ISO 26262-6:2018 to Rust 1.93.1

Complete Language and Standard Library Mapping

Reference for Safety-Critical Rust Coding Standards (QM to ASIL D)

Revision 2 — February 2026

Working reference for Safety-Critical Rust Consortium — Coding Guidelines Subcommittee

Baseline: Rust stable 1.93.1 (2026-02-12), Edition 2024 (no nightly features)

Table of Contents

Note: This document uses Word heading styles. To generate an automatic table of contents, insert a TOC field in Word and update.

1. Introduction and Scope

This document maps ISO 26262:2018 (Road vehicles — Functional safety) software development expectations onto the Rust programming language and its standard library. It is intended to be used as an input to a project- or consortium-specific coding standard, and to support a defensible safety case (QM through ASIL D) for Rust-based automotive software.

1.1 Normative baselines and versioning

All technical guidance in this revision is anchored to the following baselines:
• ISO 26262-6:2018 clauses and tables (software development at the software level).
• Rust stable 1.93.1 standard library documentation (std/core/alloc) and Rust Reference keywords.
• Ferrocene Language Specification (FLS) as the qualification-oriented normative description of Rust language behavior, where applicable.

Because Rust evolves, this mapping is explicitly versioned. Any change to the Rust toolchain version, the Edition, or the set of approved dependencies requires re-assessing the impacted rows in the inventories in Sections 10–11 and updating the safety argument.

1.2 QM and ASIL applicability

ISO 26262 defines four Automotive Safety Integrity Levels (ASIL A–D) and a non-safety classification QM (Quality Management). QM items do not impose ISO 26262 safety requirements, but they still benefit from the same disciplined engineering practices. This document therefore provides a graded profile from QM (least restrictive) to ASIL D (most restrictive).

1.3 Interpretation of ISO 26262 table symbols

ISO 26262 tables classify methods and principles using symbols:
• “++” = highly recommended
• “+” = recommended
• “o” = neither recommended nor discouraged
and some clauses apply to ASIL values shown in parentheses as recommendations rather than requirements. This mapping preserves the intent by translating it into enforceable rules and evidence expectations.

1.4 Classification codes used in this mapping

Every Rust language construct, standard library module, or API surface is classified per safety level using the following codes:
• P — Permitted (allowed without additional justification beyond normal review/testing)
• HR — Highly recommended (preferred default)
• M — Mandatory (required when applicable)
• R — Restricted (allowed only under stated constraints and documented rationale)
• Q — Qualified-only / Trusted component (allowed only inside pre-approved, independently reviewed ‘TCB’ modules such as HAL/FFI/allocator)
• X — Prohibited (not allowed in certified code)

• U — Unavailable/unstable/experimental (not allowed in certified builds; requires nightly or non-qualified behavior)

1.5 Completeness rule and default-deny policy

Completeness is enforced by a default-deny rule: any Rust feature, attribute, macro, or standard library API not explicitly classified in Sections 10–11 is treated as X (Prohibited) for ASIL code, until reviewed and added. QM may optionally operate with default-allow, but doing so weakens re-use across ASIL levels; therefore, this document recommends using the same default-deny mechanism for all safety levels.

To keep the inventories complete as Rust evolves, projects should generate an automated API inventory (e.g., from rustdoc JSON) during CI and compare it against the approved allowlist. Any delta (new or removed items) must trigger a review and a documented mapping update.

1.6 ISO 26262-6:2018 clause and table cross-reference

This section provides an explicit mapping between the structure of this Rust language/library mapping and the corresponding ISO 26262-6:2018 clauses, tables, and annexes referenced throughout.

Document section	ISO 26262-6:2018 reference	Scope / notes
Section 3 (Clause 5)	Clause 5, Table 1	General topics and modelling/coding guideline topics.
Section 4 (Clause 7)	Clause 7, Tables 2-4	Software architectural design: notations, principles, and verification methods.
Section 5 (Clause 8)	Clause 8, Tables 5-6	Software unit design and implementation: notations and unit design principles.
Section 6 (Clause 9)	Clause 9, Tables 7-9	Software unit verification: verification methods, test derivation, and structural coverage metrics.
Section 7 (Clause 10)	Clause 10, Tables 10-12	Software integration and verification: methods, test derivation, and architectural-level structural coverage.
Section 8 (Clause 11)	Clause 11, Tables 13-15	Testing of embedded software: test environments, methods,

		and test case derivation.
Section 9 (Configurable Software)	Annex C	Configurable software guidance (configuration/calibration management considerations).

2. Review Notes and Critiques of Draft v1

This revision started from a prior internal draft and applies several corrective and completeness-driven improvements. The critiques below are intended to be constructive review feedback and to explain why the document was restructured.

ISO 26262 table alignment: multiple tables were mis-numbered or mapped to the wrong content. For example, ISO 26262-6 Table 1 concerns topics for modelling/coding guidelines; architectural design principles are in Table 3, and unit design principles are in Table 6. Structural coverage metrics for unit testing are in Table 9; function/call coverage at the architectural level is Table 12.
Missing ISO topics: Table 1 includes an explicit concurrency topic (1i) which must be addressed by coding guidelines; earlier drafts focused on concurrency later but did not trace it back to Table 1.
Incomplete inventory: the standard library mapping covered only a subset of core/std modules and did not enumerate the full module/macro surface of Rust 1.93.1 (including experimental and deprecation-planned areas). A complete mapping must start from a complete inventory.
Insufficient QM coverage: earlier drafts began at ASIL A; this revision adds QM as an explicit profile and clarifies which restrictions are safety-motivated vs general quality-driven.
Ambiguity around panics and unwinding: earlier drafts discussed unwrap/expect but did not specify a complete panic policy (panic=abort vs unwind), FFI boundaries, or how panic-related APIs in std/core are treated per ASIL.
Unclear governance: a complete mapping must include an update mechanism (default-deny + automated inventory diff) so that new Rust releases or new dependencies cannot silently bypass the coding standard.

All subsequent sections incorporate these corrections and extend the mapping to a complete module/macro inventory for core/alloc/std as of Rust 1.93.1.

3. ISO 26262-6:2018 Clause 5 — General Topics & Table 1

ISO 26262-6 Clause 5 establishes expectations for the software development process and environment, including the selection of modelling/design/programming languages and the use of guidelines. Table 1 lists the topics that coding/modelling guidelines should cover. In Rust, many of these topics are partially satisfied by the language design (ownership, typing, safe/unsafe boundary), but they still require explicit project rules, tool configuration, and evidence.

3.1 ISO 26262-6:2018 Table 1 — Topics to be covered by modelling and coding guidelines

ISO topic	ISO intent	Rust mechanisms	Coding standard implications	ASIL notes

1a Low complexity	Control complexity to improve correctness, reviewability, and testability.	Modules/visibility; ownership; iterators; type-driven design; compiler + Clippy complexity lints.	Define quantitative limits: function length, cyclomatic complexity, nesting depth, generic/lifetime complexity. Prefer small cohesive modules and explicit interfaces.	++ for all ASIL; enforce tighter thresholds at ASIL C/D.
1b Language subsets	Define and enforce a safe subset of the language to avoid error-prone constructs.	Safe Rust as default; crate-level forbid/deny unsafe; edition pinning; feature-gating; linting.	Default profile: forbid unsafe in application crates; allow unsafe only in named TCB crates with documented invariants and independent review. Prohibit nightly features.	++ for all ASIL; ASIL C/D require strict subset and inventory gating.
1c Strong typing	Prevent type confusion and unintended conversions.	No implicit numeric conversions; enums/newtypes; Option/Result; const generics; trait bounds.	Require fixed-width integer types at interfaces; use newtypes for units; restrict `as` casts; require TryFrom for narrowing conversions; avoid typographical errors via naming rules.	++ for all ASIL.
1d Defensive implementation techniques	Detect and handle errors, invalid inputs, and internal failures.	Result/Option; bounds checks; checked/saturating arithmetic; assertions; pattern matching.	Define error-handling policy (no panics for expected errors); define precondition checks; require checked arithmetic where overflow possible; define assertion usage per build mode.	+ for ASIL A/B; ++ for ASIL C/D.
1e Well-trusted design principles	Prefer proven design principles (modularity, encapsulation, RAII).	RAII/Drop; composition via traits; exhaustive match; explicit ownership; zero-cost abstractions.	Prefer state machines encoded as enums; enforce encapsulation via visibility; define resource ownership patterns and lifecycle rules; document invariants and safety mechanisms.	+ for ASIL A/B; ++ for ASIL C/D.
1f Unambiguous graphical representation	Ensure design representations are unambiguous and understandable.	Rustdoc for API docs; module dependency graphs; UML/Arch diagrams generated from Cargo metadata.	Mandate architecture diagrams for safety-related crates, including component interfaces and data/control flow. Keep diagrams consistent with code via CI checks where	++ for ASIL B–D; + for ASIL A.

			feasible.	
1g Style guides	Consistency and readability to support reviews and maintenance.	rustfmt; Clippy; Rust API Guidelines; project-specific lints.	Define rustfmt profile and lock it; deny warnings; define mandatory Clippy lint set; ban wildcard imports; require documentation for public APIs.	++ for all ASIL.
1h Naming conventions	Avoid ambiguity and improve traceability.	Compiler lints for non_snake_case, non_camel_case_types, etc.	Define naming for signals, units, safety mechanisms, and diagnostic IDs; require consistent prefixes for safety-related items; prohibit shadowing and confusing names.	++ for all ASIL.
1i Concurrency aspects	Prevent unintended interference via shared resources and concurrency.	Send/Sync; ownership; Mutex/RwLock/Atomics; no data races in safe code; explicit synchronization primitives.	Define approved concurrency model: message passing vs shared state; lock ordering; forbid `static mut`; restrict atomics to documented orderings; require deadlock analysis and bounded blocking; define RT scheduling constraints.	+ for ASIL A/B; ++ for ASIL C/D (often required by architecture).

