LDRA
Software Technology

Technical White Paper

MISRA C: 2012 – The evolution of a coding standard

www.ldra.com

Contents

Introduction

MISRA C is a language subset of the C programming language that is developed and maintained by the Motor Industry Software Reliability Association (MISRA). It is colloquially referred to as a "coding standard" – but never by MISRA themselves. Originally designed to promote the use of the C language in safety-critical embedded applications within the motor industry, the original version, MISRA C:1998[1], was released in 1998 to target C90.

Over the years, MISRA C has gained widespread acceptance for safety-, security-, life-, and mission-critical applications in aerospace, telecom, medical devices, defence, railway, and other industries. Misra C:2004[2] was renamed to reflect that more widespread use, and included a host of extensions and improvements to the original version.

The third revision of the standard, MISRA C:2012[3], was first released in early 2013 to provide support for ISO 9899:1999[4] (C99) while retaining support for C90. The design remit to allow programmers to spend more time coding and less on compliance efforts has been retained throughout subsequent amendments and revisions, which also reflect the evolution of the C language and the environments in which it is applied.

LDRA have been influential in the creation and ongoing development of MISRA C, and has had strong representation on the working group responsible for it throughout its evolution. This document reflects that unique insight into the creation of the standard, and into the incremental changes resulting from a policy of continuous improvement.

MISRA C:2012, first published 2013

When MISRA C:2012 superseded Misra C:2004 in early 2013, rules were made more precise to avoid the prevention of reasonable uses or behaviours that have no undesirable consequences. In addition, developers gained better guidance on rules enforcement, such as whether a rule defines a general behaviour across the project or only in specific cases.

A design objective for the 2012 version was to ensure that, wherever possible, the thrust of each rule would allow it to be "decidable"– so that an analysis tool can always determine compliance or non-compliance, as opposed to "undecidable" – generally due to pointer or data values affecting control flow (Figure 1). Hyperlinks to the offending code and to descriptions of the violations help in their resolution, and the clear reporting illustrated here complements the user–friendly nature of the standard.). Undecidable rules can result in false-positive or false-negative test results because the tool has inadequate information available to it during analysis. This improvement significantly reduces manual code-review requirements.

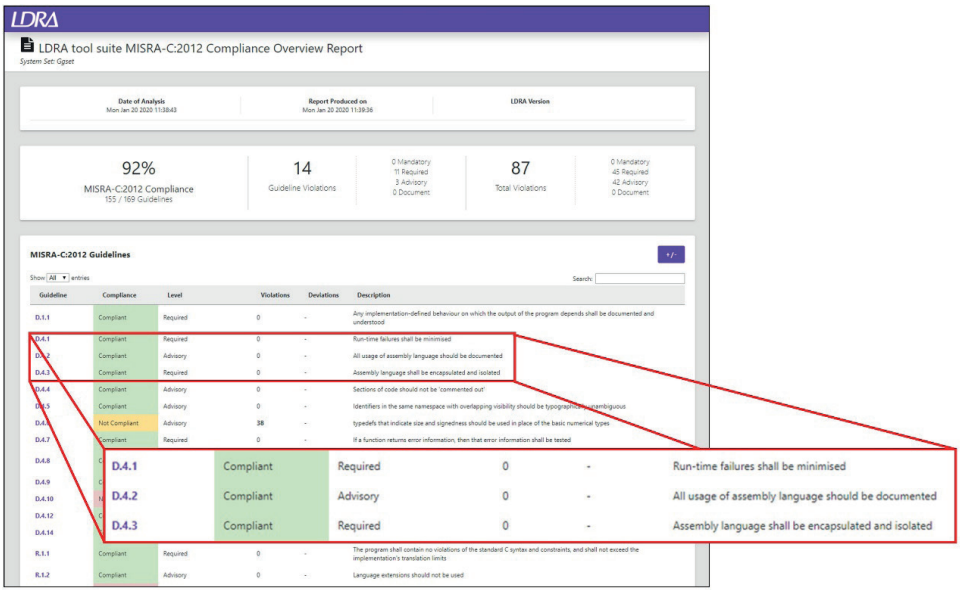

Figure 1: The LDRA tool suite can identify "decidable" MISRA C:2012 rule violations and help in the checking of some "undecidable" rules. Hyperlinks to the offending code and to descriptions of the violations help in their resolution, and the clear reporting illustrated here complements the user-friendly nature of the standard.

