LDRA
Software Technology

Technical White Paper

Get ahead with the MISRA C guidelines

An embedded developer's guide to compliance, including the latest MISRA C:2023 update

www.ldra.com

From automotive software to medical devices, MISRA C continues to be the leading set of guidelines that drive safe, secure, and reliable code. Aligning with the increasingly complex and connected nature of embedded products, MISRA C meets the needs of embedded software developers right at the source of potential issues. Mature organizations have seen the long-term benefits of MISRA Compliance in their market adoption and customer satisfaction rates while new entrants are leveraging MISRA as a competitive differentiator in highly regulated industries like electric vehicles (EV) and aerospace.

The recent releases of MISRA C:2012 Amendment 4 (AMD4) and MISRA C:2023, which consolidates previous versions of the guidelines, bring new guidance on multithreading and atomic types in support of C11 and C18, making MISRA more relevant to embedded products than ever before.

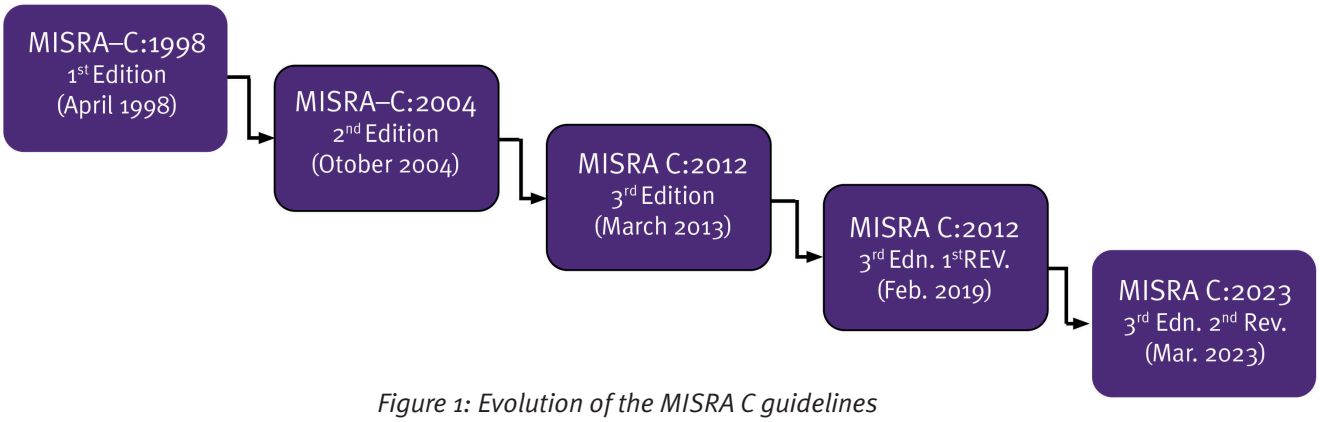

Figure 1: Evolution of the MISRA C guidelines

By reading this paper, you get an understanding of the MISRA C guidelines, an explanation and examples from AMD4, and strategies towards championing and deploying a compliance tools framework within your team.

Understanding the principles behind MISRA C

Software safety, security, and reliability are critical imperatives for many embedded systems. The C language, originally built as a lightweight and friendlier alternative to assembler, saw developer adoption rates increase much faster than any syntax or semantic updates for safety- or security-critical concerns. The MISRA C guidelines brought the C language in line with the requirements of safety- and security-critical systems through rules and directives that minimize or eliminate coding practices known to be hazardous.

The influence of MISRA C on the embedded software industry cannot be overstated, as it is either directly cited by or commonly practiced in projects following AUTOSAR, IEC 62304, IEC 61508, ISO 26262, and DO-178C processes. It also forms the basis of the Joint Strike Fighter C++ Coding Standard and the NASA Jet Propulsion Library C Coding Standard.

The evolution of the MISRA guidelines and the increasing complexity of embedded software has also fostered decades of advancements in compliance checking tools – MISRA C itself recommends the use of static analysis tools to ensure compliance.

