LDRA
Software Technology

Technical Briefing

Getting to grips with MISRA C:2012

www.ldra.com

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.

What are coding standards?

In the world of critical embedded software, there are two kinds of standards to be concerned with. The first kind can be thought of as a "process standard" – guidelines and rules that dictate how to go about writing your embedded software (amongst other things) so that errors are minimized. The more critical the application has to be, the more demanding the rules to be followed.

Most of those process standards are concerned with functional safety, and there are many standards with slight variations between industrial sectors (so DO-178C[5] applies to commercial aircraft and ISO 26262 applies to cars, for example.) Increasingly these functional safety standards are being complemented by standards concerned with cybersecurity, so that ISO 26262[6] is complemented by SAE J3061[7].

The second kind of standard are collectively known as coding standards (more formally, in MISRA's case, language subsets). Again, there are many of these which serve similar but slightly different purposes and they apply at a different level to these process standards. In any high level language there are hundreds of instructions and constructs. Some of them are very easy to get wrong (especially in C and C++) so the coding standards were introduced to disallow the use of those error-prone functions and hence make the resulting code more likely to be error free.

[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
[5] "RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification", Prepared by SC-205, December 13, 2011
[6] ISO 26262 second edition Road vehicles – Functional safety https://www.iso.org/standard/68383.html
[7] Cybersecurity Guidebook For Cyber-Physical Vehicle Systems https://webstore.ansi.org/standards/sae/sae30612016j3061

MISRA Compliance

It would be easy to rush to reference the MISRA C standard, especially given that 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 and it is vitally important to get the foundations right.

Stepping back from the detailed analysis of violations, it is easy to see bigger questions about the overall process. With MISRA's document MISRA Compliance 2016[8] being updated in 2020 and also being made mandatory for MISRA C compliant developments, it is important to understand how the information highlighted by static analysis relates to the bigger picture of a MISRA compliant application.

Examples 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 1).

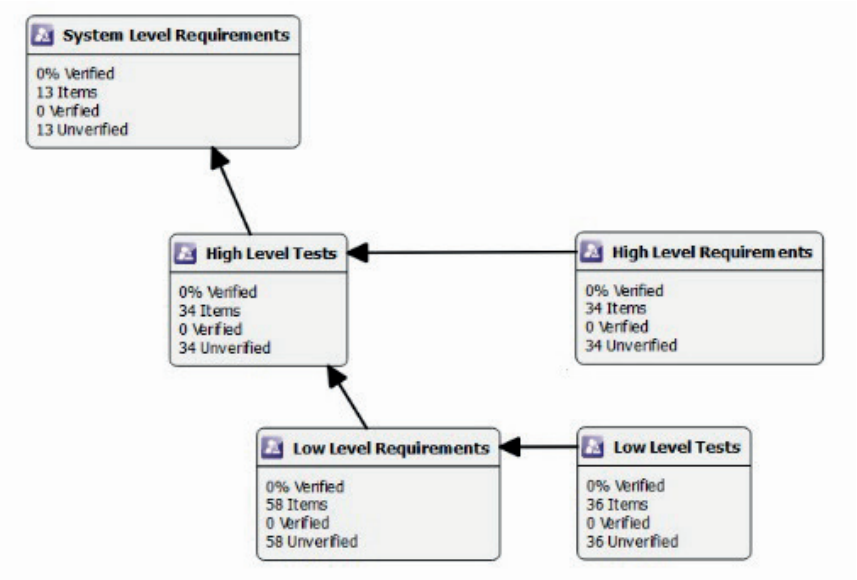

Figure 1: Structured development lifecycle is essential to MISRA compliance, as illustrated by the "Uniview" in the TBmanager component of the LDRA tool suite.

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

Not all MISRA Guidelines can be checked by analysis tools

Each MISRA 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*".

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

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.

