Electronic Design

GRAEME WINTLE, Director, ByteSnap Design,
https://www.bytesnap.com/.

The Human Side of MISRA C Compliance

MISRA C compliance succeeds when teams understand its safety purpose, apply it pragmatically, and build it into daily embedded-software development.

"We're implementing MISRA C on this project." I've watched that announcement land in team meetings with the same response, almost always crossed arms, heavy sighs, and a palpable shift in the room's energy. The real battle is winning over your development team.

What often gets missed is that MISRA C compliance rarely fails because of technical difficulty. The rules can be learned, the tooling can be configured, and the violations can be fixed. The real challenge is convincing experienced developers that the standard exists to help them write safer software, not to undermine their judgment or slow them down.

After managing MISRA C implementations across a wide range of automotive infotainment systems and ECUs, one thing has become clear: Successful compliance depends far more on how teams are brought along than on which tools are used to enforce the rules.

Why Developers Hate MISRA C and Why They're Not Wrong

Developers aren't being difficult when they push back. They're just responding to legitimate concerns that management often dismisses too quickly. MISRA C faces such resistance because it often feels like bureaucracy for bureaucracy's sake.

When you're an experienced embedded developer who's been writing tight, efficient C code for years, being told that your go-to statement violates Rule 15.1 feels insulting. You know exactly why that go-to is there. You've considered the alternatives, so the rule feels arbitrary.

This reaction intensifies when developers encounter what they perceive as pedantic violations flagged by static-analysis tools: A perfectly safe pointer operation gets flagged, while an efficient bit manipulation triggers warnings. Soon, developers begin to see MISRA C not as a safety standard, but as an obstacle to getting real work done.

MISRA C for automotive safety. (Source: ByteSnap Design)

In our experience at ByteSnap Design, this is where the first critical mistake happens. Teams don't take the process seriously at the start. Instead, they view MISRA as a box-ticking exercise rather than a genuine safety framework.

The Real Productivity Cost of MISRA C Compliance

This isn't paranoia; it's a reality. When building MISRA compliance into new development from the start, we see approximately 10% to 15% additional development time. But retrofitting existing code? That's a completely different story. Legacy code compliance can add 30% to 35% to your effort, and that's assuming your existing practices generally align with MISRA C principles.

For developers measured on feature delivery, this feels like punishment. Suddenly, they're spending more time justifying their code than writing it.

The natural response is resentment when developers encounter rules that seem divorced from real-world embedded constraints. They begin to question the entire standard's validity. However, the reality is that many MISRA C rules exist because of real failures, such as buffer overflows that caused ECU crashes, pointer errors that corrupted sensor data, and type conversion bugs that led to vehicle recalls.

The moment developers conclude that MISRA C was created in an ivory tower by people who don't understand their real-world constraints, you've already lost them. At that point, compliance becomes an exercise in appeasement rather than a shared commitment to safety, and no amount of tooling or enforcement will fix that disconnect.