[1] Guidelines for the Use of the C Language in Vehicle Based Software, ISBN 978-0-9524156-6-4, April 1998, October 2002.
[2] Guidelines for the Use of the C Language in Critical Systems, ISBN 0 9524156 2 3 (paperback), ISBN 0 9524156 4 X (PDF), October 2004.
[3] Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.
[4] ISO/IEC 9899:1999 Programming languages — C

Developing the new version

There were some disadvantages to consider before deciding to develop this third MISRA C version. It would mean, for example, that developers would be required to learn a new version, and that tool vendors would need to develop supporting tools for it.

Ultimately, these disadvantages were minimized through a process of refinement in the face of feedback and experience from the earlier revisions. However, although the requirement to support C99 was the key motivator in the decision to develop a third version, the revisions went much further than simply adding new rules to cover any new C99 functionality.

In evolving Misra C:2004 into MISRA C:2012, a review of the rules followed some suitability "rules of thumb":

- It was to be made clear which versions of the C standard each rule applied to - C90, C99, or both.
- All instances of "Undefined" and "Unspecified" C language behaviour would be covered by a rule.
- All rules would be categorized as "Decidable" or "Undecidable". "Undecidable" rules were to be assessed further to see whether they could be repositioned a little to become "Decidable". (Most remaining "Undecidable" rules deal with pointer or data values which affect control flow).
- Rule enforcement procedure would be described by a single line "Headline text", complemented in more detail by means of "Normative text".
- Rule precision was to be such that only problem uses were to be restricted, and hence that reasonable uses were not.

In general, the net result was a document which was much more educational and informative than its more prescriptive predecessors.

Examples of specific beneficial changes

This evolution of the MISRA C standard into a more user-friendly document could be seen in a number of different ways.

Introduction of "Mandatory" rules

In addition to existing "Required" and "Advisory" rules, the "Mandatory" category was introduced to identify rules which must NEVER be broken (Figure 2). The other categories can be broken with varying degrees of justification required.

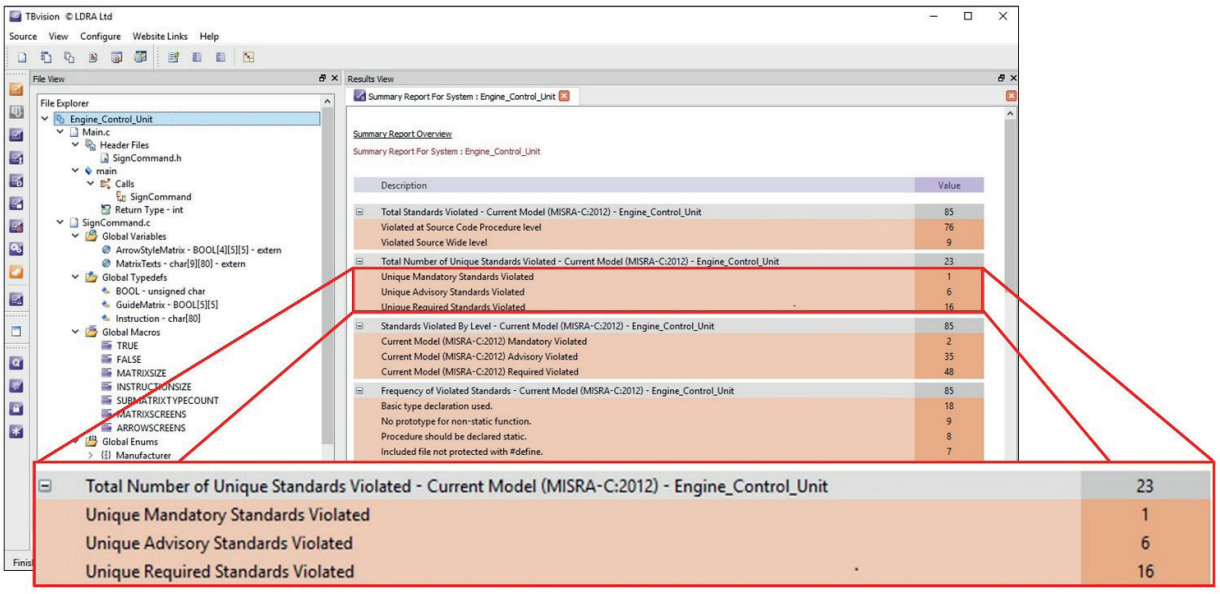

Figure 2: In its first version, MISRA C:2012 introduced a "Mandatory" category for rules which must never be broken to complement the existing "Required" and "Advisory" categories. As illustrated, TBvision can summarize information on any identified violations.

For example:

- Rule 22.2 A block of memory shall only be freed if it was allocated by means of a Standard Library function.