4. ISO 26262-6:2018 Clause 7 — Software Architectural Design (Tables 2–4)

Clause 7 focuses on constructing and verifying a software architecture that satisfies software safety requirements and supports implementation and verification. The architecture must capture static aspects (component structure, interfaces, data types) and dynamic aspects (control flow, concurrency, timing).

4.1 ISO 26262-6:2018 Table 2 — Notations for software architectural design

Rust does not prescribe a single architectural notation, but it provides strong support for architecture documentation through explicit module boundaries, visibility control, and auto-generated API documentation (rustdoc). For ASIL C/D, semi-formal representations (state machines, sequence diagrams, timing models) are typically needed alongside rustdoc.

Notation type (Table 2)	Rust-aligned artifacts	Guideline / evidence
Natural language (++)	Architecture doc + rustdoc module docs	Mandatory rationale for key decisions; bidirectional traceability to safety requirements.
Informal notations (++/+)	Block diagrams, interface sketches, simple sequence diagrams	Allowed for early design; must be converged to semi-formal for ASIL C/D where timing/concurrency matters.
Semi-formal notations (+/++)	UML/SysML/statecharts; timing models; interface contracts	Preferred for safety mechanisms, mode management, and partitioning; keep synchronized with code via review

		gates.
Formal notations (+)	Formal specs, model checking, theorem proving	Use selectively for high-risk algorithms, concurrency protocols, and unsafe abstractions; prioritize for ASIL D hotspots.

4.2 ISO 26262-6:2018 Table 3 — Architectural design principles mapped to Rust

Table 3 architectural principles map naturally to Rust's crate/module system and to explicit interface boundaries. The key safety contribution is to keep components small, cohesive, and isolated, and to manage shared resources and timing determinism explicitly.

Principle	Rust mapping	Key rules	ASIL tightening	Notes / examples
1a Hierarchical structure	Crates → modules → submodules; layered architecture; explicit dependency direction.	Define architectural layers (e.g., HAL → platform → safety services → application). Enforce via crate boundaries and dependency linting.	C/D: no layer violations; safety mechanisms isolated in dedicated crates.	Use `pub(crate)` and `pub(in path)` to enforce encapsulation.
1b Restricted size/complexity	Metrics + linting; idiomatic small functions; avoid mega-modules.	Set limits for LOC, cyclomatic complexity, nesting, generic bounds; require refactoring when exceeded.	C/D: stricter thresholds; require independent review on exceptions.	Prefer composition over deep trait hierarchies.
1c Restricted interface size	Small public APIs; narrow traits; explicit types.	Minimize `pub` surface; forbid wildcard re-exports; use newtypes to prevent accidental misuse.	D: treat all shared interfaces as safety-related; require formal interface review.	Prefer `pub(crate)` for internal reuse.
1d Strong cohesion	Cohesive modules with single responsibility; private helpers.	Group by domain concept; avoid 'util' grab-bags; require module-level docs describing responsibilities and invariants.	C/D: explicit design rationale for module boundaries.	Use internal modules to encapsulate unsafe or concurrency details.
1e Loose coupling	Dependency inversion via traits; avoid global state; explicit data flow.	Prefer passing dependencies explicitly; avoid hidden singletons; restrict `lazy_static`/global caches.	C/D: forbid circular dependencies; require dependency graph checks.	Use message passing for cross-component interactions where practical.
1f Scheduling properties	Deterministic execution model; avoid unbounded work; bounded loops; time budgets.	For RT code, require bounded execution time per cycle; avoid allocation on hot paths; avoid dynamic dispatch in timing-critical code.	D: timing analysis required; no hidden blocking; bounded locks.	Use `#[inline(never)]` selectively to support WCET analysis (tool-dependent).
1g Restricted interrupts	Interrupt handlers as minimal,	Keep ISRs short; no allocation; no locks	C/D: formally justify ISR design; verify	In bare-metal Rust, use a qualified HAL +

	deterministic, side-effect controlled functions; use atomics/critical sections.	that can deadlock; communicate via lock-free queues or deferred processing.	priority scheme; avoid shared mutable state.	critical-section abstraction.
1h Spatial isolation	Memory protection + partitioning; Rust's memory safety complements but does not replace MPU/MMU partitioning.	No shared mutable globals; isolate safety partitions into separate processes/partitions where needed.	D: require evidence of freedom from interference; use MPU/MMU or OS partitioning.	Rust prevents many memory errors but not timing/resource interference.
1i Shared resource management	Mutex/RwLock/ Atomics; ownership; explicit resource arbitration; avoid global contention.	Define lock ordering; require bounded blocking; prefer lock-free only with documented memory ordering; resource usage budgets.	C/D: deadlock analysis; resource usage evaluation (stack/heap/time).	Applies to shared HW resources (buses, peripherals) and SW resources.

4.3 ISO 26262-6:2018 Table 4 — Verification methods for the software architectural design

Table 4 lists verification methods (walk-through, inspection, simulation, prototype generation, formal verification, control/data flow analysis, scheduling analysis). In Rust projects, these map to architecture reviews, static analysis, model-based simulations (where applicable), and timing/resource analysis.

Method	Rust-aligned practice	Evidence artifacts	ASIL expectations
Walk-through (++)/+(o)	Structured architecture review with stakeholders	Review records, issue log, decisions, traceability to requirements	++ at ASIL A; + at ASIL B; o at ASIL C/D (still useful but not sufficient).
Inspection (+/++)	Formal inspections of architecture and interface specs	Inspection checklist, defect metrics, sign-off	+ at ASIL A; ++ at ASIL B–D.
Simulation (+/++ for D)	Model-level or harness-level simulation of dynamic behavior	Simulation results, assumptions, coverage of modes	+ at ASIL A–C; ++ at ASIL D for timing/concurrency risks.
Prototype generation (o/++ for D)	Executable spike for risky interfaces/protocols	Prototype report and lessons learned	Typically o at ASIL A/B; +/++ at C/D when uncertainty is high.
Formal verification (o/+)	Formalize and verify critical protocols, unsafe abstractions, or control logic	Proofs, model checking results, assumptions and limitations	Primarily for ASIL C/D hotspots.
Control flow analysis (+/++)	Control-flow graph analysis; restrict complex branching; review unsafe code	Static analysis reports; complexity metrics	+ at ASIL A/B; ++ at ASIL C/D.
Data flow analysis (+/++)	Data-flow analysis for initialization, lifetimes, shared resources	Static analysis; taint/dataflow results	+ at ASIL A/B; ++ at ASIL C/D.
Scheduling analysis (+/++)	WCET/timing budget analysis; priority inversion review	Timing models; measurement reports; lock analysis	+ at ASIL A/B; ++ at ASIL C/D (often required).

5. ISO 26262-6:2018 Clause 8 — Software Unit Design & Implementation (Tables 5–6)

Clause 8 focuses on unit-level design and implementation rules that prevent systematic faults and produce verifiable, readable source code. Rust eliminates some C/C++ failure modes by construction, but Clause 8 still requires explicit coding rules, especially for allocation, unsafe code, macros, and error handling.

5.1 ISO 26262-6:2018 Table 5 — Notations for software unit design

Unit-level design descriptions should be detailed enough to support implementation and verification. In Rust, this typically means:
• module-level docs describing responsibilities and invariants,
• state machine definitions (enums + transitions),
• interface contracts (preconditions/postconditions), and
• data structure invariants (including unsafe invariants when applicable).

