



SURFACE VEHICLE/AEROSPACE RECOMMENDED PRACTICE

JA1020™

MAR2026

Issued

2026-03

Safety and Cybersecurity Recommendations for the Use of the Rust Language in Critical Systems

RATIONALE

This document discusses guidance on using the Rust programming language in critical systems.

Rust is a general purpose programming language which combines high run time efficiency and modern design patterns with an ecosystem for rigorous compile time checking. While this language is well suited for critical systems, the current lack of common coding guidelines and best practices is considered an impediment to its use in this domain. This recommended practice exists to complement the existing cybersecurity guideline.

FOREWORD

This document extends the cybersecurity rules in "ANSSI Guide - Programming Rules to Develop Secure Applications With Rust" toward functional safety standards like ISO 26262, RTCA DO-178C, and RTCA DO-332. It provides guidance for the design, implementation, and testing stage.

TABLE OF CONTENTS

1.	SCOPE.....	4
1.1	The Programming Language Rust.....	4
1.1.1	The Connection between Safety and Security.....	4
2.	REFERENCES.....	4
2.1	Applicable Documents	4
2.1.1	ISO Publications.....	4
2.1.2	RTCA Publications	5
2.1.3	Rust-Specific Publications	5
2.1.4	MISRA Publications	5
2.2	Related Publications	5
3.	DEFINITIONS	6
4.	GENERAL GUIDELINES FOR RUST DEVELOPMENT IN CRITICAL OR SAFETY-RELATED SYSTEMS	6
5.	STRUCTURE OF THIS DOCUMENT	7

SAE Executive Standards Committee Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2026 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, or used for text and data mining, AI training, or similar technologies, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 1-877-606-7323 (U.S. and Canada only)
Tel: 1-724-776-4970 (outside U.S. and Canada)
Fax: 724-776-0790
Email: CustomerService@sae.org
http://www.sae.org

SAE WEB ADDRESS:

Full document details: SAE JA1020_202603

For more information on this standard, visit https://www.sae.org/standards/content/JA1020_202603/

Documents

Standard, visit Sharing Hub

6.	SOFTWARE PLANNING PROCESS.....	9
6.1	Tool Selection and Confidence	9
6.1.1	Dependency Management.....	10
6.2	Recommendation to Evaluate and Define Project-Specific Guidelines	10
6.2.1	async/await	10
6.2.2	Extension Traits	11
6.2.3	Recent Additions to the Language	11
6.2.4	Variable Shadowing	11
6.2.5	Code Review Environment	11
6.2.6	Macros.....	12
6.2.7	Module File Naming	13
6.2.8	Dependency Criteria	13
6.2.9	Project-Specific Trait Recommendations.....	13
6.2.10	Other Project-Specific Design Decisions	13
6.3	Definition of a Language Subset.....	14
6.3.1	Use of the "Unsafe" Keyword.....	14
6.3.2	Panic versus Result	18
6.3.3	Use of Deref.....	18
6.3.4	Variable Shadowing	20
6.3.5	Interior Mutability	21
6.3.6	Conditional Compilation	21
6.3.7	Deprecated Interfaces.....	22
6.3.8	Unstable Language Features.....	22
6.3.9	Import and Wildcards	22
6.4	Recommended Design Principles.....	22
6.4.1	Naming Conventions and Style Guides	24
6.4.2	Rustc Linter Rules.....	24
6.4.3	Clippy Rules	26
6.4.4	Sanitizers	27
6.4.5	Strong Typing.....	27
6.4.6	Inheritance	29
6.4.7	Parametric Polymorphism	29
6.4.8	Overloading.....	29
6.4.9	Type Conversion	29
6.4.10	Error Handling	29
6.4.11	Structured Control Flow	30
6.4.12	Exception Management	30
6.4.13	Dynamic Memory Management.....	31
6.4.14	Concurrency.....	31
6.4.15	Async Rust.....	31
6.4.16	Modularity.....	32
6.4.17	Relationship to MISRA C and C++ Rules	33
6.4.18	Cybersecurity and Supply Chain Attack Prevention	33
6.4.19	Specific Cybersecurity Recommendations	34
6.4.20	Documentation	34
6.4.21	Marker Traits	34
6.4.22	Known Limitations of (Current) Rust.....	34
6.4.23	Common Mistakes in Rust.....	35
7.	SOFTWARE REQUIREMENTS AND SPECIFICATION	35
8.	SOFTWARE ARCHITECTURAL DESIGN.....	36
8.1	UML.....	36
8.2	Module Design	36
8.3	Scheduling and Interrupts	36
8.4	Shared Resources	36
8.5	Formal Verification	36
8.6	Data Flow Analysis.....	36