  There have been instances of developers freeing memory automatically allocated to variables for use elsewhere. This remains possible and is legitimate C syntax, but is dangerous and unnecessary. The rule is designed to prevent developers being "too clever for their own good."

  void fn (void) { int32_t a; free (&a); / Non-compliant - a does not point to allocated storage */*
  }

- Rule 22.4 There shall be no attempt to write to a stream which has been opened as read-only. ISO/IEC 9899 (i.e., the C language standard) does not specify the behaviour if an attempt is made to write to a read-only stream. For this reason it is considered unsafe to write to a read-only stream.

  This rule is designed to stop mistakes. There is no benefit from attempting to do this and so it is unlikely to be deliberate.

Variation of existing rules

Existing Misra C:2004 rules were revisited, refined, adjusted and justified. Compared to its predecessors, MISRA C:2012 enhances the established concept of "rationale" descriptions of why each rule is a good idea. This approach is beneficial both to those looking to implement MISRA C:2012 in its entirety, and those looking to use it as the basis for in-house standards (Figure 3).

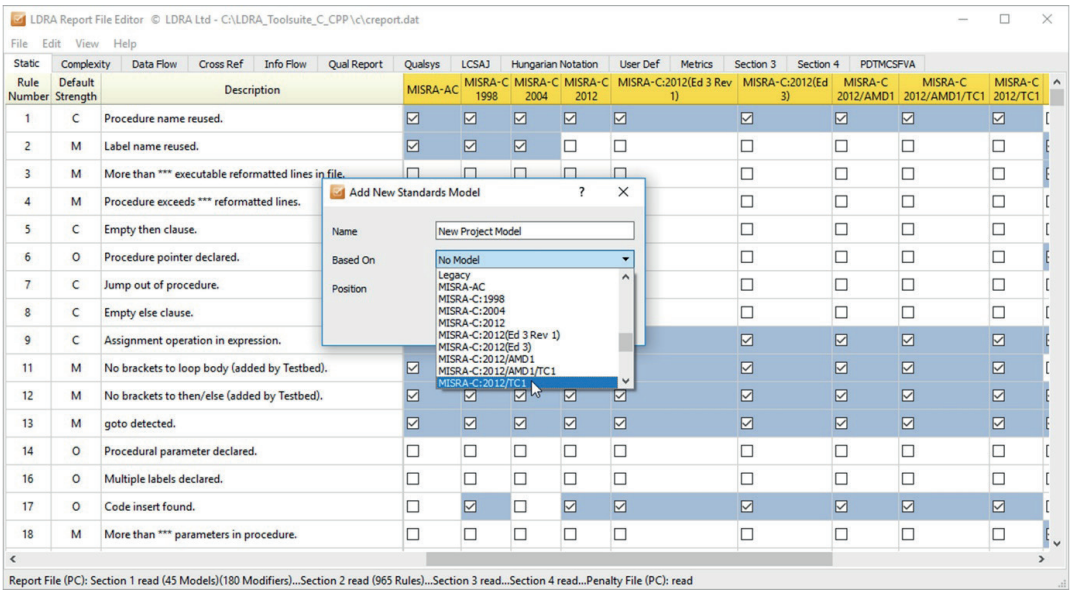

Figure 3: LDRA static analysis tools allow MISRA standards to be used as the basis for company or project specific rule sets, so that in-house rules can be added and selected MISRA rules disabled.

For example:

- There are places where the use of the "goto" statement is justified, but all too often in the past it has been used to patch up woolly thinking or an ill-defined algorithm.

  Suppose there is an emergency situation in a process control application. Is it really better to set a flag and check it later in the algorithm than to take a direct route via a goto?

  Rule 15.1 The goto statement should not be used became advisory rather than required, and there are an additional two required rules to narrow down the circumstances under which it is acceptable, vis.

  Rule 15.2 The goto statement shall jump to a label declared later in the same function.

  and

  Rule 15.3 Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

- *Rule 12.1 The precedence of operators within expressions should be made explicit* replaces a rule from the MISRA C:2004 vis. *Limited dependence should be placed on C's operator precedence rules in expressions.*

  The improved precision both in the wording and in the form of a decision table provided with MISRA C:2012 has led to both users and (importantly) tool developers having a more consistent interpretation.

Refinement and clarification of an approach to implement specific behaviour

From the perspective of individuals involved with the hardware/software interface there were several revisions that made MISRA C:2012 better defined. For example:

- The directive on the use of typedefs was revised to "Advisory" from "Required".
  Dir 4.6 typedefs that indicate size and signedness should be used in place of the basic numerical types.

