

# C# Coding Standard

Version: 1.2

Date: November 2025

Based on: JPL Institutional Coding Standard (C) & RTCA DO-178B

Applicability: Mission-Critical .NET Applications

## Table of Contents

- 1. Rule Summary
- 2. Introduction
- 3. Scope
- 4. Conventions
- 5. Levels of Compliance
- 6. LOC-1: Language Compliance
- 7. LOC-2: Predictable Execution
- 8. LOC-3: Defensive Coding
- 9. LOC-4: Code Clarity
- 10. LOC-5: Mission Assurance
- 11. References

## 1. Rule Summary

### Language Compliance

- 1. **Conform to LTS C# Version:** Do not stray outside the language definition or use experimental features.
- 2. **Zero Warnings:** Compile with all warnings enabled ( `TreatWarningsAsErrors` ) and pass static analysis.

**Predictable Execution** 3. **Bounded Loops:** Use verifiable loop bounds; prefer `foreach` over `while` . 4. **No Recursion:** Do not use direct or indirect recursion. (Note: `goto` is obsolete for cleanup in C#). 5. **Zero-Allocation Hot Paths:** Do not use dynamic memory allocation ( `new` ) in critical paths. Use `Span<T>` / `ArrayPool` . 6. **Async/Await Sync:** Use `Task` -based patterns; do not use `Thread.Sleep` or blocking locks.

**Defensive Coding** 7. **Null Safety:** Enable Nullable Reference Types and strictly adhere to null checks. 8. **Limited Scope:** Declare data objects at the smallest possible level of scope; prefer immutability. 9. **Guard Clauses:** Check the validity of all values passed to public functions immediately. 10. **Specific Exceptions:** Do not catch generic `System.Exception` ; do not use exceptions for control flow.

**Code Clarity** 11. **No Unsafe Code:** The use of pointers ( `unsafe` blocks) is prohibited. 12. **Limited Preprocessor:** Limit preprocessor use to build configurations; no logic in directives. 13. **Small Functions:** Methods should not exceed 60 lines or a cyclomatic complexity of 10. 14. **Clear LINQ:** Avoid complex, multi-line LINQ query chains in critical logic.

**Mission Assurance** 15. **Fault Isolation:** Use the Bulkhead pattern; non-critical subsystems must not crash critical paths. 16. **Deterministic Cleanup:** Enforce `IDisposable` ; do not rely on the Garbage Collector/Finalizers for resource release. 17. **Auditability:** All critical state changes must be append-only and traceable (Event Sourcing).

## 2. Introduction

Considerable efforts have been invested by organizations like NASA/JPL and MISRA to develop coding standards for C/C++. This standard translates those high-integrity principles into the managed C# environment.

Crucially, this version incorporates principles from high-assurance domains where system failure, latency stalling, or non-

determinism can have catastrophic consequences. It applies strict rigor to managed code to ensure stability in environments where "restarting" or "patching later" is not an acceptable mitigation strategy.

### 3. Scope

The coding rules defined here primarily target the development of **High-Integrity C# Software**. This includes:

- Real-time control or simulation systems.
- High-volume, high-availability transaction processors.
- Safety-critical data processing pipelines.
- Core infrastructure shared across multiple critical systems.

### 4. Conventions

- **Shall:** Indicates a requirement that must be followed. Non-compliance must be flagged as a build error.
- **Should:** Indicates a preference that must be addressed. Non-compliance generates a warning and requires justification (suppression).

### 5. Levels of Compliance

- **LOC-1 Language Compliance:** Ensures the code builds reliably across environments.
- **LOC-2 Predictable Execution:** Focuses on resource constraints (Stack, Memory, CPU).
- **LOC-3 Defensive Coding:** Focuses on robustness against invalid state.
- **LOC-4 Code Clarity:** Focuses on maintainability.
- **LOC-5 Mission Assurance:** Focuses on systemic stability, fault isolation, and auditability.