9.	SOFTWARE UNIT DESIGN AND IMPLEMENTATION.....	36
9.1	Result Handling.....	37
9.2	Data and Control Flow	37
10.	SOFTWARE UNIT VERIFICATION	37
10.1	Coverage Metrics	37
11.	SOFTWARE INTEGRATION AND VERIFICATION	37
12.	SOFTWARE TESTING	37
13.	ADDITIONAL COMMENTS ON SUPPORTING PROCESSES.....	37
13.1	Configuration Management (Part 8, Clause 7)	37
13.2	Commentary on "Proven in Use Argument" (Part 8, Clause 14)	38
14.	NOTES	38
14.1	Revision Indicator.....	38
APPENDIX A		39
APPENDIX B	INDEX	44

1. SCOPE

This document provides guidance to using Rust in critical and safety-related software. The document summarizes how the usage of Rust supports in arguing safety according to ISO 26262 or RTCA DO-178C combined with RTCA DO-332. Cybersecurity best practices are referenced as these requirements largely overlap with those implemented for safety.

As the Rust language is still evolving, this document targets the 2021 and 2024 editions of the language. Older or newer editions might require additional or changed rules and guidelines. Generally, the newest available edition of the language should be selected as newer editions remove ambiguities and outdated parts from the language.

1.1 The Programming Language Rust

Rust is a modern system programming language (inception in 2006, first public stable release in 2015) which combines a high level of control with extensive correctness checks at compile time. It achieves memory safety without relying on garbage collection and is the first general purpose language to combine borrow checking with move semantics. Thus, it is well suited for embedded, critical, and safety-related systems that typically have stringent response time requirements.

Rust language development started at Mozilla and is now supported by an industry-backed foundation. The founding members are AWS, Huawei, Google, Microsoft, and Mozilla. Decisions about language features are voted on by various contributor-elected teams.

Rust's strengths are correctness, speed, and concurrency. It was designed to prevent utmost errors at compile time.

1.1.1 The Connection between Safety and Security

While standards and threat models are different between functional safety and cybersecurity, there is a large overlap in practices between the two domains. There is no safety without security as a security breach can alter the designed behavior of the system. Gaps in system safety design, e.g., electrostatic discharge (ESD) or electromagnetic interference (EMI) handling, can enable sensitive data access by malicious actors. Thus, functional safety rule sets for correctness and maintainability help to address aspects required for cybersecurity.

This document has been developed with achieving both safety and security in mind, and although it does not reference individual cybersecurity standards, the coding guidelines are helpful for security as well. Due to the different threat weighing, the disposition between making a rule mandatory or recommended will vary and a project should cross-check the recommendations in this document against its specific safety and security goals.

Following the rules in this document alone does not fulfil all cybersecurity or safety requirements as they require established processes which exist beyond the scope of this document.

2. REFERENCES

2.1 Applicable Documents

The following publications form a part of this specification to the extent specified herein. Unless otherwise indicated, the latest issue of SAE publications shall apply.

2.1.1 ISO Publications

Copies of these documents are available online at www.iso.org/store.

ISO 26262:2018 Road vehicles – Functional safety

ISO/PAS 8926:2024 Road vehicles – Functional safety – Use of pre-existing software architectural elements

2.1.2 RTCA Publications

Available from RTCA, Inc., 1150 18th Street NW, Suite 910, Washington, DC 20036, Tel: 202-833-9339, www.rtca.org.

RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification

RTCA DO-332 Object Oriented Technology and Related Techniques Supplement

2.1.3 Rust-Specific Publications

The following Rust-specific references and guidelines are referenced by this document:

Rust language reference, <https://doc.rust-lang.org/stable/reference/>

ANSSI Guide - Programming Rules to Develop Secure Applications With Rust, v1.0, <https://anssi-fr.github.io/rust-guide/>

Rust style guides, <https://doc.rust-lang.org/stable/style-guide/>

Rust API guidelines, <https://rust-lang.github.io/api-guidelines/>

"Unsafe" Code Guidelines Reference, <https://rust-lang.github.io/unsafe-code-guidelines/>

Standard library developers Guide, <https://std-dev-guide.rust-lang.org/>

2.1.4 MISRA Publications

Copies of these documents are available online at <https://misra.org.uk/>.

MISRA C:2025 Addendum 6, Applicability of MISRA C:2025 to the Rust Programming Language, <https://misra.org.uk/app/uploads/2025/03/MISRA-C-2025-ADD6.pdf>

MISRA discussion of compliance by coding standards, <https://www.misra.org.uk/compliance>

2.2 Related Publications

The following publications are provided for information purposes only and are not a required part of this SAE Recommended Practice.

Rust specification by the Ferrocene Project, <https://spec.ferrocene.dev/>

Rust "unsafe" coding guidelines working group ("t-opsem" in zulip), <https://github.com/rust-lang/unsafe-code-guidelines>

Safety-Critical Rust Consortium, <https://github.com/rustfoundation/safety-critical-rust-consortium>

Rust Formal Methods Interest Group, <https://rust-formal-methods.github.io/>

Mara Bos: Rust Atomics and Locks, O'Reilly, January 2023 or <https://marabos.nl/atomics/>

The Rustonomicon: The Dark Arts of Unsafe Rust, <https://doc.rust-lang.org/nomicon/>

High Assurance Rust, Developing Secure and Robust Software, <https://highassurance.rs/>

Coercion (describes the remaining implicit type conversions in Rust), <https://doc.rust-lang.org/reference/type-coercions.html#coercion-types>

Undefined Behavior at "unsafe" boundaries (describes necessary checks in "unsafe" code to prevent undefined behavior), <https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html>

3. DEFINITIONS

These definitions are provided for convenience. For a more in-depth discussion, refer to the ["Unsafe" Code Guidelines Reference](https://rust-lang.github.io/unsafe-code-guidelines/) (<https://rust-lang.github.io/unsafe-code-guidelines/>) by the Rust WG "unsafe code guidelines."

This document follows the classification of MISRA documents regarding the distinction between Rules and Directives. *Rules* generally permit a clear-cut decision of compliance regardless of context; *Directives* require additional context to decide whether they were followed or violated.

3.1 SAFE

The term "safe" is defined differently in the context of functional safety and Rust.

In the domain of Functional Safety, a system is considered safe if it sufficiently minimizes exposure to and impact of hazards, as identified during the concept phase of a product.

"Safe" Rust code restricts to a subset of the language and verifies absence of undefined behavior by relying on the soundness of invoked modules. "Unsafe" in Rust terms is not a statement about hazards and risks; it merely means that the compiler needs assistance to guarantee absence of undefined behavior.

3.2 UNSAFE

"Unsafe" Rust code explicitly documents the risk of causing undefined behavior if it is used incorrectly. Thus, "unsafe" Rust code has documented boundaries which require *manual* checking, like with a traditional safety manual. "Unsafe" Rust code is not automatically a hazard in itself, but any violation of the designed preconditions for the code creates undefined behavior, which can invalidate the safety concept.

3.3 UNDEFINED BEHAVIOR

To avoid confusion with the term's use in functional safety, where unsafe indicates hazardous situations such as unsafe state and unsafe behavior, this document will put "unsafe" in quotes when talking about the Rust qualifier.

Undefined behavior means the loss of any guarantees about the behavior of a program; this also affects all other parts the faulty code can interfere with. Thus, this marks a forbidden area for safety-related programs. Refer to the C++ Standard explaining this topic: <https://en.cppreference.com/w/cpp/language/ub>.