Why MISRA C matters

The C language allows developers to control application behavior and memory access in ways that could compromise the safety, security, and reliability of a system. While application code may meet the requirements of the C language standard, it could cause undesirable effects in critical systems that lead to a security breach or loss of life. Examples include:

- Writing to a file stream opened as read-only, leading to potentially undesirable behavior
- Using functions that call themselves (i.e., recursion), leading to a potential stack overflow
- Accessing memory outside the bounds of a data structure, which could lead to a common exploit used by hackers

Some C compilers can identify risky coding implementations but it's more efficient and cost effective to prevent issues before they're introduced into the code base. MISRA C restricts language use to a safety-critical subset, guiding developers to avoid and eliminate potentially dangerous code.

Using a static analysis tool to automate MISRA C Compliance helps the team identify and remediate issues as early as possible in the lifecycle.

Aligning to the C language

The evolution of the MISRA C guidelines follows the path of C language editions and the real-world experiences of users as captured by the MISRA C Working Group:

- **MISRA C:1998** – the first edition, titled "Guidelines for the use of the C language in vehicle based software," contained 127 rules for C90 and is still used today for the maintenance of legacy systems.
- **MISRA C:2004** – renamed the document to "Guidelines for the use of the C language in critical systems," in recognition of its applicability to software beyond automotive and added new rules under sections corresponding to different aspects on an embedded system.
- **MISRA C:2012** – extended coverage to C99 and incorporated feedback from users, including rationale behind the guidelines, a classification of "decidable" and "undecidable" rules to clarify how checking tools can identify compliance issues, and the addition of "directives" for which compliance is more flexible in interpretation.

Subsequently, MISRA C:2012 has continued to evolve, particularly to address additional security considerations and more recently, to incorporate the newer editions of the C standard (C11 and C18):