  For example, on a 32-bit C90 implementation the following definitions might be suitable:

  typedef signed char int8_t;
  typedef signed short int16_t;
  typedef signed int int32_t;
  typedef signed long int64_t;

  From the perspective of portability, the rationale debunks the possible false perception that adherence to this guideline guarantees portability because the size of the int type may determine whether or not an expression is subject to integral promotion. For example, an expression with type int16_t will not be promoted if int is implemented using 16 bits but will be promoted if int is implemented using 32 bits.

Distinction between "System Wide" analysis and "Single Translation Unit" analysis

Although all "Decidable" rules are theoretically decidable, it does not follow that all compliance checkers can check all "Decidable" rules. For example, the more sophisticated tool suites are capable of system wide analysis (analogous to the work of a linker) whereas inferior tools offer only translation unit analysis (analogous to the compiler).

In recognition of this, rules are further categorized as requiring "System Wide" versus "Single Translation Unit" analysis, identifying those rules which will require an alternative approach to checking where a tool is in use which is capable of only the latter.

For example:

- *"Rule 22.3 The same file shall not be open for read and write access at the same time on different streams"* provides a good example of a rule requiring system wide analysis. Like rule 22.4 above, it is unlikely that anyone would do this deliberately, but a raised violation will help to prevent mistakes. A tool can only help confirm or deny a violation if it can reference all source code in that system through system wide analysis.

Summary

MISRA C:2012 was an evolution of the earlier MISRA C standards, designed to ensure that its contents felt familiar for existing MISRA users while providing additional benefits for newcomers too. It benefitted not only from the addition of rules to accommodate C99 functionality but also from improved rule precision, better rule categorisation, and more comprehensive rule explanation to educate as well as instruct.

MISRA C:2012 Amendment 1, first published 2016

By 2016, the MISRA C guidelines had long been designed to be suitable for developing any highly critical application, including those that were security critical and although the MISRA language subset had strong ties to the safety-critical market, it has become a proven approach for coding best practices for any embedded application. As the embedded software industry has evolved, so have the threats to lives, equipment, and businesses. Although MISRA guidelines have always been designed to help developers write high-quality code, which is by nature more safe and secure. But increased industry awareness of security risks promoted the MISRA C working group to review the guidelines with that specific benefit in mind.

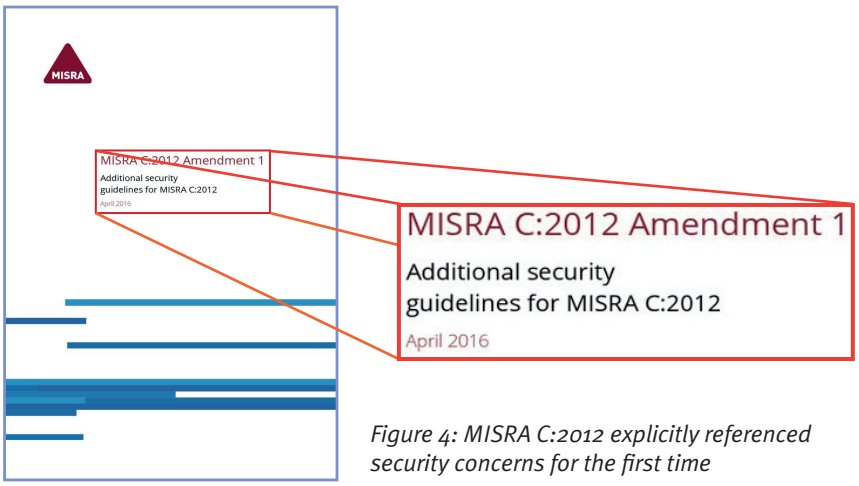

Figure 4: MISRA C:2012 explicitly referenced security concerns for the first time

Security risks had increased significantly with the advent of the Internet of Things. Secure application code may only be one component of the defence-in-depth strategy needed to develop a secure connected device, but it is nevertheless an important part if vulnerabilities are to be minimized.

MISRA C security amendments

After the publication of MISRA C:2012, the WG14[5] committee responsible for maintaining the C standard published the ISO/IEC 17961:2013 C Language Security Guidelines[6], designed to limit the use of the C language to a subset excluding the more vulnerable features of the language. The intention was for all rules to be enforceable using static analysis such that their detection could be automated without generating excessive false positives.

It was in response to ISO/IEC 17961 requirements that the MISRA committee developed MISRA C:2012 – Addendum 2[7], highlighting which of the 46 C Secure rules are covered by the original MISRA C: 2012 guidelines. As part of the same initiative, the committee also developed MISRA C:2012 – Addendum 3[8], making a similar mapping of MISRA C to the CERT C recommendations.

