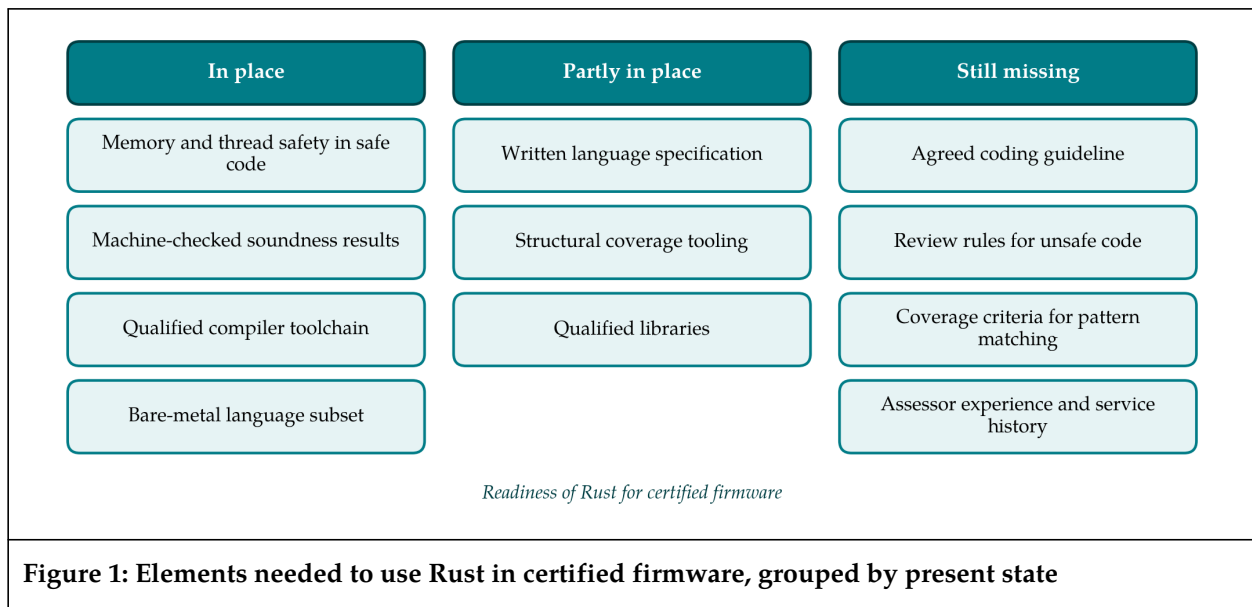
© 2026 Yonas Carvalho et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Rust offers a different bargain. Its ownership and borrowing rules, enforced by the compiler, rule out use-after-free, double free and data races in code that does not opt out of the checks (Matsakis and Klock, 2014). The guarantees cost nothing at run time and need no garbage collector, which is why the language is of interest for microcontrollers at all.

The position taken here is in the title. The language is ready, in the sense that a competent team can write firmware in it today that is safer by construction than the C it replaces. What is not ready is the practice of certification, the accumulated guidelines, criteria, tools and habits through which the standards are applied. The article defends both halves of the claim and considers what follows.

2. The Case That the Language Is Ready

Four developments support the first half of the claim. Figure 1 sets them beside the items that remain open, which later sections take up.



2.1. Guarantees with a formal basis

The safety claims of Rust are no longer only a matter of design intent. The RustBelt project gave a machine-checked proof that a core model of the language is sound, and that standard library abstractions which use unsafe code internally do not break the guarantees offered to safe callers (Jung et al., 2018). An accessible account of the result and its limits is given in Jung et al. (2021). No comparable result exists for C as used in firmware, where undefined behaviour is reachable from ordinary code and its absence must be argued program by program.

2.2. A workable bare-metal subset

Rust was built with a layer that needs no operating system and no heap. Firmware can be written against the core library alone, with interrupt handlers, memory-mapped registers and static allocation expressed in the type system. The Tock operating system demonstrated early that a kernel for a microcontroller with tens of kilobytes of memory can be written this way, using the language's guarantees to isolate drivers from one another without hardware support (Levy et al., 2017). The acceptance of Rust into the Linux kernel in 2022 was a further signal that the language can meet the demands of low-level systems code.

2.3. A qualified toolchain

For years the decisive objection was that no Rust compiler had been qualified for safety work. That changed in 2023, when the Ferrocene toolchain, a downstream of the main compiler, was qualified by an independent assessor for use under ISO 26262 and IEC 61508. The qualification came with a written specification of the language as the compiler implements it. A team can now cite a qualified compiler in its

safety case, as it would for a C compiler.

2.4. Evidence about unsafe code

Firmware must touch hardware, and in Rust that requires blocks marked unsafe. Studies of open source code find that unsafe blocks are typically few, small and used for a narrow set of purposes such as calling foreign functions ([Astrauskas et al., 2020](#)). Studies of reported memory safety bugs in Rust software find that they involve unsafe code, as the design intends ([Qin et al., 2020](#); [Xu et al., 2021](#)). The practical effect is to concentrate review. Where a C code base must be examined everywhere for memory errors, a Rust code base marks the places that need that examination.