5.2 ISO 26262-6:2018 Table 6 — Design principles for software unit design and implementation

Principle	Rust mapping	Default rule	ASIL C/D tightening	Rationale
1a One entry/one exit	Rust permits early returns (`return`, `?`) and multiple `match` arms; RAII reduces cleanup hazards.	Allow early return for error propagation (`?`). Avoid multiple ad-hoc early returns in complex functions; prefer structured control flow.	C/D: enforce 'structured exits': only `?` and single success return in complex functions unless justified.	Improves readability and supports traceability without reintroducing manual-cleanup patterns.
1b No dynamic objects / online test	Allocation is explicit (`Box`, `Vec`, `String`, `Arc`, `Rc`). `no_std`/no-alloc profiles eliminate heap use entirely.	QM–B: allocation permitted with bounded capacity and failure handling. C/D: prefer no-alloc; otherwise require qualified allocator + fallible allocation patterns.	C/D: forbid unbounded growth; require deterministic allocation strategy and resource usage evaluation.	Controls runtime failure modes and timing jitter due to allocation.
1c Initialization of variables	Rust enforces definite initialization in safe code; `MaybeUninit` exists for controlled patterns.	All variables must be initialized before use. Prohibit use of uninitialized memory APIs; allow `MaybeUninit` only in Q modules with proof of initialization.	C/D: require Miri/UB checking of unsafe initialization paths; document initialization invariants.	Prevents undefined behavior and latent faults.
1d No multiple use of variable names	Rust allows shadowing via `let`; can harm clarity.	Prohibit variable shadowing in safety-related code (`let x = …` reusing names). Allow shadowing only in tiny scopes (e.g., iterator adapters) with explicit lint	C/D: no shadowing (hard error).	Avoids confusion during reviews and reduces mistake risk.

		suppression and justification.		
1e Avoid global variables	Globals exist as `static` and `const`; mutable globals (`static mut`) require unsafe.	Prefer dependency injection and explicit state. Prohibit `static mut`. Restrict `static` to immutable data or to `OnceLock`/`LazyLock` with controlled initialization.	C/D: all global state must be justified, documented, and concurrency-safe; prefer partition-local state.	Global mutable state is a common source of interference and hidden coupling.
1f Restricted use of pointers	References are non-null and lifetime-checked; raw pointers exist but require unsafe to dereference.	Use references (`&T`, `&mut T`) by default. Raw pointers only inside Q modules. Prohibit pointer arithmetic in application code.	C/D: require formal argument and independent review for any raw-pointer deref; ban `offset`/`add` unless proven safe.	Reduces memory safety hazards.
1g No implicit conversions	Rust has no implicit numeric conversions; but has `Deref` coercions and `as` casts.	Restrict `as` to safe, non-lossy cases; require `TryFrom/TryInto` for narrowing. Avoid relying on `Deref` coercions in public APIs (make types explicit).	C/D: ban lossy casts; require dedicated conversion functions with tests for boundary cases.	Prevents truncation/sign errors and clarifies data flow.
1h No hidden data/control flow	No exceptions; but macros, trait objects, and `Drop` can hide effects.	Restrict macros that introduce control flow; require explicit error handling (`Result`). Document `Drop` side effects. Restrict dynamic dispatch (`dyn Trait`) in critical timing paths.	C/D: macro allowlist; no custom proc macros unless qualified; require audit of `Drop` and dynamic dispatch.	Supports comprehension and verification.
1i No unconditional jumps	Rust has no `goto`; `break`/`continue` are structured; labels exist.	Allow structured `break`/`continue`. Restrict labelled `break`/`continue` to cases where they simplify nesting; otherwise refactor.	C/D: require justification for labels; no label-based flow spanning large blocks.	Maintains structured control flow.
1j No recursion (or justify/bound)	Recursion permitted; no guaranteed tail-call optimization.	Prohibit unbounded recursion. Allow bounded recursion only with explicit maximum depth and stack usage analysis.	C/D: bounded recursion only; prefer iterative implementations.	Prevents stack overflow and supports WCET reasoning.

6. ISO 26262-6:2018 Clause 9 — Software Unit Verification (Tables 7–9)

Clause 9 requires evidence that the unit design and implementation satisfy allocated requirements, implement safety measures, and contain neither unintended functionality nor unsafe properties. Rust contributes via strong static guarantees (initialization, borrow checking, data-race freedom in safe code), but verification still requires reviews, static analysis, testing, and coverage evidence.

6.1 ISO 26262-6:2018 Table 7 — Methods for software unit verification mapped to Rust

Method (Table 7)	Rust-aligned implementation	Suggested tools / artifacts	Notes
1a Walk-through	Peer walk-through of design + source (include safety arguments and invariants)	Review minutes, issue tracking, sign-off	Useful early; not sufficient alone for ASIL C/D.
1b Pair-programming	Structured pair work on safety-critical or unsafe sections	Pairing records; reviewer sign-off	Treat as 'enhanced review', not a substitute for formal inspection.
1c Inspection	Checklist-based inspection of code, especially unsafe boundaries	Inspection checklist; defect density metrics	Mandatory intensity increases with ASIL.
1d Semi-formal verification	Formalized reasoning about invariants, contracts, and state machines	Proof sketches, contracts, model checks	Use for concurrency protocols and safety mechanisms.
1e Formal verification	Theorem proving / model checking for highest-risk components	Formal artifacts, assumptions, traceability	Target unsafe abstractions and complex decision logic.
1f Control flow analysis	Control-flow graphs; complexity metrics; unreachable/dead-code checks	Clippy lints; custom CFG tools; coverage reports	Tie to complexity thresholds and Table 6 principles.
1g Data flow analysis	Initialization, aliasing, shared resource access, and value range checks	Static analysis; value-range tools; lint reports	For Rust, include unsafe and FFI boundary checks.
1h Static code analysis	Systematic linting and pattern-based checks	Clippy reports; `#![deny(warnings)]`; MISRA-style rules implemented as lints	Make lints part of CI; qualify tool where required.
1i Abstract interpretation	High-assurance static analyzers for runtime errors, overflow, ranges	Abstract interpretation reports	Especially useful for ASIL C/D where value ranges matter.
1j Requirements-based test	Unit tests derived from requirements; property tests where appropriate	Test specs; test results; traceability	Avoid relying solely on 'happy path'; include boundary cases.
1k Interface test	Unit interface tests validating preconditions, error paths, and contracts	Interface test suites; mocks/harnesses	Critical for FFI and hardware interaction layers.

6.2 ISO 26262-6:2018 Table 8 — Test case derivation methods for unit testing

Rust unit tests should be derived systematically, not ad hoc. Table 8's methods map well to typical Rust testing practices.

Method (Table 8)	Rust testing approach	Concrete guidance
1a Requirements analysis	Write tests that trace to each requirement clause / acceptance criterion	Require requirement IDs in test names or attributes; maintain traceability matrix.
1b Equivalence classes	Partition inputs/outputs; test one representative per class	Use table-driven tests; consider property tests for broad partitions.

1c Boundary values	Test edges: min/max, off-by-one, overflow boundaries	For numeric code, prefer checked arithmetic and test failure modes explicitly.
1d Error guessing	Use ‘lessons learned’ defect patterns to design additional cases	Maintain a bug-pattern checklist; include fuzzing for parsers where applicable.

6.3 ISO 26262-6:2018 Table 9 — Structural coverage metrics at the software unit level

Table 9 requires measurement of unit-level structural coverage. For Rust, coverage tools must account for monomorphization, inlining, and compiler-generated code. For ASIL D, MC/DC is typically required and may need specialized tooling and test design.

Coverage metric	Rust-specific considerations	Evidence / rules
Statement coverage	Compiler inlining can hide statements; use consistent build flags for coverage	Define target thresholds per ASIL; require coverage reports as work products.
Branch coverage	Match exhaustiveness helps but does not replace measurement; guard conditions matter	Require high branch coverage for safety-related units; justify any exclusions.
MC/DC	Boolean expressions in `if`, match guards, and short-circuit operators require careful test design	ASIL D: require MC/DC evidence; restrict complex boolean expressions and prefer named predicates.

7. ISO 26262-6:2018 Clause 10 — Software Integration & Verification (Tables 10–12)

Clause 10 focuses on integrating software units into components and verifying the integrated behavior against the architecture, hardware-software interface, and safety measures. Rust helps enforce interface correctness at compile time, but integration verification must also address timing, resources, and freedom from interference.

7.1 ISO 26262-6:2018 Table 10 — Methods for verification of software integration

Method (Table 10)	Rust-aligned execution	Evidence artifacts	ASIL notes
1a Requirements-based test	Integration tests (`cargo test`) at component boundaries; HSI-level tests for drivers	Integration test specs/results; traceability	++ for all ASIL.
1b Interface test	Contract tests for module/crate interfaces; FFI boundary validation	Interface test suite; negative tests	++ for all ASIL.
1c Fault injection test	Inject corrupted messages/config/inputs; simulate peripheral failures	Fault injection plan/results	+ at A/B; ++ at C/D.
1d Resource usage evaluation	Measure stack/heap/CPU/time usage; detect contention	Resource measurement reports; budgets	++ for all ASIL; must be deterministic at C/D.
1e Back-to-back model/code comparison	If model-based design is used, compare model output to Rust implementation	Comparison results; tolerances	+ at A/B; ++ at C/D.

1f Control/data flow verification	Verify scheduling order, data flow, and mode switching across units	Integration analysis report	+ at A/B; ++ at C/D.
1g Static code analysis	Run linting and static analyzers on integrated builds; check cfg variants	Static analysis reports	++ for all ASIL.
1h Abstract interpretation	Use high-assurance analyzers for integration-level runtime errors	Abstract interpretation report	+ across ASIL; often used for C/D.

7.2 ISO 26262-6:2018 Table 11 — Test case derivation for software integration testing

Use the same Table 8 derivation methods at the integration level, emphasizing interface boundaries and operational modes.

7.3 ISO 26262-6:2018 Table 12 — Structural coverage at the software architectural level

Architectural-level structural coverage focuses on functions/components being executed and call paths being exercised. For Rust, generic monomorphization means ‘function coverage’ must consider the relevant instantiations that appear in the final binary.

Coverage metric (Table 12)	Rust considerations	Evidence / rules
Function coverage	Count executed functions in final binary; be careful with inlining and generics	Set target thresholds per ASIL; justify excluded code (debug/instrumentation).
Call coverage	Exercise call sites, including trait dispatch paths (static and dyn)	Require call-path tests for safety mechanisms and error handling paths.