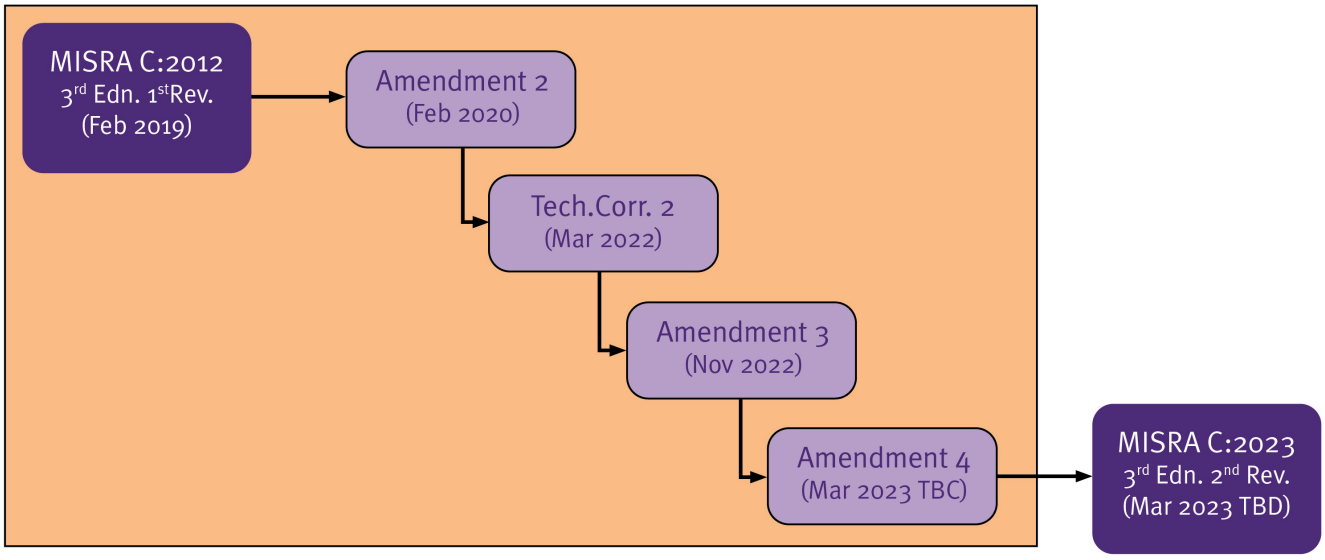

Figure 2: Evolution of MISRA C:2012 to MISRA C:2023

- **MISRA C:2012 Amendment 1 (2016)** – published as the result of a coverage comparison between the existing MISRA C:2012 guidelines and ISO/IEC 17961:2013, the C secure coding rules published by the C standard committee, this specified 14 security guidelines in recognition of the applicability to improving the cybersecurity posture of embedded systems.
- **MISRA C:2012 Amendment 2 (2020)** – introduced coverage of the ISO/IEC 9899:2011 standard, by mapping the undefined and unspecified behaviours to the existing MISRA C guidance.
- **MISRA C:2012 Amendment 3 (2022)** – added 24 rules and 1 directive to reflect new C11 and C18 language features outlined in the ISO/IEC 9899:2011/2018 standard.
- **MISRA C:2012 Amendment 4 (2023)** – added 19 rules and 3 directives to reflect multithreading and atomic types as specified in the ISO/IEC 9899:2011/2018 standards.

In support of these sets of documents, MISRA has produced mappings of their guidelines to relevant standards:

- MISRA C:2012 Addendum 2 – mapping MISRA C:2012 to ISO/IEC 17961 "C Secure"
- MISRA C:2012 Addendum 3 – mapping MISRA C:2012 to the CERT C coding standard

Lastly, MISRA C:2023 consolidates all prior MISRA C:2012 editions, amendments, and technical corrigenda into one document.

"By consolidating the recent enhancements into a single source, MISRA C:2023 provides the new benchmark guidance for developers of safety- or security-related,or indeed any high-integrity, software."

Andrew Banks, LDRA technical specialist and MISRA C Working Group chairman

Read more
Amazing MISRA C:2023 Update

How MISRA structures the guidelines

MISRA C:2012 introduced the classification of guidelines as rules or directives:

Rule: A source code requirement that is complete, objective, and unambiguous. Developers can use analysis tools to check compliance with rules. The guidelines further classify rules as decidable if they can be conclusively verified through analysis and undecidable if no verification guarantee is possible.

Example rule: "Conversions shall not be performed between a pointer to a function and any other type." (MISRA C:2012 Rule 11.1)

Directive: A guideline that may be satisfied through code, processes, documentation, or functional requirement. Directives can be subject to interpretation and analysis tools may be able to assist in checking compliance.

Example directive: "Any implementation-defined behavior on which the output of the program depends shall be documented and understood." (MISRA C:2012 Directive 1.1)

Read more
How LDRA supports MISRA C Compliance

MISRA C and cybersecurity

The perception that the guidelines applied only to safety-related concerns and not cybersecurity prompted the MISRA C Working Group to release the first of the MISRA C:2012 amendments. While many requirements for code safety apply equally to code security, establishing explicit guidelines to address known vulnerabilities gave developers and static analysis tools a framework to improve the security posture of applications.

Take MISRA C:2012 Rule 12.5, stating "The sizeof operator shall not have an operand which is a function parameter declared as 'array of type'." Commonly used to calculate the size of an array, the `sizeof()` function could lead to a potential threat vector when its argument is an array passed as a function parameter, as shown in Figure 2. Passing A[] to `sizeof()` on line 6 may result in an incorrect value, as it was passed into the function as a "pointer to type". The result could lead to an array bound being exceeded – a common exploit used by hackers.

```
1  #include <stddef.h>
2  #include <stdint.h>
3
4  static int32_t calculate_threshold (int32_t A[ 2 ])
5  {
6      size_t size = sizeof ( A ) / sizeof ( A[ 0 ] );
7      ...
8      return A[ size - 2 ];
9  }
10
11 int main (void)
12 {
13     int32_t A[] = { 1, 2 };
14     ...
15     return calculate_threshold ( A );
16 }
```

Figure 3: Code sample showing the potentially dangerous use of the sizeof() function

Introducing MISRA C:2012 Amendment 4

Following hot on the footsteps of the Amendment 3 release in late 2022, MISRA C:2012 Amendment 4 extends support to many new features introduced by the C11 and C18 standards, ISO/IEC 9899:2011 and 2018. Focused on the growing use of concurrency in embedded software, AMD4 specifies rules and directives for multithreading and atomic types as well as modifications and clarifications on existing guidance to better align with how developers use the C language today.