The First 30 Days are Where Most MISRA Implementations Fail

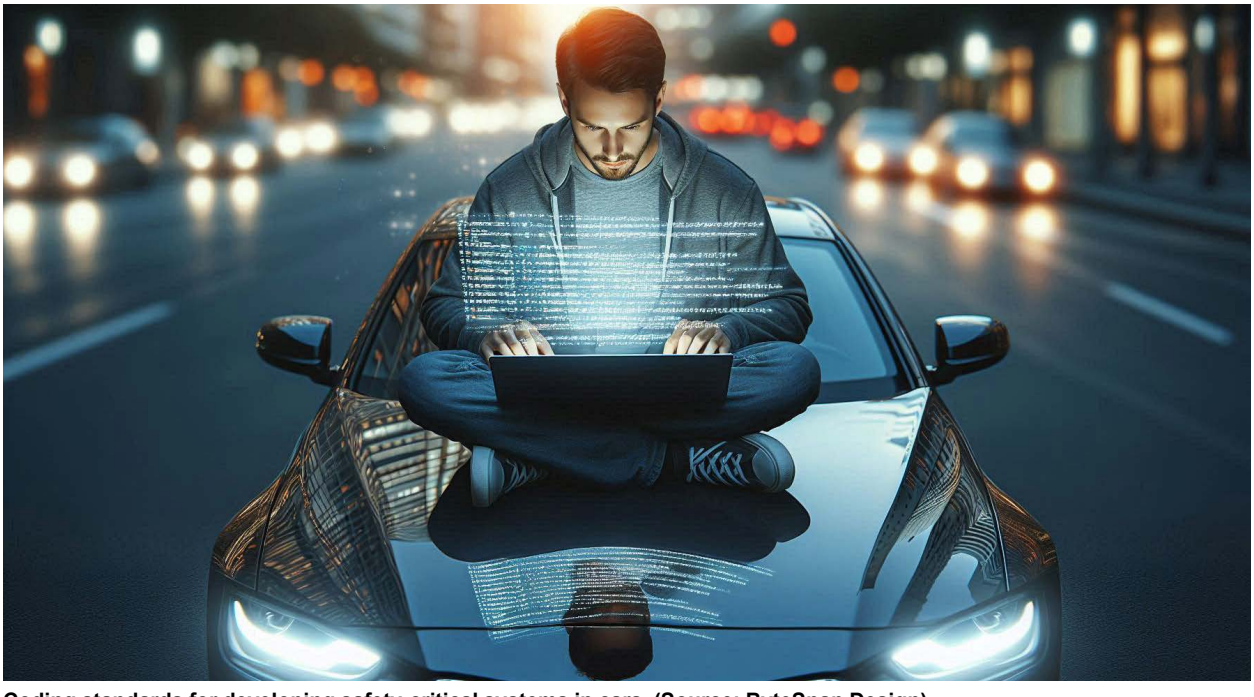
The way you introduce MISRA C to your team determines whether it becomes a constructive part of everyday development or an ongoing source of friction. The most common failure is not taking the process seriously from day one. Teams treat MISRA as a checkbox exercise rather than a fundamental shift in how software is developed, and that mindset sets the tone for everything that follows.

The solution starts before you write a single line of code, when you create a compliance plan with your stakeholders. This means sitting down with your customer (or whoever is mandating MISRA C) and agreeing on which rules are required versus advisory for your specific application.

MISRA C does provide flexibility, whereby some rules are essential for safety; others are contextual good practices. If you skip this categorization phase, you'll spend months arguing about which rules truly matter for your specific system.

The second major mistake is running MISRA C checks only at the end of a project. We addressed violations as we developed a Bluetooth project, which is why remediation was quick despite the complexity.

When you integrate compliance checking into your daily development workflow, issues stay small and manageable. If you wait until the end, violations accumulate into technical debt that forces risky late-stage changes. The difference between 10% to 15% additional effort (continuous compliance) and 30% to 35% (retrofit) isn't just timing — it's the



Coding standards for developing safety-critical systems in cars. (Source: ByteSnap Design)

difference between building in quality versus bolting it on afterwards.

Most MISRA C violations occur in memory management. Pointer checks, bounds handling, and allocation rules all prevent buffer overflows, crashes, and security vulnerabilities that can lead to real-world failures. When developers understand that careful memory management is the core safety issue, not an arbitrary restriction, resistance drops.

Excessive compliance can also harm your project just as much as insufficient compliance. Making all advisory rules mandatory unnecessarily restricts developers. When every advisory rule becomes mandatory, developers waste time working around arbitrary constraints instead of solving real problems. This leads to convoluted code without improving safety. For example, blanket bans on go-to statements ignore that go-to has legitimate uses in embedded systems for error handling and resource clean-up.

A thoughtful approach examines each use case rather than applying blanket prohibitions. The key is understanding which rules carry the greatest safety impact for your specific application. In our experience, memory-management rules — specifically accessing memory and checking pointers — consistently matter most.