3.4 UNSOUND

Unsound code exhibits undefined behavior from safe Rust and thus invalidates the safety guarantees by Rust.

4. GENERAL GUIDELINES FOR RUST DEVELOPMENT IN CRITICAL OR SAFETY-RELATED SYSTEMS

Program code used in critical or safety-related systems should maximize correctness, understandability, maintainability, and verifiability. Unexpected behavior, a superset of undefined behavior, should be minimized.

Thus, features and practices that improve maintainability and reduce the risk for errors should be used. Features and practices that could confuse code reviewers and hide errors should be avoided.

The choice of the language alone cannot make a system safe; appropriate specification, design, verification, and validation cannot be avoided. The Rust language can, on the other hand, reduce the complexity of testing through:

- Extensive correctness checks at compile time, and
- Separation of concerns via modularization, and
- Minimized interference at runtime.

Rust offers helpful features which automate correctness verification by the compiler:

- Ownership and borrowing rules: Prevent data races and multiple deallocation
- Lifetimes: Prevent dangling references and use after free
- Soundness: Excludes undefined behavior outside parts explicitly marked by the keyword "unsafe"
- Typestate: Avoids confusion between domains, e.g., physical units, by making them explicit
 - Can prevent logic errors by restricting functionality to states
 - Can compile-time-limit exclusive access to resources to specific parts of the code
- Sum types: No access to returned values without handling potential errors
 - Type-safe errors and type-safe absence of a value replace special indicator values of the same type
- Propagation: Explicit error handling and early return
- Trait bounds: Interfaces are only accessible when preconditions are fulfilled
- Resource acquisition is initialization: Resources are automatically freed at scope end
- Exhaustive pattern matching: Avoids unhandled states

In addition, Rust provides runtime features to detect program errors which cannot be checked at compile time:

- Spatial memory safety through pervasive bounds checking
- Integer overflow checking: Always on during development, advisable in final product

The following features improve readability and maintainability:

- Higher levels of abstraction: Iterators and other functional programming constructs
 - Higher order functions: Fn traits and closures
- Generic code with explicit constraints (trait bounds)
- Traits: Common interfaces across types
- Async+await: Enables coding state machines in a linear representation
- Multiple versions of a dependency are distinguished and can be combined across a single binary

5. STRUCTURE OF THIS DOCUMENT

This document follows a typical safety-related project lifecycle and gives step-specific recommendations for the usage of the programming language Rust. As both ISO-26262 and RTCA DO-178C/332 structures are following the V model of software development, we use this structure in this document as well.

This matches the structure of the referenced guidelines RTCA DO-178C and ISO 26262.

RTCA DO-178C section	Name	Work products	ISO 26262:2018 equivalent	RTCA DO-332	This document
4	Software planning	Software plans (5×) Software standards Methods and tools Lifecycle plans	Part 6: 5 Part 6: Annex C Part 2: 6 Part 8: 11	OO.4	6
5	Software development	Requirements Design Code Integration	Part 6: 6 Part 6: 7 Part 6: 8 Part 6: 10	OO.5	9
6	Software verification	Reviews and Analyses (4×) Test plans and Results	Part 6: 9 Part 6: 10 Part 6: 11	OO.6	11
7	Configuration management	Baselines Traceability Problem reporting Change Control Release	Part 6: Annex C Part 8: 7 Part 8: 8 Part 2: 5 Part 2: 6	unchanged	13.1
8	SW QA	Audit reports Conformity report	Part 2: 6	OO.8.2	unchanged
9	Certification liaison process	Plan for Software Aspects of Certification SW accomplishment summary Configuration index	Not applicable	OO.9.0-2	unchanged
10	Certification	Not applicable	Not applicable	unchanged	unchanged
11	SW lifecycle data	Not applicable	Not applicable	OO.11.1, 7, 8, 14, 20, 21	unchanged
12	Additional considerations	SW reuse Tool qualification Redundancy	Qualification: Part 8: 11 Part 8: 12	OO.12.1.1, OO.12.3.4.2	13.2