8. ISO 26262-6:2018 Clause 11 — Testing of Embedded Software (Tables 13–15)

Clause 11 validates the integrated embedded software in the target environment. While not Rust-specific, the coding standard should anticipate how Rust artifacts are exercised in HIL rigs, ECU networks, and vehicle tests.

8.1 ISO 26262-6:2018 Table 13 — Test environments

Environment	Rust considerations	Guidance
Hardware-in-the-loop (HIL)	Ensure build reproducibility and deterministic configuration; instrument safely for coverage	Use qualified build toolchain; separate test instrumentation from production builds.
ECU network environments	Verify end-to-end timing, message ordering, and fault handling; test coexistence/partitioning	Run integration tests under realistic bus load; include fault injection on buses and sensors.
Vehicles	Validate operational modes, degraded modes, and safe-state behavior	Ensure test cases cover operational use cases and safety mechanisms under real timing.

8.2 ISO 26262-6:2018 Table 14 — Methods for tests of embedded software

Requirements-based tests remain primary. For higher ASIL, fault injection at the software level is used to validate safety mechanisms (e.g., corrupted configuration/calibration parameters, injected communication errors).

8.3 ISO 26262-6:2018 Table 15 — Test case derivation for embedded software testing

Table 15 extends unit/integration derivation with functional dependency analysis and operational use cases. For Rust, explicitly include: boot/integrity checks, mode transitions, update mechanisms, and any HAL/driver edge cases.

9. Configurable Software (Annex C) and Rust Configuration Practices

Annex C addresses configuration and calibration data that influence embedded software behavior. Rust supports robust configuration handling through strong typing (newtypes), explicit parsing with validation, and compile-time configuration (feature flags, const generics) — but configurable behavior increases verification scope.

- Treat configuration/calibration inputs as untrusted: validate ranges, cross-field constraints, and units at the boundary.
- Represent configuration with typed structs and newtypes; avoid 'stringly typed' key-value access in safety-related code.
- Prefer compile-time configuration for safety-critical variability when feasible (const generics, build-time features) to reduce runtime states — but control `#[cfg]` carefully to avoid untested variants.
- Maintain separate safety cases and verification evidence for each configuration set intended for production release.
- For ASIL C/D: configuration parsing and application logic should be in a dedicated module with high test coverage and boundary-value tests; consider formalizing constraints (contracts) for critical parameters.

10. Rust Language Inventory — Complete Construct Classification (QM to ASIL D)

This section provides a complete, auditable inventory of Rust language constructs as they appear in the source language. The intent is to classify every construct for use in a safety-related coding standard. Unstable/nightly features are treated as U (Unavailable) and therefore X for certified builds.

10.1 Global language-profile rules

- Certified builds shall use stable Rust only (no `#![feature(...)]`).
- Edition shall be pinned in Cargo.toml (this baseline assumes Edition 2024).
- Application crates for ASIL code shall use `#![forbid(unsafe_code)]`. Unsafe is allowed only in explicitly named Q/TCB crates.
- Panics shall not be used for expected errors. For ASIL C/D, `panic=abort` is required to avoid unwinding across boundaries.
- All public safety-related APIs shall have explicit lifetimes (no lifetime elision) and explicit error semantics.

10.2 Keywords and lexical constructs (complete list)

Rust keywords are categorized as strict, reserved, and weak. Reserved keywords are not currently used but are prohibited as identifiers (except as raw identifiers). For safety-critical readability, use of raw identifiers (e.g., `r#try`) is restricted.

Keyword / construct	QM	A	B	C	D	Notes / restrictions
_	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
as	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
async	P	R	R	R	X	Async/await is allowed only with a qualified runtime and a defined cancellation/pinning policy; prohibited by default at ASIL D.
await	P	R	R	R	X	Async/await is allowed only with a qualified runtime and a defined cancellation/pinning policy; prohibited by default at ASIL D.
break	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
const	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
continue	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
crate	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
dyn	P	P	R	R	R	Trait objects introduce dynamic dispatch; restrict in timing-critical code; justify use over generics.

else	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
enum	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
extern	R	R	R	Q	Q	FFI boundary: only in qualified wrappers; require input validation and no unwinding across FFI.
false	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
fn	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
for	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
if	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
impl	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
in	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
let	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
loop	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
match	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
mod	P	P	P	P	P	Fundamental language construct; apply

						category-specific rules elsewhere.
move	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
mut	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
pub	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
ref	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
return	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
self	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
Self	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
static	P	R	R	R	R	Immutables allowed; `static mut` prohibited. Prefer `OnceLock`/`LazyLock` with controlled init.
struct	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
super	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
trait	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
true	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
type	P	P	P	P	P	Fundamental

						language construct; apply category-specific rules elsewhere.
unsafe	R	R	R	Q	Q	Unsafe boundary: allowed only in Q/TCB crates at ASIL C/D; forbid in ASIL application crates.
use	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
where	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
while	P	P	P	P	P	Fundamental language construct; apply category-specific rules elsewhere.
abstract	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
become	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
box	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
do	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
final	U	U	U	U	U	Reserved for future use;

						cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
gen	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
macro	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
override	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
priv	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
try	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
typeof	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
unsized	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw

						identifiers (e.g., r#try) is restricted (see below).
virtual	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
yield	U	U	U	U	U	Reserved for future use; cannot be used as identifier. Using raw identifiers (e.g., r#try) is restricted (see below).
'static	P	P	P	P	P	Static lifetime permitted; avoid overusing `'static` bounds that hide ownership issues.
macro_rules	P	R	R	R	R	Declarative macros restricted: allowlist only; expansions must not hide control flow.
raw	R	R	R	Q	Q	Raw borrow operators and raw pointers are unsafe-adjacent; restrict to Q/TCB.
safe	U	U	U	U	U	Contextual keyword in external blocks; avoid relying on experimental syntax for certified code.
union	R	R	R	Q	Q	Unions allowed only for FFI; field access is unsafe; require safe wrapper enum.
Raw identifiers (r#ident)	P	R	R	R	R	Allowed only when interfacing with external APIs requiring keyword names; otherwise prohibited for readability.
Edition-2024 reserved syntax (#"..."#,	U	U	U	U	U	Reserved syntax in Rust 2024 edition; avoid in

ident##"..."##)						certified code to prevent forward-compat surprises.

10.3 Items, modules, and visibility

Rust’s item system (modules, structs, enums, traits, impls) and visibility controls are the primary architectural tools for enforcing encapsulation and limiting interface size.

Items and visibility

Construct / feature	**QM**	**A**	**B**	**C**	**D**	**Restrictions / notes**
Modules (`mod`)	P	P	P	P	P	Use modules to enforce encapsulation and layering; require module-level docs for safety-related modules.
Imports (`use`)	P	P	P	P	P	Prohibit wildcard imports (`use foo::*`) in ASIL code; prefer explicit paths for traceability.
Public visibility (`pub`)	P	R	R	R	R	Minimize public API surface; all `pub` items require documentation and review.
Restricted visibility (`pub(crate)`, `pub(super)`, `pub(in path)`)	HR	HR	HR	HR	HR	Preferred over `pub` to support encapsulation.
Constants (`const` items)	P	P	P	P	P	Prefer `const` for pure values; avoid large computed const expressions unless toolchain supports.
Statics (`static` immutable)	P	R	R	R	R	Allowed for immutable data; must be justified if used for shared resources.
Mutable statics (`static mut`)	X	X	X	X	X	Prohibited; use safe synchronization (`Mutex`, atomics) or partitioning.
Functions (`fn`)	P	P	P	P	P	Apply complexity limits and error-handling policy; prefer explicit

						`Result`.
Traits (`trait`)	P	P	P	P	P	Document trait contracts and invariants; restrict deep hierarchies.
Trait objects (`dyn Trait`)	P	P	R	R	R	Dynamic dispatch requires justification; restrict in timing-critical paths.
Implementations (`impl`)	P	P	P	P	P	Prefer inherent methods for cohesion; avoid orphan impls in application code.
Type aliases (`type`)	P	P	P	P	P	Use for readability only; prefer newtypes for type safety.
Structs (`struct`)	P	P	P	P	P	Preferred for domain modeling; document invariants; use private fields.
Enums (`enum`)	HR	HR	HR	HR	HR	Prefer to encode state machines; require exhaustive matching; avoid wildcard arms for evolving enums.
Unions (`union`)	R	R	R	Q	Q	Allowed only for FFI; must be wrapped and audited; unsafe field access.
Extern blocks (`extern "C" { ... } `)	R	R	R	Q	Q	FFI allowed only in qualified boundary modules; validate all inputs; forbid unwinding across boundary.
Unsafe functions (`unsafe fn`)	R	R	R	Q	Q	Allowed only where preconditions cannot be expressed; require documented SAFETY contract.
Unsafe traits/impls	R	R	R	Q	Q	Allowed only with explicit invariant proof; independent review at ASIL C/D.
Macro definitions (`macro_rules!`)	P	R	R	R	R	Allowlist only; expansions must be reviewed;

						avoid hidden control flow.
Procedural macros	R	R	R	Q	Q	Allowed only if macro crate is qualified and generated code is reviewable; prefer explicit code.

10.4 Types, generics, and lifetimes

Rust’s type system is a key safety mechanism. Restrictions primarily target platform-dependent types, unsafe-adjacent representations, and constructs that complicate reasoning (excessive generics, advanced lifetime features) without clear benefit.