New guidance on multithreading

C11 introduced standard methods of multithreading to support the concurrency needs of embedded systems without requiring developers tying themselves to one specific implementation (e.g., POSIX pthreads). The language added features for threads, condition variables, mutexes, and more which led to the possibility of developers introducing unexpected and undesirable side effects into the system.

AMD4 addresses many of these issues by adding twelve rules and three directives to restrict multithreading features to a safe subset. These guidelines cover critical elements of thread usage:

- Restricting dynamic thread creation to foster more deterministic approaches to concurrency
- Ensuring threads are created before mutexes are linked to them
- Minimizing the risk of deadlocks and data races in the system
- Managing the safe use of thread objects and thread identifiers

For example, AMD4 includes a new rule that states memory storage that's only supposed to be accessed within one thread shall only be accessed by that thread. Since the C language doesn't prevent a developer from allocating thread-specific storage in the common user space, this rule helps minimize unintended or undesirable access by other threads.

New guidance on C atomic types

C11 also introduced semantics and a memory model for a set of atomic types and operations that allowed developers to manipulate data objects indivisibly, or without risk of interference by another thread. While the goal was to reduce the likelihood of data races, the C language has other features that make it difficult for a developer to guarantee atomicity across the entire system.

With AMD4 comes three new rules and modifications to three existing rules to support developers in their atomic pursuits:

- Ensuring the correct configuration of atomic types
- Preventing the unintended removal of atomicity when referencing atomic types through pointers
- Restricting the use of multiple atomic types in the same statement

Additional guidance

The AMD4 update includes four new rules for C language features that have proven to be problematic over the years. These rules restrict the following:

1. Use of small integer macros to protect against cases of unexpected or unpredictable behavior when the compiler expands the macro
2. For aggregate types (arrays, structures, unions), the combined use of undesignated initializers (variables in a comma-separated list) and designated initializers (explicit element names) on the same type to prevent multiple initializations of the same element
3. Relaxing the restriction on variably modified arrays (with guidance) while maintaining the restriction on the use of variable length arrays (VLA)
4. Use of variables that the application never references (this adds to existing guidance on unused macros, structures, and other data types)

Additionally, there are several minor updates to modernize existing guidance, such as updating MISRA C:2012, 3.1 (“The character sequences /* and // shall not be used within a comment”) with an exception to allow the use of “//” to support website URLs.

Introducing MISRA C:2023

To commemorate the 25th anniversary of the publication of the original MISRA C guidelines, the MISRA C Working Group published MISRA C:2023. This edition consolidates earlier editions of the guidelines with the recent AMD4 enhancements to provide a single, comprehensive baseline for existing users of MISRA C and ease entry for companies starting new projects.

How to incorporate MISRA C Compliance into development processes

Compliance to MISRA C begins by updating software development processes to ensure training, documentation, tools, and metrics incorporate testing and reporting against the guidelines. Given the nature and scope of what embedded software can do, these processes must validate compliance to MISRA C’s decidable and undecidable rules, usually through a combination of static analysis tools, documentation, and manual effort.

When updating development processes, teams should consider the following items.

Document the MISRA Compliance process

MISRA recommends the inclusion of the MISRA C guidelines into a project's software development processes. These processes should be formalized and specify a rigorous approach to ensuring complete, unambiguous, and correct software requirements that are wholly and solely implemented by the outputs of each phase of the development lifecycle.

Part of this approach should be MISRA Compliance – how people, processes, and tools work together to achieve it and detailing the requirements for claiming compliance to the MISRA C guidelines, including the reporting of violations.