MISRA C:2012 Amendment 1[9] was written to further ensure complete coverage of the C Secure rules, and to be used as an extension to MISRA C:2012.

It establishes 14 new guidelines for secure C coding to improve the coverage of the concerns highlighted by the ISO C Secure Guidelines including, for example, issues pertaining to the use of "untrustworthy" data—a well-known security vulnerability. By following the additional guidelines, developers can more thoroughly analyse their code and can assure regulatory authorities that they have adopted best practice. This is becoming critical in many fields of endeavour including the automotive industry, the Industrial Internet of Things (IIoT), and the medical device sector – in short, wherever security threats have led to OEM demands for developers to prove that their software meets the highest standards for security as well as safety.

Insecure coding examples and related rules

To put the amendment into context, it is useful to review examples of where the additional rules apply.

Example 1: Rule 12.5

Rule 12.5 states, "The sizeof operator shall not have an operand which is a function parameter declared as "array of type"

Many developers use the "sizeof" operator to calculate the size of an array. In a normal scenario that works fine. But when that approach is used on an array passed as a function parameter, that parameter is passed as a "pointer to type." Consequently the attempt to calculate the number of elements usually returns an incorrect value, possibly resulting in an array bound being exceeded (Figure 5).

[5] International standardization working group for the programming language C JTC1/SC22/WG14 http://www.open-std.org/jtc1/sc22/wg14/
[6] ISO/IEC TS 17961:2013 Information technology -- Programming languages, their environments and system software interfaces -- C secure coding rules
[7] MISRA C:2012 - Addendum 2: Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure", ISBN 978-906400-18-7 (PDF), Second Edition, January 2018.
[8] MISRA C:2012 - Addendum 3: Coverage of MISRA C:2012 against CERT C, ISBN 978-906400-19-4 (PDF), January 2018.
[9] MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012, ISBN 978-906400-16-3 (PDF), April 2016.

```c
#include <stddef.h>
#include <stdint.h>

static int32_t calculate_threshold (int32_t A[ 2 ])
{
   size_t size = sizeof ( A ) / sizeof ( A[ 0 ] );
   ...
   return A[ size - 2 ];
}

int main (void)
{
   int32_t A[] = { 1, 2 };
   ...
   return calculate_threshold ( A );
}
```

Figure 5: sizeof source code example

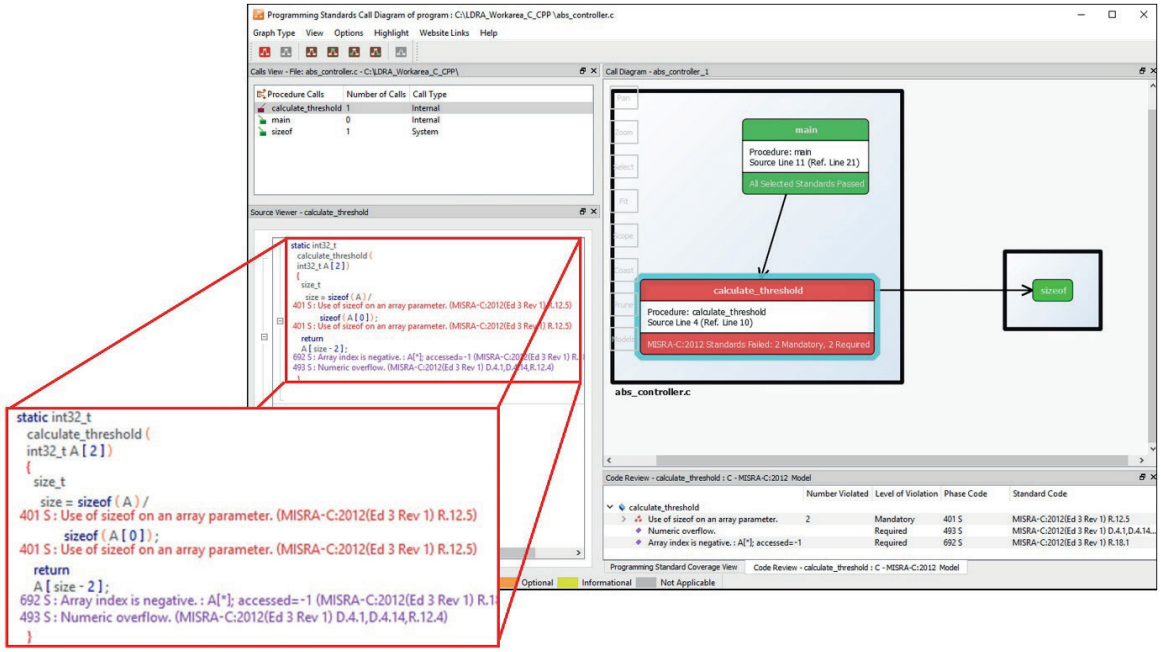

Figure 6: The TBvision component of the LDRA tool suite detects the MISRA C:2012 rule violation for the "sizeof" operator example

Example 2: Rule 22.7

Rule 22.7 states "The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF."