ISO 26262-6:2018 clause	Name	Work products	RTCA DO-178C equivalent	This document
5	General topics	Documented SW development environment	4	6
6	Specification of SW safety requirements	SW Safety requirements specification HW-SW-Interface Specification (updated) SW verification report	4	6.1.1
7	SW architectural design	SW architectural design specification Safety analysis report Dependent failure analysis report	4	8
8	SW unit design	SW unit design specification SW unit implementation	4, 5	9
9	SW unit verification	SW verification specification SW verification report (updated)	6	10
10	SW integration	SW verification specification SW verification report (updated) Embedded SW	5	11
11	Embedded SW testing	SW verification specification (updated) SW verification report (updated)	6	12
Annex C	Software configuration	Specification, data, verification	7	unchanged
Annex D	Freedom from interference	Not applicable	Not applicable	unchanged

ISO 26262-8:2018 clause	Name	Work products	RTCA DO-178C equivalent	This document
7	Configuration management	CM plan	7	13.1
11	Tool confidence	SW tool criteria evaluation report SW tool qualification report	12.2	6.1
12	Qualification of SW components	Not applicable	Not applicable	6.1
14	Proven in use argument	Not applicable	12.1	13.2



6. SOFTWARE PLANNING PROCESS

This section matches Section 4 of RTCA DO-178C and ISO 26262-6:2018 clause 5.

6.1 Tool Selection and Confidence

The purpose of this step is to select an adequate strategy for sufficient confidence in the use of tools and libraries, e.g., the Rust compiler and crates, for a specific project. This can range from acquiring a qualified tool over establishing sufficient quality control mechanisms for the tool to fully distrusting the inputs and compilation process and doing a full verification of the resulting binary.

Selecting a qualified compiler is a more common practice within the automotive industry, while the aerospace industry relies more on verification of the resulting binary.

Generally, there are three situations in the use of software tools in safety-related projects described in ISO-26262-8:2018 Section 11:

- A malfunction of the tool has no impact on the safety of the product. This should only apply to tools which cannot influence the product, and thus is not applicable to a compiler or crates used during compilation.
(TI1 results in TCL1 independently of tool error detection)
- Any malfunction of the tool will be systematically detected during development.
(TD1 results in TCL1 independent of tool impact level)
- Otherwise, the tool needs to be evaluated and, if required, qualified for the use in safety-related projects. Depending on the criticality level (ASIL) and the confidence to detect malfunction (middle to unknown), the rigor of qualification increases.

For the Rust compiler and crates, this gives several options:

- A malfunction of the compiler or errors in the used crates will be detected with high confidence (TD1) because all requirements are verified on the output of the compiler (that is, on binary level, e.g., on target hardware or a sufficiently trusted simulator).
- Based on ISO 26262, enough confidence from use and suitable development process can provide sufficient confidence for lower criticality levels (ASIL B for TCL3 and ASIL C for TCL2), while for higher criticality levels, a validation is also suggested. Collecting enough evidence on correctness is considered feasible for an organization determined to go this way, but obtaining a qualified compiler or crate should be considered from an economical standpoint.
- A full qualification of the tool for the target architecture is necessary. This involves rigorous testing against a specification and applying a safety standard to its development. Pooling this effort across organizations is common for complex tools, e.g., the compiler.

A compiler or crate qualification according to ISO 26262-8:2018 clause 11 or 12 or ISO/PAS 8926:2024 will involve:

- (Safety) requirements, and
- Derivation of sufficient test cases with traceability to the requirements, and
- A history of changes to the tool with regard to these requirements and test results.

The test cases form a validation set, typically based on a (language) standard. Please note that a qualification is specific to the tested versions of the tool, to the target architectures, and to the tested (compiler) configurations.

For open-source projects already existing, quality standard mechanisms can contribute to satisfying these requirements. Guidelines for this judgement include Automotive SPICE, CMMI, or ISO/IEC 33000.

Binary to source traceability and intermediate representation forms (MIR) can aid to increase confidence in the correctness of the compiler and crates.