Understand where automation can help

In general, compliance to MISRA's decidable rules can be proven through automated techniques like static analysis while the undecidable rules may require additional supporting evidence. Attempting to use a single tool to validate an undecidable rule can result in false positives or false negatives because there isn't enough information to guarantee an accurate assessment.

MISRA directives are usually subjective and difficult to validate with a single automated tool. For example, Directive 1.1 of MISRA C:2012 requires that "Any implementation-defined behaviour on which the output of the program depends shall be documented and understood" – an artifact that static analysis cannot validate.

Know how to document deviations

> "The management of guideline violations is a critical issue. Violations are sometimes unavoidable and a claim of compliance can only be meaningful when they are authorized through a clearly defined process and supported by deviation records."
>
> - MISRA Compliance:2020 guide

MISRA allows deviations from its guidelines where proving compliance might be impractical or impossible to follow. These deviations must be documented and authorized, including the guideline, location of deviation, rationale for deviation, and a risk analysis.

Project	F10_BCM		
Deviation ID	D_00102	Status	Approved
Permit	Permit / Example / C:2012 / R.10.6.A.1		
Rule 10.6	The value of a composite expression shall not be assigned to an object with wider essential type		
Use case	The value of a composite expression is assigned to an object of wider essential type to avoid sub-optimal compiler code generation		
Reason	Code Quality (Time behaviour)	Scope	Project
Tracing tags	D_00102_1 to D_00102_10		

Raised by	E C Unwin	Approved by	D B Stevens
	Signature		*Signature*
Position	Software Team Leader	Position	Engineering Director
Date	14-Mar-2015	Date	12-Apr-2015

Figure 4: Example MISRA deviation record (Source: MISRA Compliance:2020 guide)

Define your MISRA Compliance tool stack

Selecting a new compliance tool can be tricky but MISRA itself provides guidance. The guidelines recommend using static analysis tools to validate code against rules and directives. The MISRA Compliance:2020 guide provides information on how to select and configure a static analysis tool, following these basic principles:

1. The tool should support the MISRA version relevant to the project and enforce as many guidelines as possible, including the latest AMD4/MISRA C:2023.
2. The team should validate the tool to gain confidence in its outputs (some standards, including IEC 61508, ISO 26262, and DO-178, require qualification of static analysis tools in certain situations).
3. The tool should support the necessary languages and characteristics of the environment (e.g., compiler settings and language extensions).
4. The tool should be capable of producing metrics that support code coverage and the identification of code that may require additional review.
5. The tool should support the analysis of legacy and third-party code as much as possible, including open source software, vendor libraries, and device drivers.

Not explicitly mentioned is the need for the static analysis tool to support the speed of today's embedded software development, fitting into agile processes and continuous integration and continuous delivery CI/CD) pipelines.

Watch:

Accelerating the Path to Standards Compliance with TBmanager®

How LDRA supports MISRA Compliance

LDRA streamlines and improves the effectiveness of MISRA Compliance through automated code checking and reporting through the LDRA tools suite, and expert training and consultancy services. Embedded software teams from aerospace, industrial and energy, medical device, and automotive sectors use the LDRA MISRA checkers to ensure the safety, security, and reliability of their code.

The LDRA tool suite employs static analysis to identify areas of non-conformant code to aid documentation and modification and includes extensive reports and graphical displays to enhance understanding of the source code in line with MISRA guidelines. LDRA static analysis tools also facilitate structural coverage analysis to ensure developers can measure and maintain the amount of tested code, as recommended by the MISRA guidelines.

LDRA has long supported MISRA, bringing its expertise in standards compliance, automated software verification, static code analysis, and test tools to evolve the standard to better align to modern C language and development practices. LDRA technical specialists Andrew Banks and Chris Tapp serve as the MISRA C Working Group chairman and the MISRA C++ Working Group chairman, respectively.