3. Where Certification Practice Lags

The standards themselves are not the obstacle. ISO 26262 asks for a language with unambiguous definition, strong typing and the use of subsets and guidelines ([ISO, 2018](#)); IEC 61508 makes similar recommendations ([IEC, 2010](#)); DO-178C states objectives and leaves the language open ([RTCA, 2011](#)). Rust satisfies the spirit of these requirements more fully than C does. The difficulty lies in the layer between standard and project. Table 1 compares the state of that layer for the two languages.

Table 1: State of certification practice for C and for Rust on six recurring concerns		
Concern	Established practice for C	Present state for Rust
Language definition	ISO standard with catalogued undefined behaviour	Specification tied to one qualified toolchain; no international standard
Coding guideline	MISRA C, checked by many qualified analysers	No guideline of comparable maturity and acceptance
Low-level code	Whole code base treated as potentially unsafe	Unsafe blocks isolated; rules for them not fully settled
Structural coverage	Statement, branch and MC/DC tooling long available	Tooling emerging; criteria for match expressions and generics debated
Third-party code	Few dependencies, vendored and assessed individually	Rich package ecosystem that invites many transitive dependencies
Assessor familiarity	Decades of precedent	Few completed assessments to refer to

3.1. Guidelines and the rules for unsafe code

An assessor reviewing C firmware asks for MISRA compliance and a list of justified deviations. The question is routine and so is the answer. For Rust there is no guideline with that standing. Many MISRA rules are simply unnecessary, since the compiler rejects what they forbid, but a guideline is still needed for what remains: when unsafe is permitted, how it is encapsulated and reviewed, how panics are handled, which language features are excluded. The semantics of unsafe code are themselves still being refined. Proposals such as Stacked Borrows define which pointer aliasing patterns are permitted ([Jung et al., 2020](#)), and until a model of that kind is adopted the precise obligations on the author of an unsafe block are not fully fixed. [Evans et al. \(2020\)](#) also show that unsafe code reaches further than its line count suggests, because many safe functions depend on unsafe ones through their call chains.

3.2. Coverage and verification tooling

The highest assurance levels require structural coverage up to modified condition and decision coverage, a criterion defined with the boolean expressions of procedural languages in mind ([Hayhurst et al., 2001](#)). Rust raises questions the tutorial literature does not answer. A match expression with nested patterns and guards is a decision of a kind C does not have. Generic functions are compiled once per type they are used with, and macros generate code the author never wrote out. Whether coverage is to be shown on the source as written or on each instantiation is a matter on which practice has yet to converge. Stack depth

analysis, timing analysis and qualified static analysers exist for C in several competing products and are only now appearing for Rust.

3.3. Run-time failure behaviour

Safe Rust turns many memory errors into panics: an index out of range or a failed arithmetic check stops the offending computation instead of corrupting memory. That is the right default, and it moves a question into the safety case that C projects have long-practised answers for. A panic in firmware must lead somewhere defined, whether a reset, a degraded mode or a safe state, and the team must show which operations can panic and that the handler behaves as claimed. Tools that list the panic sites reachable in a build, or that prove their absence, are less established than their counterparts for run-time errors in C, and assessors have little guidance on how much of this evidence to demand.

3.4. Release cadence and dependencies

The Rust compiler is released every six weeks, and the ecosystem moves with it. A safety project freezes its toolchain for years. A qualified downstream compiler bridges the two, at the price of trailing the language. Libraries are the harder problem. Embedded Rust projects draw on a public registry of packages, and a modest application can pull in dozens of them indirectly. Each is pre-existing software that the standards require to be assessed or qualified. C projects face the same rule but meet it with a handful of vendored components, and the routines of assessment have grown up around that habit.

4. Objections

The claim can be attacked from either side. One objection holds that the language is not ready: that reliance on unsafe code in drivers hollows out the guarantee, and that a single compiler implementation is too narrow a base. The first point has force, and the reply is one of degree. Unsafe Rust is no worse than C, and it is confined and labelled. The second point also applies to several widely used C compilers, each of which is in effect the definition of its own dialect and extensions. Compiler correctness is a concern for every language, as the effort needed to verify even one C compiler formally makes plain ([Leroy, 2009](#)).

The opposite objection holds that certification practice is ready enough: the standards are neutral, a qualified compiler exists, and the rest is ordinary engineering. This underrates how much of certification is precedent. An assessor's confidence comes from having seen the argument made before, and a project that must invent its own coding guideline, its own coverage rationale and its own policy for dependencies carries a schedule risk that few managers will accept for a first product.

A third view is that restricted C with modern static analysis already delivers what Rust promises. The evidence that language subsets reduce defects is thinner than their status implies, a point [Hatton \(2004\)](#) made about MISRA C itself. Guidelines and analysers detect a share of memory errors after they are written. A type system that rejects them before the program compiles is a stronger form of the same idea, and it does not depend on which rules a project chose to enforce.