Type-system features

Construct / feature	QM	A	B	C	D	Restrictions / notes
Fixed-width integers (i8…i128, u8…u128)	P	P	P	P	P	Prefer fixed-width for interfaces; define explicit overflow strategy (checked/saturating/wrapping).
Platform-sized ints (isize/usize)	P	R	R	R	R	Use for indexing and pointer-sized values only; avoid for domain quantities crossing interfaces.
Floating point (f32/f64)	P	R	R	R	R	Allowed with explicit handling of NaN/Inf and rounding; avoid in safety-critical control unless justified.
Experimental floats (f16/f128)	U	U	U	U	U	Experimental in Rust 1.93.1 docs; not allowed in certified builds.
Bool/char	P	P	P	P	P	Permitted; for char/str ensure UTF-8 assumptions match domain needs.
Never type (`!`)	R	R	R	R	R	Used for diverging functions (panic/loop). Restrict to panic handlers and deliberate diverging APIs.
Arrays `[T; N]`	HR	HR	HR	HR	HR	Preferred for

						fixed-size buffers; supports bounds safety and determinism.
Slices `[T]` / `&[T]`	P	P	P	P	P	Prefer iterator access; restrict indexing that can panic; use `get()` for checked access.
Tuples	P	R	R	R	R	Allow small tuples (2–3) for returns; prefer named structs for larger tuples.
References `&T`, `&mut T`	HR	HR	HR	HR	HR	Default pointer-like mechanism; require explicit lifetimes in public APIs at ASIL C/D.
Raw pointers `*const T`, `*mut T`	R	R	R	Q	Q	Allowed only in qualified unsafe code; no arithmetic/deref in application code.
Function pointers `fn(...) -> ...`	P	P	P	P	P	Permitted; document calling conventions and lifetimes; be careful with callbacks crossing FFI.
Trait bounds / generics	P	P	P	P	P	Limit generic complexity; prefer associated types over many parameters.
Higher-ranked trait bounds (`for<'a>`)	R	R	R	R	R	Powerful but complex; restrict to library/TCB code with justification.
Const generics	P	P	P	P	P	Encourage for fixed-capacity buffers and dimensioned arrays; verify const expressions remain simple.
Lifetime elision	P	R	R	R	R	Allow in internal functions; prohibit in public safety-related APIs to improve auditability.
UnsafeCell / interior mutability primitives	R	R	R	Q	Q	Direct `UnsafeCell` usage only in qualified synchronization

						primitives; prefer safe wrappers.
Auto traits (Send/Sync/Unpin)	P	P	P	P	P	Auto-implemented is preferred; manual impls are unsafe and therefore Q.

10.5 Control flow, patterns, and error propagation

Structured control flow supports verification. Rust lacks `goto`, but macros and divergence can still obscure flow. Error handling should be explicit (`Result`/`Option`), and panics should be reserved for unreachable invariant violations.

Control flow and error propagation

Construct / feature	**QM**	**A**	**B**	**C**	**D**	**Restrictions / notes**
`if` / `else`	P	P	P	P	P	Standard conditional; keep boolean expressions simple for MC/DC.
`match` (exhaustive)	HR	HR	HR	HR	HR	Prefer exhaustive matching over if-chains; avoid wildcard arms for enums (esp. C/D).
`if let` / `while let`	P	P	P	P	P	Use for single-pattern matches; still consider exhaustiveness for enums.
`let ... else`	P	P	P	P	P	Permitted; use primarily for early error exits with clear rationale.
Loops (`for`)	HR	HR	HR	HR	HR	Preferred iteration form; avoid manual indexing where possible.
Loops (`while`/`loop`)	P	P	P	P	P	Allowed; require documented termination condition and bounds for complex loops.
`break`/ `continue`	P	P	P	P	P	Permitted; keep scope local; prefer structured refactoring over complex jumps.
Labelled breaks/continues	P	R	R	R	R	Restrict to cases that reduce complexity;

						otherwise refactor.
Early `return`	P	P	P	R	R	Allowed for error paths and simple guards; restrict in complex functions at C/D.
`?` operator	HR	HR	HR	HR	HR	Preferred for propagating `Result`/`Option`; ensure error types are explicit.
Divergence (`-> !`, `panic!`, infinite loops)	R	R	R	R	R	Allowed only for panic handlers, fatal invariants, and main loops; document rationale.
Recursion	P	R	R	R	R	Only bounded recursion allowed with explicit depth and stack analysis (C/D).
Async/await suspension points	P	R	R	R	X	See keyword table: requires qualified runtime and cancellation policy; prohibited by default at D.

10.6 Attributes, compiler configuration, and build directives

Attributes and build directives can significantly affect compiled behavior and verification scope. Safety profiles must restrict conditional compilation, representation attributes, and any attribute that changes linkage or memory layout.

Attributes and configuration

Construct / feature	QM	A	B	C	D	Restrictions / notes
Lint levels (`#![deny(warnings)]`, `#![forbid(unsafe_code)]`)	HR	HR	HR	HR	HR	Make deny/forbid policies part of CI; define allowlist of permitted `#[allow(...)]` suppressions.
Conditional compilation (`#[cfg]`, `cfg!`, `cfg_attr`)	P	R	R	R	R	Minimize cfg variants; require test evidence for each enabled configuration; document all feature flags.
Derive (`#[derive(...)]`)	P	P	P	P	P	Allow only known derives by default;

						custom derive/proc macros require qualification.
Representation (`#[repr(C)]`, `#[repr(transparent)]`)	R	R	R	Q	Q	Allowed for FFI and ABI stability only; must be reviewed; prefer explicit layout tests.
Representation (`#[repr(packed)]`)	X	X	X	X	X	Prohibited: can create unaligned references and UB; use byte arrays and parsing.
Alignment (`#[repr(align(N))]`)	R	R	R	R	R	Allow only with justification; affects memory footprint and may hide assumptions.
Inlining hints (`#[inline]`, `#[inline(always)]`, `#[inline(never)]`)	P	R	R	R	R	Allow sparingly; `always` can harm timing analysis and code size; document when used.
Linkage (`#[no_mangle]`, `#[export_name]`)	R	R	R	Q	Q	FFI/ABI surface only; require controlled symbol naming and versioning.
Section placement (`#[link_section]`)	R	R	R	Q	Q	Only in qualified low-level modules; document memory map assumptions.
`#[used]` / `#[used(linker)]`	R	R	R	Q	Q	Only for linker/interrupt vectors in low-level code; require review.
`#![no_std]`	P	P	P	HR	HR	Preferred for ASIL C/D embedded targets; ensures OS-free determinism.
`#![no_main]` / `#[panic_handler]`	R	R	R	R	R	Used in bare-metal; require validated startup and panic behavior.
Global allocator (`#[global_allocator]`)	R	R	R	Q	Q	Allocator must be qualified; for ASIL D prefer no-alloc profile.
Nightly feature gates (`#![feature(...)]`)	U	U	U	U	U	Not allowed in certified builds.
Testing attrs (`#[test]`,	P	P	P	P	P	Allowed in test builds only;

`#[should_panic]`)						production code must not depend on test-only behavior.

10.7 Macros and compile-time code generation

Macros can hide control flow and data flow, impacting reviewability (Table 6 principle 1h). This mapping therefore uses an allowlist approach: standard macros are classified; custom macros are restricted.

Macro usage

Construct / feature	**QM**	**A**	**B**	**C**	**D**	**Restrictions / notes**
Built-in assert macros (`assert!`, `assert_eq!`, `assert_ne!`)	P	R	R	R	R	Permitted for invariant checks; must not be used for expected errors; define policy for production builds.
Debug-only asserts (`debug_assert!` etc.)	P	R	R	R	R	Allowed only if build system ensures release builds do not rely on them for safety.
Panic macros (`panic!`, `todo!`, `unimplemented!`, `unreachable!`)	R	R	R	R	R	Only for unrecoverable invariants; `todo!/unimplemented!` prohibited in released software.
Formatting/IO macros (`println!`, `eprintln!`, `dbg!`)	P	R	R	R	R	Prohibit in production safety code unless routed through a qualified logging/diagnostic layer.
Collection macros (`vec!`)	P	P	P	R	R	Allocates; subject to heap-allocation policy.
Include macros (`include!`, `include_bytes!`, `include_str!`)	P	R	R	R	R	Increases build-time coupling; require justification and traceability; validate included data.
Declarative macros (`macro_rules!`)	P	R	R	R	R	Allowlist only; require documentation of expansion and prohibit hidden early returns.
Procedural macros	R	R	R	Q	Q	Only if macro crate is qualified;

(derive/attribute /function-like)						generated code must be reviewable and stable.

10.8 Unsafe Rust and undefined behavior boundaries

Unsafe Rust is the Trusted Computing Base (TCB) of a Rust safety argument. All unsafe usage must be minimized, isolated, and justified with documented invariants. For ASIL C/D, unsafe code is permitted only inside explicitly named Q modules with enhanced review and (where feasible) formal reasoning.