Building Buy-In

If you're about to introduce MISRA C to your development team, these are the points that set you up for genuine success:

- **Be explicit about the cost:** Building MISRA C compliance into development from the start typically adds around 10% to 15% to delivery time. Retrofitting compliance later can add 30% to 35%, even when existing practices are reasonably aligned.
- **Explain why compliance is non-negotiable:** Tie MISRA C adoption to real, project-specific consequences, such as vehicle recalls, safety failures, security risks, or certification requirements. Make it clear this isn't about bureaucracy, but about avoiding unacceptable outcomes.
- **Define how MISRA will be applied:** Agree upfront which rules are required and those that are advisory for your application. MISRA C is deliberately flexible; treating every rule as mandatory wastes effort without improving safety.
- **Integrate compliance into daily development:** Run MISRA C checks continuously so that violations remain small and manageable, rather than accumulating into late-stage technical debt.
- **Handle unavoidable violations pragmatically:** For third-party or legacy code you can't fully control, document justified deviations rather than forcing unsafe or artificial workarounds.
- **Keep the focus on safety, not reports:** The goal isn't perfect compliance metrics, but software that behaves predictably and fails safely when the unexpected occurs.

The Tools You Actually Need

One area where teams waste money is over-investing in expensive tools before understanding their actual needs. For development, we primarily use Cppcheck with the MISRA plugin. The free version handles basic checks reasonably well. The premium service costs around £1,800 to £2,000 per project annually, which is far more affordable than automotive-certified tools that can cost tens of thousands.

We've evaluated tools like Helix QAC and SonarCube. One client uses Helix internally, but for many projects, the ROI doesn't justify the cost during development phases. The staged approach works, so use cost-effective tools during development, and reserve expensive automotive-certified tools for final certification when customer requirements mandate them.

MISRA C doesn't operate in isolation. It's typically a component-level requirement within ISO 26262, the overarching automotive safety standard that encompasses hardware, software, suppliers, and vehicle design. Understanding this relationship is crucial. You can have MISRA C-compliant code that still fails ISO 26262 if your architecture is fundamentally flawed. MISRA C addresses coding practices; ISO 26262 addresses system safety.

We apply similar principles across industrial, medical, and defense sectors, where different standards apply. Still, the fundamental challenge remains in writing safe embedded code that behaves predictably under all conditions.

When to Bring in External Help for Better Code Beyond Compliance

Clients approach us because they lack the internal time or resources to complete MISRA C compliance in-house. They recognize the importance of the standard but don't have engineers who can focus solely on compliance while also delivering features. Our expertise is particularly strong in making existing legacy codebases compliant.

We're a good fit for small- to medium-scale embedded C projects where teams need practical, experience-led compliance. For extremely large systems or complex multi-language environments, dedicated certification partners are often a better fit.

The key differentiator is that we bring breadth of experience from various industries, such as industrial, medical, and defense, rather than automotive-only specialization. This cross-industry perspective helps teams avoid rote rule-following in favor of genuine safety improvement.

After developers move through initial resistance and into genuine engagement with MISRA principles, something interesting happens. Their code quality improves in ways that

have nothing to do with automotive certification. MISRA C's memory-management rules force developers to think more carefully about data ownership, lifetime, and access patterns. The resulting code is cleaner, more maintainable, and easier to reason about.

Teams consistently report that onboarding junior engineers becomes easier after MISRA C adoption. The standard provides clear guidelines that reduce the "tribal knowledge" new team members must absorb.

The Difference Between Success and Failure

MISRA C compliance fails most often due to organizations underestimating the human dynamics involved in enforcing the rules, not because they can't understand them. Teams either adopt the standard as a living part of how they write software, or they reduce it to something developers endure to satisfy a process.

Successful implementations are marked by clarity of intent, consistency of application, and trust between engineers and decision-makers. Failed ones rely on enforcement, late-stage remediation, and the assumption that tools can compensate for disengaged teams.

That transformation doesn't happen through executive decree or expensive tools. It happens through:

- Taking the process seriously from day one with clear stakeholder alignment.
- Running continuous checks throughout development, not just at the end.
- Focusing on what actually matters — memory management and real safety risks.
- Avoiding over-compliance that wastes time without improving safety.
- Being honest about the productivity impact (15% for new development, 30% to 35% for retrofit).

When you get these fundamentals right, MISRA C transforms from a dreaded mandate into a valued framework for writing safer embedded software.

ByteSnap co-founder Graeme Wintle is one of the country's most experienced Microsoft Windows CE developers. A software engineer graduate from Newcastle University, Graeme quickly distinguished himself in software development at Nokia, GST Technology, and Intrinsyc. His cutting-edge design work has been instrumental in establishing ByteSnap Design at the forefront of embedded system design. He's the inventor of the powerful user interface development framework SnapUI.