5. Implications

If the argument holds, effort should go into the practice and not the language. Guideline bodies can publish and maintain a Rust guideline with the authority that MISRA C has, covering unsafe code, panics and feature restrictions, so that assessors have a document to ask for. Standards committees can clarify how structural coverage criteria apply to pattern matching, generics and macros. Tool vendors can bring coverage, stack and timing analysis to qualified status, and library maintainers can offer core components with the evidence an assessment needs.

Training deserves a place on the same list. Engineers who have written C for a career need time to become fluent in ownership and lifetimes, and reviewers and assessors need enough of the language to judge an unsafe block or a dependency policy on its merits. Organisations that qualify staff for safety work can add the language to their curricula, and assessment bodies can publish what they expect to see in a Rust submission. Clear expectations of this kind lower the risk of a first project more than any single

tool does.

Project teams have a part as well. Adoption need not be wholesale. A new component with a narrow interface, such as a protocol parser or a bootloader stage, can be written in Rust and linked into an existing C system, with the boundary between the two treated as the unsafe surface it is. Each such component assessed adds to the stock of precedent, the resource in shortest supply. Teams should be candid in their safety cases about what the compiler guarantees and what it does not: logic errors, timing faults and incorrect requirements are untouched by any type system.

None of these steps requires a change to the standards themselves, which is why the present moment matters. The first projects to certify Rust firmware will set the patterns that later ones copy, and patterns set in haste, such as blanket bans on unsafe code or coverage arguments borrowed unexamined from C, would be hard to dislodge. It is better that the guidance be written deliberately, with language designers, tool builders and assessors taking part together.

6. Conclusion

Rust removes by construction a class of defects that the safety-critical industries have spent decades containing by process, and it does so with a formal basis, a bare-metal subset and now a qualified compiler. The apparatus through which software is certified has not caught up. Guidelines, coverage criteria, analysis tools, library evidence and assessor experience were all built around C and are only partly available for its successor. Closing that gap is institutional work more than technical work, and the sooner it is done the sooner firmware can stop paying for errors that a compiler is able to prevent.

Conflict of interest

To be declared by the author(s) before publication.

Funding

To be declared by the author(s) before publication.

References

- Astrauskas, V., Matheja, C., Poli, F., Muller, P. and Summers, A.J. (2020). [How do programmers use unsafe Rust? *Proceedings of the ACM on Programming Languages*, 4\(OOPSLA\): 1-27.](#)
<https://doi.org/10.1145/3428204>
- Evans, A.N., Campbell, B. and Soffa, M.L. (2020). [Is Rust used safely by software developers? *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering \(ICSE\)*, 246-257.](#)
- Hatton, L. (2004). [Safer language subsets: An overview and a case history, *MISRA C. Information and Software Technology*, 46\(7\): 465-472.](#)
- Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J. and Rierson, L.K. (2001). [A Practical Tutorial on Modified Condition/Decision Coverage. NASA/TM-2001-210876, NASA Langley Research Center, Hampton, VA.](#)
- IEC (2010). [IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, 2nd edition.](#) International Electrotechnical Commission, Geneva.
- ISO (2018). [ISO 26262: Road Vehicles, Functional Safety, 2nd edition.](#) International Organization for Standardization, Geneva.
- Jung, R., Dang, H.-H., Kang, J. and Dreyer, D. (2020). [Stacked borrows: An aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, 4\(POPL\): 1-32.](#) <https://doi.org/10.1145/3371109>
- Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D. (2018). [RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2\(POPL\): 1-34.](#)
<https://doi.org/10.1145/3158154>
- Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D. (2021). [Safe systems programming in Rust.](#)

- Communications of the ACM*, 64(4): 144-152. <https://doi.org/10.1145/3418295>
- Leroy, X. (2009). *Formal verification of a realistic compiler*. *Communications of the ACM*, 52(7): 107-115. <https://doi.org/10.1145/1538788.1538814>
- Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P. and Levis, P. (2017). *Multiprogramming a 64kB computer safely and efficiently*. *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 234-251. <https://doi.org/10.1145/3132747.3132786>
- Matsakis, N.D. and Klock, F.S. (2014). *The Rust language*. *ACM SIGAda Ada Letters*, 34(3): 103-104. <https://doi.org/10.1145/2692956.2663188>
- Miller, M. (2019). *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. Presentation at BlueHat IL, Tel Aviv.
- MISRA (2013). *MISRA C:2012, Guidelines for the Use of the C Language in Critical Systems*. MIRA Limited, Nuneaton.
- NSA (2022). *Software Memory Safety*. Cybersecurity Information Sheet, National Security Agency, Fort Meade, MD.
- Qin, B., Chen, Y., Yu, Z., Song, L. and Zhang, Y. (2020). *Understanding memory and thread safety practices and issues in real-world Rust programs*. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 763-779.
- RTCA (2011). *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Washington, DC.
- Xu, H., Chen, Z., Sun, M., Zhou, Y. and Lyu, M.R. (2021). *Memory-safety challenge considered solved? An in-depth study with all Rust CVEs*. *ACM Transactions on Software Engineering and Methodology*, 31(1): 1-25.