It is also important to note that 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!" 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. It is important to ensure that any tool 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 usually impractical to retrospectively apply MISRA guidelines to such code, for MISRA compliance to be claimed it is important to ensure that this co-called "adopted code" cannot compromise the safety and security of the system as a whole.

MISRA C

Once the sound foundations described by the MISRA Compliance document are in place, the MISRA C guidelines can be considered. MISRA C was first published in 1998 with the intention of providing a "restricted subset of a standardized structured language" for automotive systems being developed to meet the requirements of Safety Integrity Level (SIL) 2 and above. Since then, the uptake and usage of MISRA C has far exceeded those limited ambitions and it has been adopted and used across a wide variety of industries and applications including the rail, aerospace, military and medical sectors.

The primary motivator behind the introduction of MISRA C:2012 was to support C99 which was gaining considerable support amongst the embedded fraternity. The policy of maintaining support for recent C language versions was underlined with the release of Amendment 2. Published in 2020, this second amendment brought 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.

Aside from the support for later C language variants there are several other characteristics of MISRA C:2012 that distinguish it from earlier versions. Here are some examples of why the net result is a document that is much more educational and informative than its more prescriptive predecessors.

Decidable rules

Wherever possible, the thrust of each rule allows it to be "decidable"—so that an analysis tool can always determine compliance or non-compliance. Undecidable rules can result in false-positive or false-negative test results because the tool has inadequate information available to it during analysis. This approach minimizes the need for manual code-review requirements.

Rule categorisation

MISRA C has always categorised guidelines. MISRA C:2012 supplemented the existing "Required" and "Advisory" categories with the "Mandatory" category to identify rules which must NEVER be broken (Figure 2). The other categories can be broken with varying degrees of justification required, so that "advisory" might be at a programmer's discretion while "required" would need to be formally supported and justified as part of the MISRA deviation process.

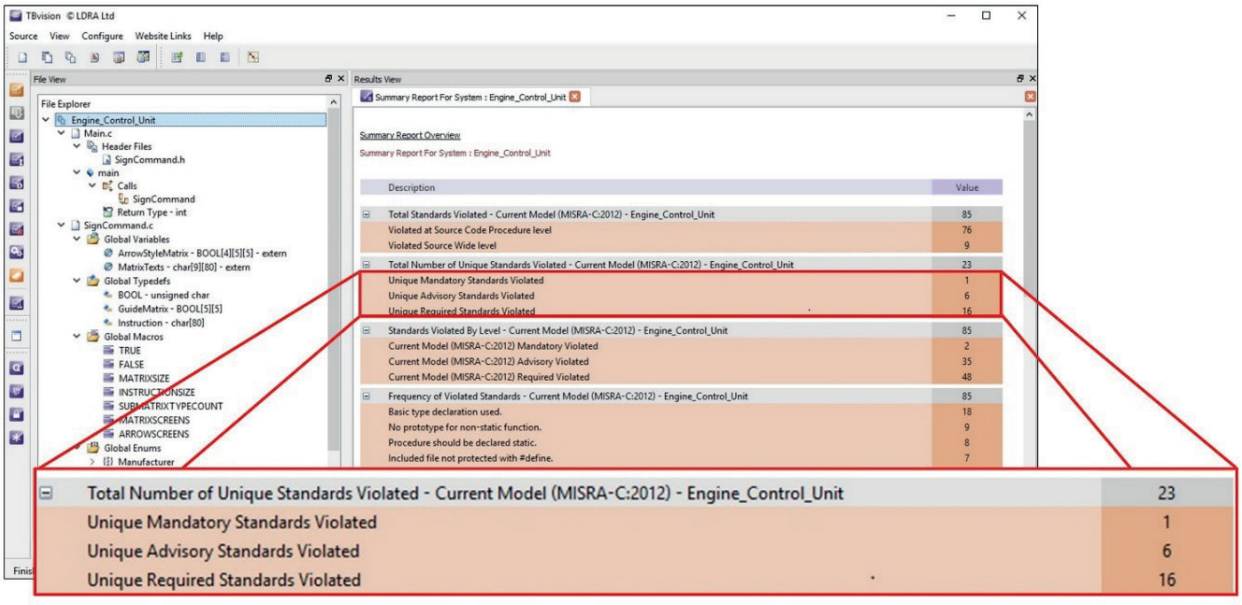

Figure 2: In its first version, MISRA C:2012 introduced a "Mandatory" category for rules which must never be broken. TBvision can summarize information on any identified violations.