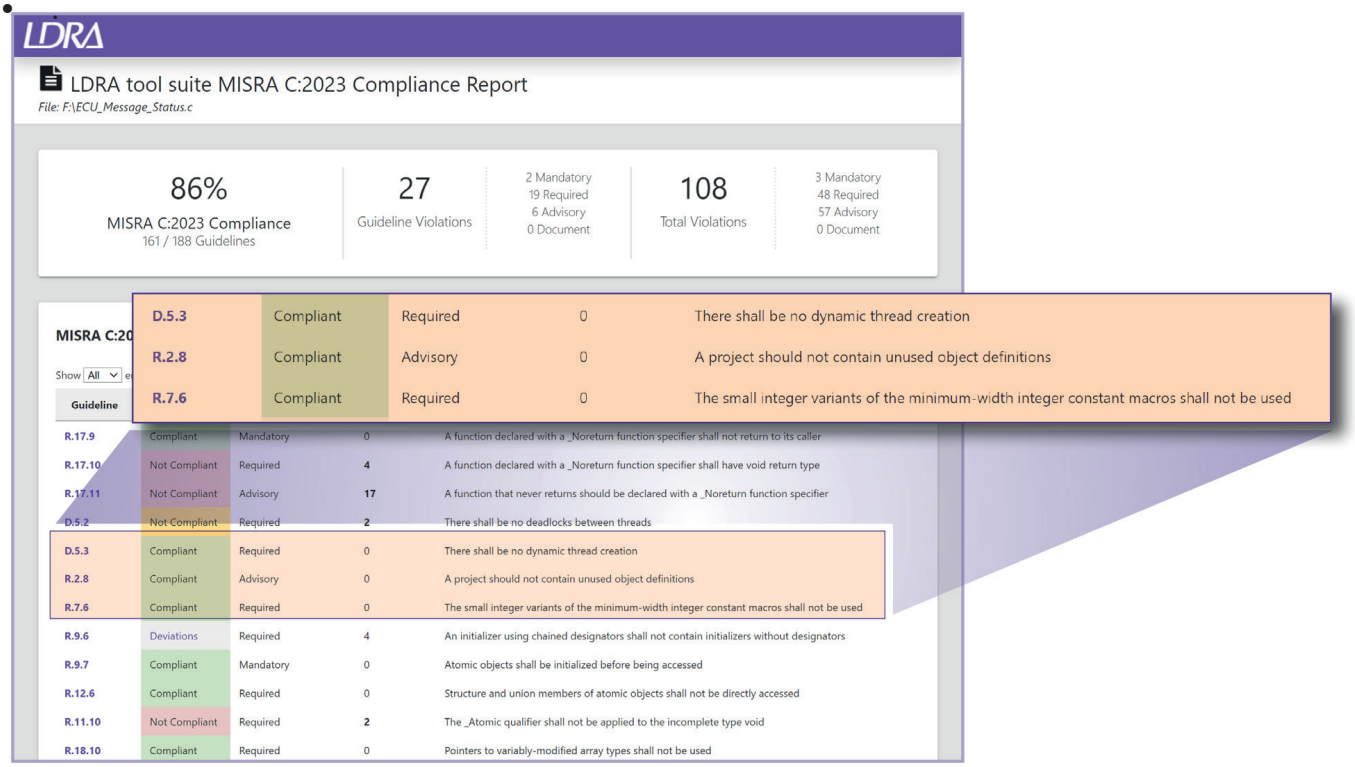

Figure 5: Example MISRA Compliance report from the LDRA tool suite

The full complement of MISRA Compliance tools from LDRA include these capabilities:

- Automated source code checking for conformance to any version of the MISRA language subsets, including the latest AMD4/MISRA C:2023.
- Extensive reports and visualizations to simplify understanding of code and remediate issues faster.
- Structural coverage analysis to increase confidence in test coverage and track changes over time, as recommended by the MISRA guidelines.
- Efficient management and tracking of justified rule deviations.
- Certification by TÜV SUD for IEC 61508, ISO 26262, IEC 62304, and EN 50128 functional safety standards.

How to champion static analysis for successful MISRA Compliance

To advocate for the adoption of static analysis tools as a vital component of MISRA Compliance, it's important to clearly communicate the potential risks and costs of not doing so, and the benefits of improved quality, security, and efficiency. It may also be helpful to provide examples of other organizations that have successfully implemented static analysis, highlighting its positive impact on compliance activities.