An "EOF" (End Of File) return value from standard library functions is used to indicate that the relevant stream has either reached the end of the file, or an error has occurred in reading from or writing to that file. The macro "EOF" is defined as an "int" with a negative value.

If the "EOF" value is captured in a variable of incorrect type, then it may become indistinguishable from a valid character code (Figure 7). It is therefore important to use an "int" to store the return code from such functions such as "getchar()" or "fgetc()", and to avoid because the common practice of storing the result in a char.

```c
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>

char measure_throttle_input (void)
{
    char ch;
    ch = ( char ) getchar ( );
    if (EOF != ( int ) ch)
    {
        /* Not Compliant - getchar returns an int which is cast to a narrower type */
    }

    return ch;
}

int main (void)
{
    char input = measure_throttle_input (void);

    return 0;
}
```

Figure 7: EOF source code example

Automatic detection of rule violation at an early stage

Peer reviews represent a traditional approach to enforcing adherence to such guidelines, and whilst they still have an important part to play, automating the more tedious checks using tools is far more efficient, less prone to error, repeatable, and demonstrable (Figure 8).

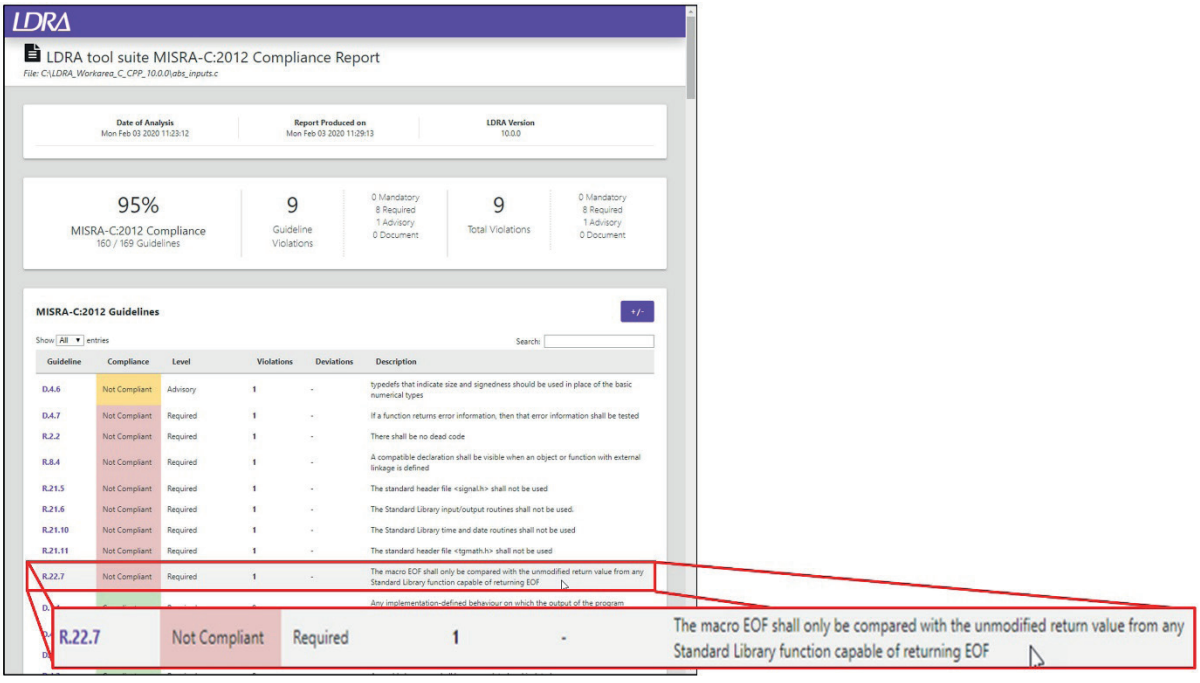

Figure 8: The TBvision component of the LDRA tool reports the Rule 22.7 (EOF comparison with char) violation in the source code

Summary

Best practise for the development of either safety or security critical code is to apply a formal software development process, starting with a set of requirements and tracing those requirements through to executable code. Even so, undefined, unspecified and implementation-defined behaviours within the C language can lead to safety or security failures in the resulting code base. And data handling errors such as invalid values, domain violations, tainted data, and leaking of confidential information can prevent both safety and security objectives from being realized.