At the time of creation of this document, the most complete specification of the Rust language, suitable for compiler qualification, is provided by the Ferrocene Language Specification (FLS). The FLS has been adopted by the Rust Project for harmonization with their language Reference and ongoing synchronization to the language's evolution.

Generally, the most recent compiler and edition feasible should be selected as newer versions reduce language ambiguities and fix known cases of unsoundness. Using editions below 2018 is not recommended in safety-related code bases. Similarly, the newest available resolver should be selected.

Directive COMPILER: The tool confidence level classification for the compiler and crates should be documented and implemented.

6.1.1 Dependency Management

Existing safety and security standards mandate careful checking of external dependencies to ensure that they meet the same level as the project specific code. They should be (e.g., at least be) version controlled, code-reviewed or qualified, monitored for safety and security related bugs, and tested against safety requirements.

Maintaining a list of build environment components and included sources, called SBOM or software bill of material, is already required by the existing safety guidelines. Putting Cargo.lock under configuration management can satisfy a large part of this requirement.

See 6.4.18 and refer to the Libraries section in ANSSI.

Directive DEP_VERS_CTRL: Dependency code needs version control.

Directive DEP_REVIEW: Dependency updates require code review, testing or qualification evidence.

Rule DUPL_DEP: The number of duplicated dependencies should be monitored and minimized.

In combination with traits, duplicate versioning often leads to difficult to understand compiler errors about mismatching types of the same name.

As duplicated dependencies, multiple included versions of the same crate, are more difficult to upgrade and patch, cybersecurity considerations often forbid duplication.

6.2 Recommendation to Evaluate and Define Project-Specific Guidelines

Some features and practices within the Rust language can confuse the unaware but offer significant benefits when used wisely. For these, a conscious decision early in the project is recommended.

6.2.1 async/await

async/await creates additional constraints on objects (often Send), a state machine which has no exact location in the source code, exposes incomplete language features (e.g., async functions in traits), and introduces additional sources of mistakes (async cancellation). But on the other hand, async can significantly reduce code complexity for stateful processing by writing stateful code linearly. Asynchronous code looks familiar to blocking code using multiple threads, but it needs less resources.

The timing of async code will be influenced by the executor; multi-threaded executors make an analysis more difficult.

This example illustrates a linearly expressed state machine which parses data received via a network.

```
async fn parse_header() -> Result<u32, ()> {
    // this function can stop, remember its state and later continue at each .await
    Ok(match next_byte().await {
        1 => next_byte().await as u32,
        2 => (next_byte().await as u32) | ((next_byte().await as u32) << 8),
        _ => Err(()),
    })
}
```

Directive ASYNC: Document whether `async/await` is permitted.

6.2.2 Extension Traits

Extension traits add methods to an interface and get usable once the trait is imported. Unfortunately, the location of the method is not obvious from the calling code and imports do not mention the function name, requiring documentation or source code browsing tools to locate the implementation. On the other hand, extension traits are common in Rust crates and improve readability by chaining on the right instead of the left side; thus, reading order matches execution order (in left to right languages).

```
// we define a new trait to add methods to an existing type
pub trait RemoveSecond {
    fn remove_second(&mut self);
}

impl<T> RemoveSecond for Vec<T> {
    fn remove_second(&mut self) {
        self.remove(1);
    }
}

fn main() {
    let mut a: Vec<u32> = vec![1, 2, 3, 4, 5];
    a.remove_second();
    println!("a is now {a:?}");
}
```

Directive EXT_TRAITS: Document whether usage of extension traits is restricted.

6.2.3 Recent Additions to the Language

Recently stabilized features will raise the minimum supported Rust version (MSRV) and might collide with requirements for qualified tooling.

Directive MSRV: Define Minimum Supported Rust Version.

6.2.4 Variable Shadowing

As identifying correct uses of variable shadowing might require manual checking, a general decision against shadowing might facilitate automated checking. See 6.3.4 for examples of good and bad use of this feature.

Directive SHADOW: Define acceptable use of shadowing.