Unsafe operations

Construct / feature	**QM**	**A**	**B**	**C**	**D**	**Restrictions / notes**
`unsafe` blocks	R	R	R	Q	Q	Every unsafe block requires a `// SAFETY:` justification describing invariants; keep blocks minimal.
Raw pointer dereference	R	R	R	Q	Q	Prohibited in application code; allowed in Q code with proof of validity/alignment/lifetime.
Pointer arithmetic (`add`, `offset`)	R	R	R	Q	Q	Only in Q code; must prove bounds and avoid overflow.
Unions and union field access	R	R	R	Q	Q	Only for FFI; wrap in safe enum; avoid type punning.
Inline assembly (`asm!`)	R	R	R	Q	Q	Only in qualified low-level modules; require architecture-specific review and tests.
Volatile access (`read_volatile`, `write_volatile`)	R	R	R	Q	Q	Hardware access only; document memory-mapped IO assumptions.
Transmute / type punning	X	X	X	X	X	Prohibited. Use safe conversion APIs or explicit byte-level parsing.
Uninitialized memory patterns (`MaybeUninit`)	R	R	R	Q	Q	Allowed only with explicit initialization proof; require tests and UB checking.
FFI boundaries (`extern "C"` ABI)	R	R	R	Q	Q	Validate all inputs; no unwinding;

						define ownership and lifetime conventions.
Manual Send/Sync impls	X	X	X	Q	Q	Treat as unsafe: only in qualified libraries with proof and independent review.

11. Standard Library Inventory — core/alloc/std Modules and Macros (QM to ASIL D)

This section enumerates the complete module and macro surface of Rust 1.93.1's `core`, `alloc`, and `std` crates, and classifies each area. For certified code, items marked Experimental/unstable are treated as U (Unavailable) and are not permitted.

Notation for the 'Profile' column below:
• The profile is written as QM/A/B/C/D.
• Example: `P/P/R/R/R` means Permitted at QM and ASIL A, Restricted at ASIL B–D.

11.1 `core` crate module inventory (Rust 1.93.1)

Module	**Purpose / risk area**	**Profile (QM/A/B/C/D)**	**Notes / key restrictions**
core::alloc	Layout and allocation-related types (even in no_std)	P/P/P/P/P	Permitted; includes `Layout`, `GlobalAlloc`-related interfaces; treat allocator implementation as Q.
core::any	Type reflection (TypeId) and downcasting	P/P/R/R/R	Restrict in ASIL C/D to avoid dynamic-type complexity; prefer static typing.
core::arch	SIMD/vendor intrinsics	R/R/R/Q/Q	Hardware-specific; allow only in qualified, reviewed low-level modules; avoid if WCET determinism impacted.
core::array	Array utilities	P/P/P/P/P	Permitted; prefer fixed-size arrays for determinism.
core::ascii	ASCII operations	P/P/P/P/P	Permitted; ensure encoding assumptions are documented.
core::borrow	Borrowed data utilities (Cow, Borrow)	P/P/P/P/P	Permitted; beware of Cow allocations when paired with alloc.
core::cell	Interior mutability (Cell/RefCell/UnsafeCell)	P/R/R/R/R	Restrict; avoid RefCell (runtime panics). UnsafeCell only in Q primitives.
core::char	char utilities	P/P/P/P/P	Permitted.
core::clone	Clone trait	P/P/P/P/P	Permitted; review manual Clone implementations.
core::cmp	Comparison and ordering	P/P/P/P/P	Permitted.
core::convert	From/Into/TryFrom/TryInto	P/P/P/P/P	Permitted; require TryFrom for fallible/narrowing conversions.
core::default	Default trait	P/P/P/P/P	Permitted; Default must represent valid state.
core::error	Error trait	P/P/P/P/P	Permitted; define project error taxonomy.
core::f32 / core::f64	Float constants	P/R/R/R/R	Restrict floating point usage in safety-critical control unless justified.

core::ffi	C-compatible primitive types	R/R/R/Q/Q	FFI boundary only; validate conversions and layouts.
core::fmt	Formatting traits	P/P/P/P/P	Permitted; avoid panics in fmt; restrict formatting in timing-critical paths.
core::future / core::task	Async primitives	P/R/R/R/X	Async requires qualified runtime/policy; default prohibit at ASIL D.
core::hash	Hashing support	P/P/R/R/R	Hash-based behavior may be nondeterministic; restrict in high ASIL unless determinism ensured.
core::hint	Compiler hints	P/R/R/R/R	Allow sparingly; document any reliance on optimization behavior.
core::iter	Iterator trait	HR/HR/HR/HR/HR	Preferred iteration model.
core::marker	Marker traits	P/P/P/P/P	Permitted; manual impls of Send/Sync are Q.
core::mem	Memory utilities (includes some unsafe-adjacent APIs)	P/R/R/R/R	Allow safe subset (size_of, replace); prohibit transmute/zeroed; unsafe parts only in Q.
core::net	Socket address types	P/P/R/R/R	Use only if network stack exists and is qualified; otherwise avoid in embedded.
core::num	Numeric wrappers (NonZero, Wrapping, Saturating)	P/P/P/P/P	Encourage explicit overflow semantics.
core::ops	Operator traits	P/P/P/P/P	Permitted with restrictions: operator overloading must preserve semantics.
core::option / core::result	Option/Result	M/M/M/M/M	Mandatory for optional and fallible operations.
core::panic	Panic support	R/R/R/R/R	Define panic policy; for ASIL C/D require panic=abort.
core::pin	Pinning	R/R/R/R/R	Restrict to advanced code; direct Pin manipulation only in qualified libraries.
core::prelude / core::primitive	Prelude and primitive reexports	P/P/P/P/P	Permitted; follow visibility and API rules.
core::ptr	Raw pointer operations	R/R/R/Q/Q	Unsafe-adjacent; application code forbidden; Q only with proofs.
core::slice / core::str	Slice and string utilities	P/P/P/P/P	Prefer safe accessors; restrict unchecked indexing.
core::sync	Core sync primitives	P/R/R/R/R	Mostly atomic and fence primitives; restrict memory ordering; require documentation.
core::time	Duration and time utilities	P/P/P/P/P	Permitted; actual clocks are in std.
core::{i8,i16,i32,i64,i128,isize,u8,u16, u32,u64,u128,usize} (const modules)	Redundant constants modules (deprecation planned)	P/P/P/P/P	Avoid in new code; prefer inherent constants (e.g., i32::MAX).
core::assert_matches	Experimental macro module	U/U/U/U/U	Unstable; not allowed in certified builds.
core::async_iter	Experimental async iteration	U/U/U/U/U	Unstable.
core::autodiff	Experimental automatic differentiation	U/U/U/U/U	Unstable.
core::bstr	Experimental byte string types	U/U/U/U/U	Unstable.
core::contracts	Experimental contracts / attribute macros	U/U/U/U/U	Unstable.
core::f16 / core::f128 (const modules)	Experimental float constants	U/U/U/U/U	Unstable.
core::from	Experimental From derive	U/U/U/U/U	Unstable.

	macro		
core::index	Experimental slice indexing helpers	U/U/U/U/U	Unstable.
core::intrinsics	Compiler intrinsics	U/U/U/U/U	Unstable; prohibited.
core::io / core::os / core::panicking	Experimental low-level support modules	U/U/U/U/U	Unstable; prohibited.
core::pat	Experimental pattern_type macro helpers	U/U/U/U/U	Unstable.
core::profiling	Experimental profiling markers	U/U/U/U/U	Unstable.
core::random	Experimental random generation	U/U/U/U/U	Unstable.
core::range	Experimental range types	U/U/U/U/U	Unstable.
core::simd	Experimental portable SIMD	U/U/U/U/U	Unstable; prohibit in certified builds.
core::ub_checks	Experimental unsafe precondition checks	U/U/U/U/U	Unstable.
core::unsafe_binder	Experimental unsafe binder operators	U/U/U/U/U	Unstable.

11.2 `alloc` crate module inventory (Rust 1.93.1)

Module	Purpose / risk area	Profile (QM/A/B/C/D)	Notes / key restrictions
alloc::alloc	Allocator interface and `Layout`	P/R/R/Q/Q	Allocator itself must be qualified; prefer no-alloc profile at ASIL D.
alloc::borrow	Borrowed data utilities (Cow)	P/P/P/R/R	May allocate when converting to owned; ensure bounded behavior.
alloc::boxed	Box heap allocation	P/P/P/R/R	Heap allocation subject to policy; require fallible allocation patterns where possible.
alloc::collections	Heap-backed collections	P/P/P/R/R	Restrict unbounded growth; prefer fixed-capacity alternatives for C/D.
alloc::ffi	FFI utilities (CString etc)	R/R/R/Q/Q	FFI boundary only; validate encoding and lengths.
alloc::fmt	Formatting for `String`	P/P/P/R/R	Formatting allocates; restrict in timing-critical paths.
alloc::rc	Single-threaded reference counting	P/P/R/R/R	Restrict due to complex ownership and potential cycles; avoid in safety-critical components.
alloc::slice / alloc::str	Utilities for slice/str	P/P/P/P/P	Permitted; allocations depend on method used.
alloc::string	Owned String	P/P/P/R/R	Heap allocation; ensure bounded size and validated inputs.
alloc::sync	Arc and related	P/P/R/R/R	Restrict to controlled concurrency patterns; avoid cycles with Weak.
alloc::task	Async task types	P/R/R/R/X	Async requires qualified runtime; prohibited by default at ASIL D.
alloc::vec	Vec heap allocation	P/P/P/R/R	Require capacity limits and no unbounded growth; prefer fallible allocation.
alloc::bstr (Experimental)	ByteStr/ByteString	U/U/U/U/U	Experimental; not allowed in certified builds.