For example, Rule 22.2 A block of memory shall only be freed if it was allocated by means of a Standard Library function. was introduced because 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.

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

Rationale descriptions

MISRA C:2012 extended the "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).

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. But 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?

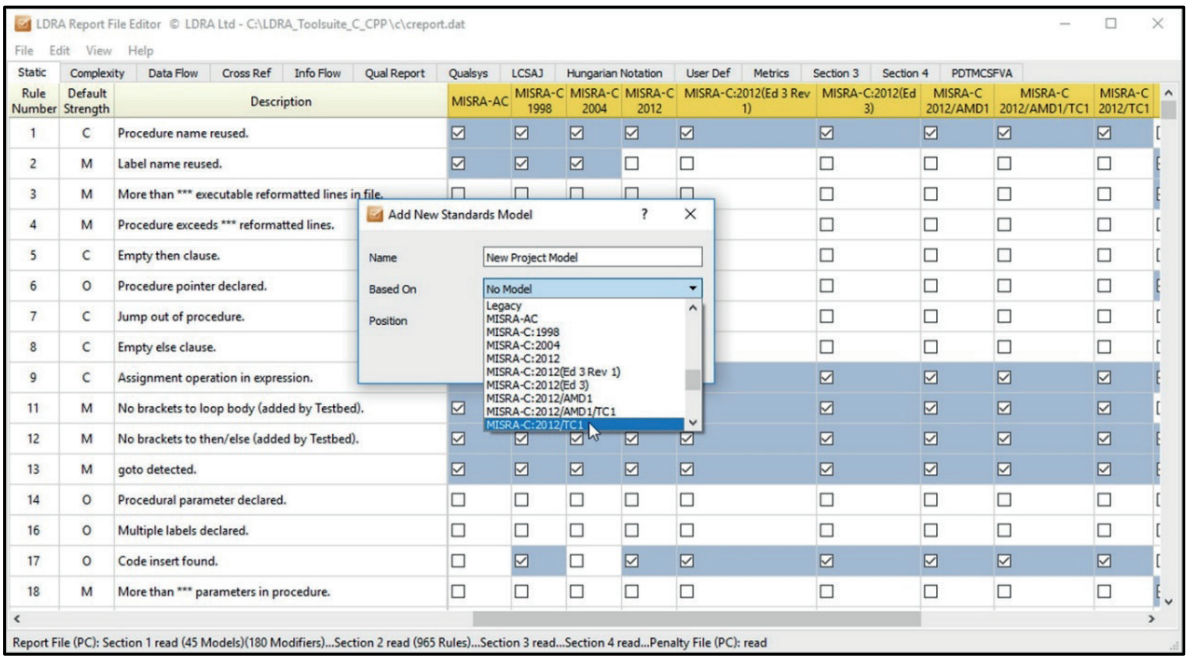

Figure 3: LDRA static analysis tools allow MISRA standards to be used as the basis for company or project specific rule sets, so that in-house rules can be added and selected MISRA rules disabled

To address that conundrum, *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.*

Implementation of specific behaviour

There were several revisions that made MISRA C:2012 better equipped to deal with hardware/software interface than its predecessors.

"System wide" versus "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).

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.

"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. It is unlikely that anyone would do this deliberately, but a raised violation will help to prevent mistakes.

MISRA C and cybersecurity

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.

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.