6.2.5 Code Review Environment

While tooling like the Rust analyzer greatly improves the usability and productivity of the language, it degrades the insight into the source code when these tools are not present, e.g., during a web-based code review. Thus, additional type annotations might be necessary to understand the code, depending on the support provided by review platforms.

Directive REVIEW: Define tooling and documentation requirements to facilitate code reviews.

6.2.6 Macros

There are four different types of macros in Rust which fall into two major categories: declarative macros and three types of procedural macros. Procedural macros are typically defined in a separate crate.

Generally, the new syntax enabled by macros should match existing language patterns to minimize misunderstandings about the generated code.

As of the 2021 edition, declarative macros, which are defined by `macro_rules! name` and invoked as `name!`, are key to emulating a variable number of function arguments and primarily should be used for this purpose.

Declarative macros can also avoid code repetition and thus improve readability and maintainability. So these macros should be used whenever repeated code has a high chance of introducing inconsistencies or errors, especially if these are difficult to review, e.g., due to a high amount of special characters, or whenever a mistake is difficult to detect. Other ways to factor out repetitions, e.g., closures, functions, or procedural macros, should be preferred where possible.

The three types of procedural macro are derive, attribute-like, and function-like macro.

Derive macros, like `#[derive(Name)]`, add functionality to a type, typically applying a mechanism to all elements of a structure or tuple. Standard derive macros provide key language features to structured data types like default initialization, equality and hashing, copying and cloning, serialization and deserialization. As these macros can only append code, their use is recommended and they should be designed with generalized re-use in mind.

Attribute macros, like `#[name]`, and function-like macros, like `name!()`, change the token stream they are applied to and thus should be used with extreme care. As these macros often play a critical role to enhance the usability of valuable but complex constructs, it is not recommended to avoid them everywhere, provided that they are typically deployed in system libraries, like `defmt` and `async executors`.

Macros should be designed with requirements, documentation, and testing, including testing for errors on incorrect use. Test coverage requirements are identical to generic code (see 6.4.7). Note that this results in test and configuration management requirements on build scripts and their dependencies, including potential input data or files. This makes macros that download data during compilation less practical.

RTCA DO-332 OO.1.6.2.7 provides additional guidance for scripting under the name "virtualization techniques." This implies applying the code qualification process to both the macro definition and invocation and ensuring that the macro processor is included in the compiler qualification.

Directive MACROS: Define acceptable use of macros.

Directive TEST_MACROS: Define and execute macro testing strategy.

6.2.6.1 Macro Hygiene

Macro hygiene is a key property of macros in Rust and needs to be understood when designing and reviewing macros.

Declarative macros work in the namespace of its *definition*; thus, they are called hygienic as the name scope is locally predictable. This means that Rust code with declarative macros expanded in text form might no longer compile correctly.

```
macro_rules! double {
    ($x:expr) => {{
        let a = 42; // never used
        $x * 2
    }};
}

fn main() {
    let a = 15;
    let y = double!(a);
    println!("{}", y);
}
```

In this example, the expanded code "let y = { let a = 42; a * 2 };" returns a value of 84, but the parameter scope is preserved when processing the macro, so the original output is 30.

Procedural macros are unhygienic, and thus all type names should include the full path to avoid ambiguities. On the other hand, the expanded code compiles correctly and thus can be reviewed as usual.

6.2.7 Module File Naming

There are three equivalent options for placing code into a submodule: a module clause inside the parent module, a file named like the module in the same directory as the parent, and a mod.rs file inside a subdirectory.

Rule MOD_FILE: A project should define an unambiguous rule on when to use which location variant.

While the module_name/mod.rs convention places the module and its submodules into the same directory, the module_name.rs convention is considered more idiomatic. Embedding a module with curly braces should be limited to internal (e.g., for testing) or very small modules.

6.2.8 Dependency Criteria

A project should specify its requirements on dependencies early; doing this later will result in unnecessary rework. Common criteria for dependencies include quality (see 6.1.1 on dependency management) and licenses. A tool like cargo-deny can check for license, exclude explicitly listed crates, absence of security advisories, and trusted origin.

Directive DEP_LICENSE: Define permitted licenses for dependencies.