11.3 `std` crate module inventory (Rust 1.93.1)

Module	Purpose / risk area	Profile (QM/A/B/C/D)	Notes / key restrictions
std::alloc	Allocator interfaces	P/R/R/Q/Q	Allocator must be qualified; prefer no-alloc for ASIL D.
std::any	Dynamic typing/reflection	P/P/R/R/R	Restrict for C/D; avoid runtime type dispatch in safety code.
std::arch	SIMD/vendor intrinsics	R/R/R/Q/Q	Hardware-specific; only in qualified modules with timing review.
std::array	Array utilities	P/P/P/P/P	Permitted.
std::ascii	ASCII utilities	P/P/P/P/P	Permitted.
std::backtrace	Backtrace capture	P/R/R/R/R	Diagnostics only; avoid in production safety paths; may allocate and depend on OS.
std::borrow	Borrow utilities	P/P/P/P/P	Permitted; watch for hidden allocations via Cow.
std::boxed	Box	P/P/P/R/R	Heap allocation policy applies.
std::cell	Interior mutability	P/R/R/R/R	Restrict; avoid runtime borrow panics.
std::char	Char utilities	P/P/P/P/P	Permitted.
std::clone	Clone trait	P/P/P/P/P	Permitted.
std::cmp	Comparisons	P/P/P/P/P	Permitted.
std::collections	Collections	P/P/P/R/R	Restrict unbounded growth; prefer deterministic maps (BTree*) for higher ASIL.
std::convert	Conversions	P/P/P/P/P	Require TryFrom for narrowing.
std::default	Default	P/P/P/P/P	Default must be valid state.
std::env	Environment variables	P/R/R/R/R	Global mutable state; restrict to initialization/config only.
std::error	Error traits	P/P/P/P/P	Permitted.
std::f32 / std::f64	Float constants	P/R/R/R/R	Restrict floating point usage unless justified.
std::ffi	FFI utilities	R/R/R/Q/Q	Qualified boundary only; validate all conversions.
std::fmt	Formatting	P/P/P/R/R	Formatting may allocate; restrict in timing-critical paths.
std::fs	Filesystem	P/R/R/R/R	Domain-dependent; allowed only in qualified platform components.
std::future	Async basics	P/R/R/R/X	Requires qualified runtime; prohibited by default at ASIL D.
std::hash	Hashing	P/P/R/R/R	Hash collections may be nondeterministic; prefer ordered collections for C/D.
std::hint	Compiler hints	P/R/R/R/R	Allow sparingly; document usage.
std::{i8,i16,i32,i64,i128,isize,u8,u16, u32,u64,u128,usize} (const modules)	Redundant constants (deprecation planned)	P/P/P/P/P	Avoid in new code; prefer inherent constants (e.g., i32::MAX).
std::io	I/O traits and helpers	P/R/R/R/R	Domain-dependent; all I/O is fallible; restrict to qualified components.
std::iter	Iterators	HR/HR/HR/HR/HR	Preferred iteration.
std::marker	Marker traits	P/P/P/P/P	Permitted.
std::mem	Memory utilities	P/R/R/R/R	Permit safe subset; unsafe-adjacent APIs only in Q.
std::net	Networking	P/R/R/R/R	Domain-dependent; typically restricted in ECU software.
std::num	Numeric utilities	P/P/P/P/P	Encourage explicit overflow handling.
std::ops	Operator traits	P/P/P/P/P	Operator overloading restricted by semantics.

std::option / std::result	Option/Result	M/M/M/M/M	Mandatory for optional/fallible operations.
std::os	OS-specific APIs	P/R/R/R/R	Qualified platform layer only; avoid in portable safety components.
std::panic	Panic support	R/R/R/R/R	Define panic policy; avoid panics in normal control flow.
std::path	Paths	P/R/R/R/R	Path parsing/normalization can be complex; restrict to platform layer.
std::pin	Pinning	R/R/R/R/R	Restrict to qualified async or self-referential code.
std::prelude / std::primitive	Prelude and primitive reexports	P/P/P/P/P	Permitted.
std::process	Process control	P/R/R/R/R	Rare in ECU; restrict to tooling or qualified diagnostics.
std::ptr	Raw pointer operations	R/R/R/Q/Q	Unsafe-adjacent; Q only.
std::rc	Rc/Weak	P/P/R/R/R	Restrict due to complex lifetimes and cycles; avoid in safety-critical.
std::slice / std::str	Slice/str utilities	P/P/P/P/P	Prefer checked access.
std::string	String	P/P/P/R/R	Heap allocation; restrict size and avoid in timing-critical code.
std::sync	Synchronization primitives	P/P/R/R/R	Deadlocks not prevented by type system; define lock order and bounded blocking.
std::task	Async task types	P/R/R/R/X	Depends on async policy.
std::thread	OS threads	P/R/R/R/R	Restrict; define thread creation policy; avoid unbounded threads.
std::time	Time measurement	P/P/P/P/P	Prefer monotonic `Instant` for timing; wall-clock for logging only.
std::vec	Vec	P/P/P/R/R	Heap allocation policy applies; bounded capacity required at C/D.
std::assert_matches (Experimental)	Unstable assert_matches macro	U/U/U/U/U	Unstable; not allowed in certified builds.
std::async_iter (Experimental)	Async iterators	U/U/U/U/U	Unstable.
std::autodiff (Experimental)	Automatic differentiation	U/U/U/U/U	Unstable.
std::bstr (Experimental)	Byte strings	U/U/U/U/U	Unstable.
std::f16 / std::f128 (Experimental)	Experimental float constants	U/U/U/U/U	Unstable.
std::from (Experimental)	Unstable From derive	U/U/U/U/U	Unstable.
std::intrinsics (Experimental)	Compiler intrinsics	U/U/U/U/U	Unstable.
std::pat (Experimental)	pattern_type helpers	U/U/U/U/U	Unstable.
std::random (Experimental)	Random generation	U/U/U/U/U	Unstable; prohibit.
std::range (Experimental)	Replacement range types	U/U/U/U/U	Unstable.
std::simd (Experimental)	Portable SIMD	U/U/U/U/U	Unstable.
std::unsafe_binder (Experimental)	Unsafe binder operators	U/U/U/U/U	Unstable.

11.4 Standard macros inventory (selected classification-critical)

Standard macros are imported by default and therefore must be explicitly classified. The list below covers the macros shown in Rust 1.93.1 standard library documentation. Macros not listed here are treated as prohibited for ASIL code by the default-deny rule.

Macro	**Purpose**	**Profile (QM/A/B/C/D)**	**Notes / key restrictions**
assert!	Runtime assertion	P/R/R/R/R	Allowed for internal invariants; must not replace error handling.
assert_eq!	Equality assertion	P/R/R/R/R	Same as assert!; avoid

			expensive formatting in timing-critical code.
assert_ne!	Inequality assertion	P/R/R/R/R	Same as assert!.
cfg!	Compile-time cfg evaluation	P/R/R/R/R	Conditional compilation increases verification scope; minimize.
column!	Source location	P/P/P/P/P	Permitted; primarily for diagnostics.
compile_error!	Forced compile error	P/P/P/P/P	Permitted for enforcing configuration constraints.
concat!	Concatenate literals	P/P/P/P/P	Permitted.
dbg!	Debug printing	P/R/R/R/R	Prohibit in production; allow in dev/test.
debug_assert! / _eq! / _ne!	Debug-only assertions	P/R/R/R/R	Safety must not depend on debug-only checks; define build policy.
env!	Compile-time environment variable	P/R/R/R/R	Restrict: introduces build-environment coupling; document usage.
eprint! / eprintln!	stderr printing	P/R/R/R/R	Restrict to diagnostics layer.
file! / line! / module_path!	Source location	P/P/P/P/P	Permitted.
format! / format_args!	String formatting	P/P/P/R/R	May allocate; restrict in timing-critical paths; avoid in ISR/RT loops.
include! / include_bytes! / include_str!	Include file at compile time	P/R/R/R/R	Restricted: traceability and integrity of included artifacts required.
is_x86_feature_detected!	CPU feature detection	R/R/R/Q/Q	Platform-specific; only in qualified modules.
matches!	Pattern match predicate	P/P/P/P/P	Permitted.
option_env!	Optional compile-time env var	P/R/R/R/R	Same restrictions as env!.
panic!	Panic	R/R/R/R/R	Only for unrecoverable invariants; C/D require panic=abort.
print! / println!	stdout printing	P/R/R/R/R	Restrict to diagnostics layer; avoid in safety logic.
stringify!	Stringify tokens	P/P/P/P/P	Permitted.
thread_local!	Thread-local storage key	P/R/R/R/R	Restrict due to hidden global state and initialization order.
todo! / unimplemented!	Placeholder panic	X/X/X/X/X	Prohibited in released software.
unreachable!	Unreachable code marker (panics)	R/R/R/R/R	Only with proof; avoid as logic substitute.
vec!	Vec literal (allocates)	P/P/P/R/R	Subject to heap policy.
write! / writeln!	Formatted write to buffer	P/P/P/R/R	May allocate depending on target; prefer preallocated buffers.
try! (deprecated)	Error propagation macro	U/U/U/U/U	Deprecated; use `?`.
cfg_select! (experimental)	Experimental cfg selection	U/U/U/U/U	Unstable; not allowed.
concat_bytes! (experimental)	Experimental bytes concat	U/U/U/U/U	Unstable.
const_format_args! (experimental)	Experimental const formatting	U/U/U/U/U	Unstable.
log_syntax! (experimental)	Macro debugging	U/U/U/U/U	Unstable.
trace_macros! (experimental)	Macro tracing	U/U/U/U/U	Unstable.