MISRA C:2012 is not the only coding standard option for those with a need to develop secure code. For example, the correct application of either CERT C or MISRA C:2012 will certainly result in more secure code than if neither were to be applied. However, MISRA C's primary focus on critical embedded applications arguable makes it more suitable for those developments, and less error prone as a result of its stricter, more decidable rules. Conversely, there is an argument for using the CERT C standard if the application is not critical but is to be adapted for connection to the internet for the first time. The retrospective application of CERT C might then be a pragmatic choice to make.

MISRA Compliance 2016, first published 2016

Test tools are designed to highlight MISRA violations in C code. But whilst identifying and addressing violations flagged by an analysis tool may well be a significant challenge for individual developers, it represents only one part of the compliance process for the development team as a whole.

In fact, it comes as a surprise to many that the MISRA C:2012 document contains six chapters of guidance before the guidelines themselves are defined!

Stepping back from the detailed analysis of violations, it is easy to see bigger questions about the overall process. MISRA's document MISRA Compliance 2016[10] receives much less press coverage than the language subsets themselves, but it is invaluable in understanding how the information highlighted by static analysis relates to the bigger picture of a MISRA compliant application.

It would be easy to misunderstand the nature of MISRA compliance, and to assume that a minimized violation count ensures optimized application safety and security. But for the concept of compliance to have credibility and for the guidelines to be effective, MISRA Guidelines need to be applied within a framework that leverages the advantages of the compliant code and manages any necessary deviations.

A flavour of MISRA Compliance principles

The following five examples typify the principles outlined in the MISRA Compliance document itself, which reflect a little technical wisdom and a lot of common sense.

MISRA Compliance requires a documented software development process

MISRA Guidelines are intended for use within the framework of a formal software development process. Such a process will ensure complete, unambiguous and correct software requirements, and that all and only those requirements are reflected by the artefacts created at each phase of the development lifecycle.

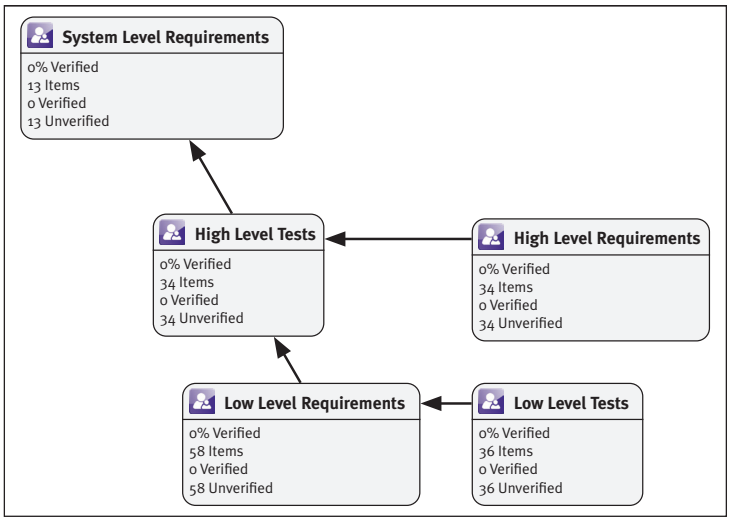

Figure 9: Structured development lifecycle is essential to MISRA compliance, as illustrated by the "Uniview" in the TBmanager component of the LDRA tool suite.

If code exhibits no violations but doesn't fulfil its required function, then it remains poor code!

[10] MISRA Compliance 2016: Achieving compliance with MISRA coding guidelines, ISBN 978-906400-13-2 (PDF), April 2016.

Not all MISRA Guidelines can be checked by analysis tools

1. The MISRA C:2012 Guidelines introduced a system under which each guideline is classified as either a rule or a directive.
   Typically, rules are sufficiently well defined to be checked by an automated tool, whereas directives are likely to be a little more subjective. 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".

2. In MISRA C:2012, some rules are labelled "Undecidable," meaning that it is fundamentally impossible to have a method that can, in general, say for sure whether a violation is present or not. A tool might warn of the potential for a problem, or it may not. Either way, some level of manual intervention is required.

3. Not all tools are the same. Some will claim more coverage of the rules than others, and some will be incapable of more subtle infringements. A tool showing "no violations" may really be saying "no violations, except the ones I've not spotted".