"The retrospective application of MISRA C:2012 proved less challenging than imagined. LDRA rules helped significantly by breaking down the high level MISRA guidelines to more granular and complete definitions."

- Case study: Renishaw

There are four key arguments to convince software development leaders to adopt a static analysis tool.

1. Streamline compliance efforts to reduce costs

Static analysis tools improve the efficiency of MISRA Compliance by identifying risks and pinpointing suspect code for developers to investigate further. Given the scale and complexity of today's code bases–often spanning multiple software units, stored in multiple locations, and different product lines–these tools are far more comprehensive and accurate than manual inspection and traditional testing methods.

Static analysis leads to faster development times and lowers MISRA Compliance costs in the following ways:

- Identification of issues before software unit testing ("shift left") by surfacing MISRA violations at the developer's desktop, in context with the original source code.
- Comprehensive code coverage that is difficult or cost-prohibitive to scale using traditional methods.
- Prevents defects before they're introduced and minimize technical debt by enforcing a continuous coding and compliance checking discipline every time the developer changes code.
- Beyond coding standards compliance, static analysis techniques can also measure the overall quality of code, including complexity, clarity, maintainability, and testability.

2. Support developer training on MISRA C

Static analysis tools tend to be rules driven and its outputs provide the ideal training ground for developers to better understand how MISRA C applies to their code. Whether an experienced developer or a new graduate, analysis findings are a type of microlearning, where compliance issues and related information are broken down into bite-sized chunks that are easier to understand in the context of code and more likely to be retained.

This "in-the-moment" training helps those unfamiliar with MISRA C to come up to speed fast and provides guardrails for experienced developers to focus on features rather than compliance.

Watch:
Software engineering: Being compliant with MISRA C and MISRA C++

3. Consolidate MISRA versions into a single source

Keeping tabs on MISRA versions and guidance documents can be troublesome, especially if the organization supports multiple streams of development. As more MISRA artifacts are published, maintaining a single source of truth becomes nearly impossible, as documents sprawl throughout file shares, desktops, and even bookshelves.

Consistency and correctness in compliance activities comes by establishing one repository for MISRA documentation. A static analysis tool is the ideal choice as it keeps track of MISRA versions across all development streams and stays current with the latest updates. Rather than rely on manual tools or a developer's hard drive, a static analysis tool enables the single source of compliance truth for everyone.

4. Reduce risks in new development

Those new to MISRA have a difficult challenge, as teams must plan, test, trace, report, and collect certification artifacts that match the expectations of long-running certifying authorities and customers. From EV start-ups to mature device manufacturers branching into new markets, static analysis tools provide a head start towards robust safety and security compliance checks under MISRA C.

Start your path towards MISRA C now

MISRA C is an integral component of any embedded software certification process and the adoption of static analysis tools is a valuable investment towards meeting compliance goals. By advocating for the adoption of these tools, embedded software developers help improve the safety, security, reliability of their code, while also demonstrating a commitment to higher-level business objectives.

Ever since its inception, LDRA has played a role in the creation and ongoing development of MISRA C and has strong representation on the MISRA C Working Group responsible for its evolution. The LDRA static analysis tools reflect this collaboration through comprehensive and up-to-date support for any organization's compliance activities.

To learn more about how LDRA can help you on the path to MISRA C compliance, visit ldra.com/misra/ or contact us: ldra.com/contact-us/

www.ldra.com

LDRA UK & Worldwide
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: +44 (0)151 649 9300
e-mail: info@ldra.com

LDRA Technology Inc.
2540 King Arthur Blvd, 3rd Floor, 12th Main Lewisville Texas 75056
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit B-3, Third floor Tower B, Golden Enclave
HAL Airport Road Bengaluru 560017
Tel: +91 80 4080 8707
e-mail: india@ldra.com

ISO 9001:2015 Quality Management FM 26376
FUNKTIONALE SICHERHEIT GEPRÜFT FUNCTIONAL SAFETY APPROVED

MISRA C: 23 White Paper v2.1 03/23