11.5 Method-level rules for the standard library

Listing every method of every standard library type inside this document would be impractical, but certification still requires method-level decisions. This mapping therefore uses (1) a method hazard taxonomy that deterministically classifies methods, and (2) an explicit list of high-risk exceptions that are always restricted/prohibited.

11.5.1 Method hazard taxonomy (deterministic classification)

Panicking methods: any method documented to panic (including indexing via `Index`/`IndexMut`) is R for QM–B and X for ASIL C/D, unless proven unreachable and justified.
Allocating methods: any method that may allocate (Vec growth, String growth, format!, collecting into Vec) is P for QM–B with bounded capacity rules; R for ASIL C; and typically X for ASIL D unless allocator is qualified and determinism is shown.
OS-dependent methods: filesystem, networking, environment variables, process control, and system time are R for QM–B and Q/R for ASIL C/D (platform layer only).
Unsafe methods: any `unsafe fn` or method requiring `unsafe` is Q for ASIL C/D and forbidden in application crates.
Concurrency methods: any method that can block, lock, or spawn threads must have a bounded-time and deadlock policy; treat as R for ASIL B–D.

Decision procedure (total mapping for every std/core/alloc function and method):

1) If the item is marked Experimental/unstable or requires nightly (`#![feature]`), classify as U (and therefore X for certified builds).
2) If calling the item requires `unsafe`, classify as Q for ASIL C/D (allowed only inside qualified TCB crates) and R for QM–B (enhanced review).
3) If the item can panic in a production build (documented panic or panicking precondition), classify as R for QM–B and X for ASIL C/D unless the panic is proven unreachable and justified as a fatal invariant.
4) If the item may allocate, classify by allocation policy: P for QM–B with bounded capacity + failure handling; R for ASIL C; and typically X for ASIL D unless a qualified allocator and determinism argument exists.
5) If the item interacts with OS/global state (filesystem, networking, env vars, processes, wall-clock time), classify as R for QM–B and Q/R for ASIL C/D (platform layer only).
6) If the item can block or introduce scheduling nondeterminism (locks, condvars, thread spawn, sleeps), classify as R for ASIL B–D unless bounded blocking and freedom-from-interference analysis is provided.
7) Otherwise, classify as P (Permitted), subject to the general coding rules (complexity limits, error handling, traceability).

11.5.2 Explicit high-risk API exceptions (always restricted/prohibited)

API / pattern	Status	Rationale	Preferred alternatives
`unwrap()` / `expect()` on Option/Result	R (QM–B), X (C/D)	Panics on error/None; hides error handling	Use `?`, `match`, or `unwrap_or` with defined fallback; treat panics as fatal invariants only.
Indexing (`v[i]`, `slice[i]`)	R (QM–B), X (C/D)	May panic; harder to prove bounds for MC/DC and review	Use iterators or `get()` returning Option and handle error path.
`panic!`, `todo!`, `unimplemented!`	R/X	Uncontrolled termination; placeholders unacceptable in released code	Return `Result` with error; use explicit fail-safe state transitions.
`mem::transmute`, `mem::zeroed`	X	High UB risk, invalid values, layout assumptions	Use explicit conversions, `from_ne_bytes`, `bytemuck`-like qualified wrappers (if approved).
`ptr::read`, `ptr::write`, `get_unchecked`	Q (C/D)	Unsafe memory access	Prefer safe APIs; encapsulate in audited safe abstractions.
`Rc` / cycles with `Arc` + `Weak`	R	Complex ownership and drop order; potential leaks	Prefer explicit ownership trees or arenas; document cycle-breaking strategy.
Hash-based collections (`HashMap`, `HashSet`)	P/R	Nondeterministic iteration order; DOS considerations	Prefer `BTreeMap/BTreeSet` for determinism; if HashMap needed, fix hasher and document.
`std::thread::spawn`	R (B–D)	Unbounded concurrency and scheduling nondeterminism	Prefer RTOS tasks with known scheduling; bound thread count;

			use message passing.

12. Tooling, Qualification, and Evidence Strategy (ISO 26262-8)

ISO 26262 relies on tools (compiler, static analysis, test/coverage tools) as part of the evidence chain. If tool output is used as safety evidence, the tool must be qualified to an appropriate confidence level, or alternative measures must compensate.

12.1 Compiler/toolchain qualification baseline

A qualification-oriented Rust language specification exists via the Ferrocene Language Specification (FLS), and Ferrocene provides qualified Rust toolchain distributions for safety-critical contexts. However, qualification is version-specific: a project must either use a qualified toolchain version, or perform its own qualification argument for the chosen compiler and standard library.

12.2 Tool roles and recommended evidence

Tool / activity	Purpose	Evidence artifacts
Compiler (rustc / qualified distribution)	Primary enforcement of the language subset, type system, and borrow checking	Qualified compiler version; locked flags; reproducible builds; documented baselines.
Formatter (rustfmt)	Deterministic formatting to reduce review noise	Locked rustfmt version/config; CI enforcement.
Static analysis (Clippy + custom lints)	Enforce subset rules, complexity limits, and bug patterns	Qualified lint set; reports stored as work products.
Unsafe analysis (Miri / UB checkers)	Detect undefined behavior in unsafe code paths	Run on all unsafe modules; store results.
Dependency auditing (cargo-audit / cargo-deny)	Detect known vulnerabilities and license issues	Regular scans; policy for remediation; SBOM.
Coverage tooling (llvm-cov/tarpaulin/etc.)	Provide structural coverage evidence	Qualified tool or validated workflow; consistent build flags; archive reports.

12.3 Inventory automation (recommended)

To enforce completeness, generate the language/library inventory automatically in CI (e.g., via rustdoc JSON) and compare it to the approved allowlist captured by this document. Any delta (new modules, new APIs, removed APIs) triggers a mandatory review and an update to Sections 10–11.

13. ASIL Profile Summary Matrices

This section summarizes the most consequential profile differences by safety level. Project-specific tailoring is allowed with documented rationale, but ASIL D deviations should be rare and strongly justified.

Policy topic	QM	ASIL A	ASIL B	ASIL C	ASIL D
Unsafe code	Allowed with normal review	Restricted; prefer safe	Restricted + enhanced review	Q only (TCB crates); 4-eye review	Q only + independent verification/formal argument
Heap allocation	Allowed	Allowed (bounded)	Allowed (bounded + monitoring)	Restricted (qualified allocator, deterministic)	Prefer no-alloc; otherwise Q allocator + strict bounds
Panics / unwinding	Allowed but	Restricted; define	Restricted;	panic=abort	panic=abort

	discouraged	policy	panic=abort preferred	required; no panics for expected errors	required; panics only for fatal invariants
Recursion	Allowed	Allowed with caution	Restricted; prefer iterative	Only bounded and justified	Only bounded and justified; prefer iterative
Dynamic dispatch (`dyn Trait`)	Allowed	Allowed with justification	Restricted in critical paths	Restricted; avoid in timing-critical	Strongly discouraged; prefer generics/static dispatch
Async/await	Allowed if needed	Restricted	Restricted	Restricted; qualified runtime required	Prohibited by default; allow only with qualified runtime and evidence
Conditional compilation (`cfg`)	Allowed	Restricted; test variants	Restricted; test variants	Minimize; verify all configs	Minimize; verify all configs; avoid safety logic behind cfg
Macros	Allowed	Allowlist recommended	Allowlist required for safety code	Allowlist required; proc macros qualified	Allowlist required; proc macros qualified; minimize macro complexity
Concurrency primitives	Allowed	Allowed with policy	Restricted (deadlock/priority)	Restricted (bounded blocking, analysis)	Restricted + scheduling analysis + freedom-from-interference evidence
HashMap/HashSet	Allowed	Allowed with justification	Restricted	Restricted; prefer BTree*	Prefer BTree*; HashMap only with fixed hasher and justification

14. References

ISO 26262:2018 — Road vehicles — Functional safety (all parts), with focus on ISO 26262-6 (software development) and ISO 26262-8 (supporting processes).
Rust Release Team. “Announcing Rust 1.93.1.” Rust Blog (Feb 12, 2026).
Rust Standard Library documentation for Rust 1.93.1: `std`, `core`, and `alloc` crates (doc.rust-lang.org).
The Rust Reference — Keywords (doc.rust-lang.org/reference/keywords.html).
Rust Edition Guide — Rust 2024 reserved syntax and `gen` keyword notes (doc.rust-lang.org/edition-guide/rust-2024/…).
Ferrocene Language Specification (FLS) repository (rust-lang/fls) and published spec (rust-lang.github.io/fls).
Ferrocene documentation and qualification information (ferrocene.dev; ferrous-systems.com blog posts on qualification).