One Oxford Dictionary definition of "tool" is "A thing used to help perform a job". Tools help – they don't do the job for you.

Guidelines are only useful if there is a plan for their enforcement

For most guidelines, the easiest, most reliable, and most cost-effective means for enforcement is to use a static analysis tool, the compiler, or a combination of the two.

For these guidelines it is important to ensure that the tool(s) to be used has been proven to be appropriate, and that its type and version are specified and fixed.

There must also be enforcement plans in place for those guidelines that require some manual verification.

"Deviation" is not a dirty word

For any real-life embedded application, it is very likely that some violations will be unavoidable. The management of these violations must be authorized through a clearly defined process and supported by appropriate "deviation record" documentation if any claim of compliance for the resulting application is to be credible.

These deviation records need to include the guideline(s) violated, the justification for this/these violation(s), the circumstances under which the deviation applies, and where in the code base it applies.

Adopted code can't be ignored

Much of the documentation and many of the standards relating to functionally safe and secure embedded software starts from an assumption of a "green field" project. In real life, developers will be required to leverage in-house legacy code or third-party code such as device drivers, mathematical libraries, or graphical libraries.

Although it is clearly impractical to retrospectively apply MISRA guidelines to such code, for MISRA compliance to be claimed it is important to ensure that this so-called "adopted code" cannot compromise the safety and security of the system as a whole.

Summary

Aside from the guidelines in the language subsets themselves, many of the underlying requirements to be met before MISRA compliance can be justifiably claimed boil down to a common sense approach, and a dedication to "doing it right".

MISRA C:2012 (3rd Edition, 1st Revision), first published 2019

MISRA C:2012 is the 3rd edition of the MISRA C Guidelines. This first revision of MISRA C:2012 represented a consolidation of the original MISRA C:2012 document with the security guidelines established in MISRA C:2012 Amendment 1, and the corrections outlined in MISRA C:2012 Technical Corrigendum 1[11] . It contained nothing new, although the naming convention caused some confusion (Figure 10).

Figure 10: Explaining the naming convention of MISRA C:2012 (3rd Edition, 1st Revision)

MISRA C:2012 Amendment 2, first published 2020

This second amendment further enhanced the document by bringing large parts of C11 and C18 into the scope of MISRA C, and by introducing a framework for further guidance in the future and new versions of the C language.

MISRA C:2012 Amendment 2 sees the updated compliance guidance become a mandatory part of the MISRA framework, initially for MISRA C:2012 and subsequently for the upcoming MISRA C++ release. As part of that process, some compliance related elements of MISRA C have been relocated and integrated within the MISRA Compliance document.

Summary and conclusions

MISRA C:2012 is subject to a policy of continuous improvement. Whether reflecting changes in the C language or changes in the nature of the applications to which it is applied, each of the revisions, amendments and supplementary documents discussed here refines and enhances the core principles that underpinned the document upon its release in 2013, and still underpin it today.

Ever since its inception, LDRA has been influential in the creation and ongoing development of MISRA C, and has had strong representation on the working group responsible for it throughout its evolution, a position that is reflected in support for the standard in LDRA's static analysis tools.

[11] MISRA C:2012 - Technical Corrigendum 1: Technical clarification of MISRA C:2012, ISBN 978-906400-17-0 (PDF), June 2017.

Works cited

Guidelines for the Use of the C Language in Vehicle Based Software, ISBN 978-0-9524156-6-4, April 1998, October 2002.

Guidelines for the Use of the C Language in Critical Systems, ISBN 0 9524156 2 3 (paperback), ISBN 0 9524156 4 X (PDF), October 2004.

Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

ISO/IEC 9899:1999 Programming languages — C

International standardization working group for the programming language C JTC1/SC22/WG14 http://www.open-std.org/jtc1/sc22/wg14/

ISO/IEC TS 17961:2013 Information technology -- Programming languages, their environments and system software interfaces -- C secure coding rules

MISRA C:2012 - Addendum 2: Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 “C Secure”, ISBN 978-906400-18-7 (PDF), Second Edition, January 2018.

MISRA C:2012 - Addendum 3: Coverage of MISRA C:2012 against CERT C, ISBN 978-906400-19-4 (PDF), January 2018.

MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012, ISBN 978-906400-16-3 (PDF), April 2016.

MISRA Compliance 2016: Achieving compliance with MISRA coding guidelines, ISBN 978-906400-13-2 (PDF), April 2016.

MISRA C:2012 - Technical Corrigendum 1: Technical clarification of MISRA C:2012, ISBN 978-906400-17-0 (PDF), June 2017.

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