6.2.9 Project-Specific Trait Recommendations

Commonly used traits like Default, Hash, and Debug improve the usability of types. Due to the orphan rule, they can only be defined in the same module as the type. But, coverage requirements push toward minimizing code.

Requiring some trait implementations by default increases the effort for the type library but can benefit many downstream consumers. A project should make a conscious decision on this tradeoff.

Directive STD_TRAITS: Define which standard traits should be defined on types.

6.2.10 Other Project-Specific Design Decisions

See 6.4.12 for a discussion of panicking strategies.

6.2.10.1 Lock Poisoning

If a lock or mutex is held by a panicking process, the lock becomes poisoned to prevent accessing inconsistent data protected by the lock. Any future attempt to acquire that lock will return a failure. Continuing the panic by unwrapping the guard smart pointer is considered good practice even in projects which otherwise restrict the use of panic.

A safety project should define its strategy for poisoned lock handling.

```
use std::sync::Mutex;

fn main() {
    let mutex: Mutex<u32> = Mutex::new(38);
    // the unwrap here will directly propagate the panic during unwinding
    *(mutex.lock().unwrap()) += 4;
    println!("Value {}", mutex.lock().unwrap());
}
```

Directive MUTEX_UNWRAP: Define acceptance of unwrap on poisoned mutex.

6.2.10.2 Hardware Interaction and "Unsafe" Functions

While it might be possible to mark all parts of the program which need careful manual review of assumptions as "unsafe," Rust defines "unsafe" as causing undefined behavior once documented invariants are violated. Clearly all potential introduction of undefined behavior should be restricted to "unsafe" blocks, but whether effects outside of the Rust abstract machine should be taken into consideration should be defined by the project.

For example, you should carefully control access to functionality in the following situations:

- A serial port is connected to the voltage regulator or clock system, potentially causing wrong computations by low voltage or high frequencies
- A function variant is compiled for a specific architecture variant, causing wrong results on incompatible devices
- Linux provides write access to the current process's memory via `/proc/self/mem`, thus potentially changing memory outside of the knowledge of the compiler, thus causing undefined behavior

e.g., by documenting and tracking these effects, and wrapping them in globally safe higher level abstractions, but whether this qualifies these functions "unsafe" is debated. Overusing "unsafe" will dilute the focus from the code, which requires careful inspection.

Directive UNSAFE_SCOPE: Which system level interactions qualify a function as "unsafe" should be documented.

Directive HW_EFFECTS: Potentially safety relevant effects by "safe" as well as "unsafe" functions should be documented.

Functions providing unconstrained DMA transfers, however, qualify as "unsafe" because of their similarity to writing to raw pointers.

6.3 Definition of a Language Subset

The purpose of restricting a language's usage to a subset of features and/or syntax is elimination of **Undefined Behavior** (aka UB, exhibition of incorrect runtime operations that violate the safety and functionality assumptions of the programmer).

The language design of Rust strongly favors reliable code, especially compared to commonly used systems languages such as C or C++. Reliability includes memory-safety, type-safety, data-race prevention, enforced error-handling, and encapsulation. Thus, the following recommendations are meant to further enhance an already remarkably safe environment.

Hint: Often, the presence and absence of problematic patterns can be diagnosed via linker symbols in the object files, e.g., use of `UnsafeCell` or `panic`.

6.3.1 Use of the "Unsafe" Keyword

See the explanation of the term "unsafe" in Section 3.

- Unless the "unsafe" keyword is used, the Rust language is automatically restricted to a UB-free safe subset. This guarantee is enforced by a combination of compile-time verification and lightweight runtime checks.
- Operations like raw pointer dereference, type transmutation, and calling a Foreign Function Interface (FFI) all require the "unsafe" keyword to extend language functionality beyond the default, safe subset. In such cases, automated guarantees are forfeited, and the programmer must manually verify absence of UB. This requires rechecking the safety argumentation within the full module containing the "unsafe" code even for changes to safe code.

UB-free code, whether safe or partially "unsafe," is considered to be **Sound**. Because of the precise language specification, compiler-specific behavior is not an issue for sound Rust code.