### 6. LOC-1: Language Compliance

#### Rule 1 (Language)

All code shall conform to a specific Long Term Support (LTS) C# Language Version. Dependency on "Preview" features is prohibited.

#### Rule 2 (Routine Checking)

All code shall be compiled with all warnings enabled and treated as errors. Set `<TreatWarningsAsErrors>true</TreatWarningsAsErrors>`.

### 7. LOC-2: Predictable Execution

#### Rule 3 (Loop Bounds)

All loops shall have a statically determinable upper bound. Prefer `foreach`. `while(true)` is prohibited except in host service lifecycles.

#### Rule 4 (Recursion)

There shall be no direct or indirect use of recursive function calls.

- **Rationale:** Recursion risks `StackOverflowException`, which instantly terminates the process.
- **Note on `goto`:** In C, `goto` is often used for single-point cleanup. In C#, this pattern is obsolete. Resource cleanup shall be handled via `using` statements and `try/finally` blocks, which guarantee execution even during exceptions.

#### Rule 5 (Heap Memory)

Dynamic memory allocation (`new`) shall be avoided in "Hot Paths".

- **Rationale:** Allocation triggers GC pauses, causing unacceptable latency ("stalling").
- **Guideline:** This rule is difficult to adhere to in C#. To comply, developers must use `struct`, `ArrayPool<T>`, and

`Span<T>` in critical loops to reuse memory buffers rather than allocating new objects.

#### Rule 6 (Task Synchronization)

Task synchronization shall not be performed through `Thread.Sleep` or blocking locks. Use `async / await` and `System.Threading.Channels`.

### 8. LOC-3: Defensive Coding

#### Rule 7 (Null Safety)

Nullable Reference Types shall be enabled project-wide. Address all CS8600-series warnings.

#### Rule 8 (Limited Scope)

Data objects shall be declared at the smallest possible level of scope. Prefer `private`, `internal`, and immutable (`record`) types.

#### Rule 9 (Checking Parameter Values)

The validity of function parameters shall be checked at the start of each public function. Throw `ArgumentException` immediately if validation fails.

#### Rule 10 (Exception Handling)

Exceptions shall not be used for control flow. Do not catch generic `System.Exception`.

### 9. LOC-4: Code Clarity

#### Rule 11 (Unsafe Code)

The `unsafe` keyword and pointer arithmetic shall not be used.

#### Rule 12 (Preprocessor Use)

Use of the C# preprocessor shall be limited to build configuration.

#### Rule 13 (Complexity)

Functions should be no longer than 60 lines of text.

#### Rule 14 (LINQ Usage)

LINQ Query Syntax should be avoided in complex logic chains.

### 10. LOC-5: Mission Assurance

#### Rule 15 (Fault Isolation)

Non-critical subsystems must not compromise critical paths.

- **Concept:** "Bulkheading" (System Partitioning).
- **Implementation:** Subsystems like Logging, Analytics, or Telemetry must run in isolated Tasks or Processes. If they crash or hang, the Core Execution loop must continue unaffected. Wrap non-critical calls in `try/catch` with timeouts.

#### Rule 16 (Deterministic Cleanup)

Resources must be released deterministically; do not rely on Finalizers.

- **Rationale:** Waiting for the Garbage Collector to close a file handle, socket, or connection is non-deterministic and unsafe.
- **Implementation:** Any class owning native resources must implement `IDisposable`. Usage must be strictly wrapped in `using` statements.

## Rule 17 (Auditability)

Critical state changes must be traceable and append-only.

- **Rationale:** In mission-critical systems, the history of *how* a state was reached is often as important as the state itself for post-incident analysis and recovery.
- **Implementation:** Use Event Sourcing or immutable Audit Logs. Never overwrite critical state data; always append a new state record.

## 11. References

1. JPL D-60411: JPL Institutional Coding Standard for C.
2. RTCA DO-178B: Software Considerations in Airborne Systems and Equipment Certification.
3. The Power of 10: Rules for Developing Safety-Critical Code, Gerard J. Holzmann.