In acknowledgement of this changing environment, 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 established 14 new guidelines for secure C coding to improve the coverage of the concerns highlighted by the ISO C Secure Guidelines, and these rules were fully integrated into the MISRA C document in MISRA C:2012 (3rd Edition, 1st Revision), first published in 2019.

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 4).

```
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 4: sizeof source code example

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

A static analysis tool can be used to check for the use of such syntax (Figure 5).

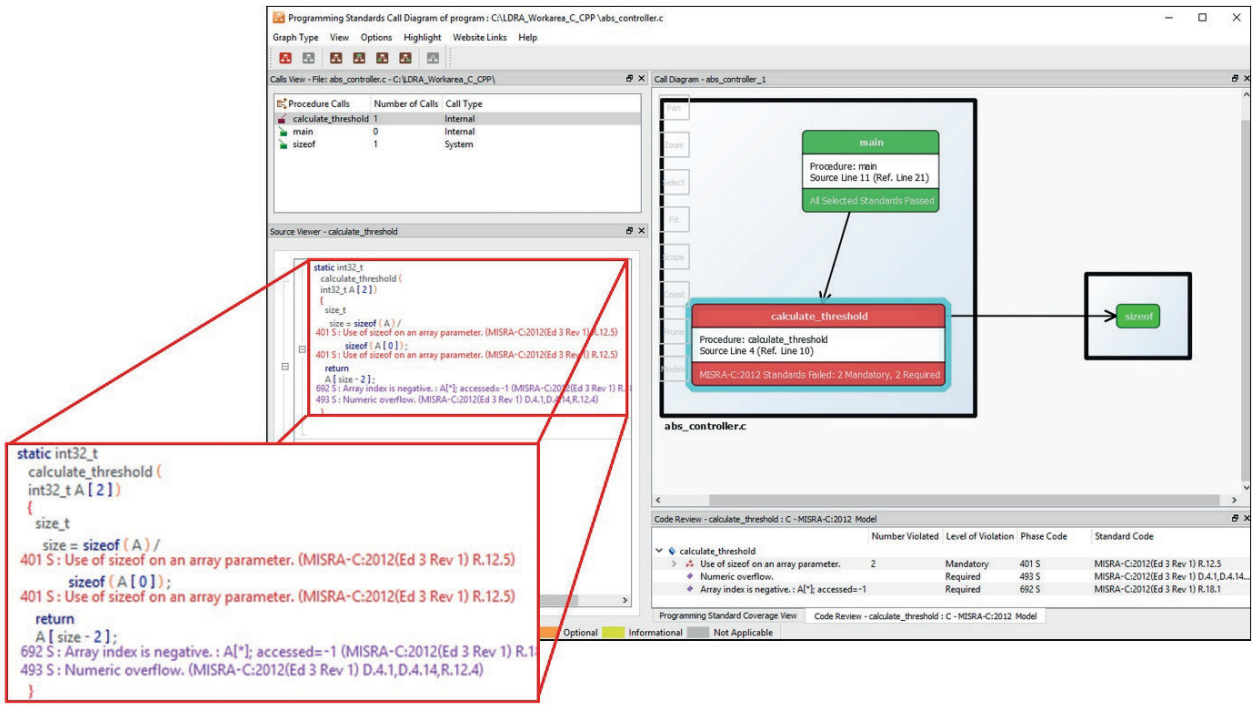

Figure 5: The TBvision component of the LDRA tool suite detects the MISRA C:2012 rule violation for the "sizeof" operator example

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 6). 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.

```
1  #include <stddef.h>
2  #include <stdint.h>
3  #include <stdio.h>
4
5  char measure_throttle_input (void)
6  {
7      char ch;
8      ch = ( char ) getchar ( );
9      if (EOF != ( int ) ch)
10     {
11         /* Not Compliant - getchar returns an int which is cast to a narrower type */
12     }
13
14     return ch;
15 }
16
17 int main (void)
18 {
19     char input = measure_throttle_input (void);
20
21     return 0;
22 }
```

Figure 6: 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 7).

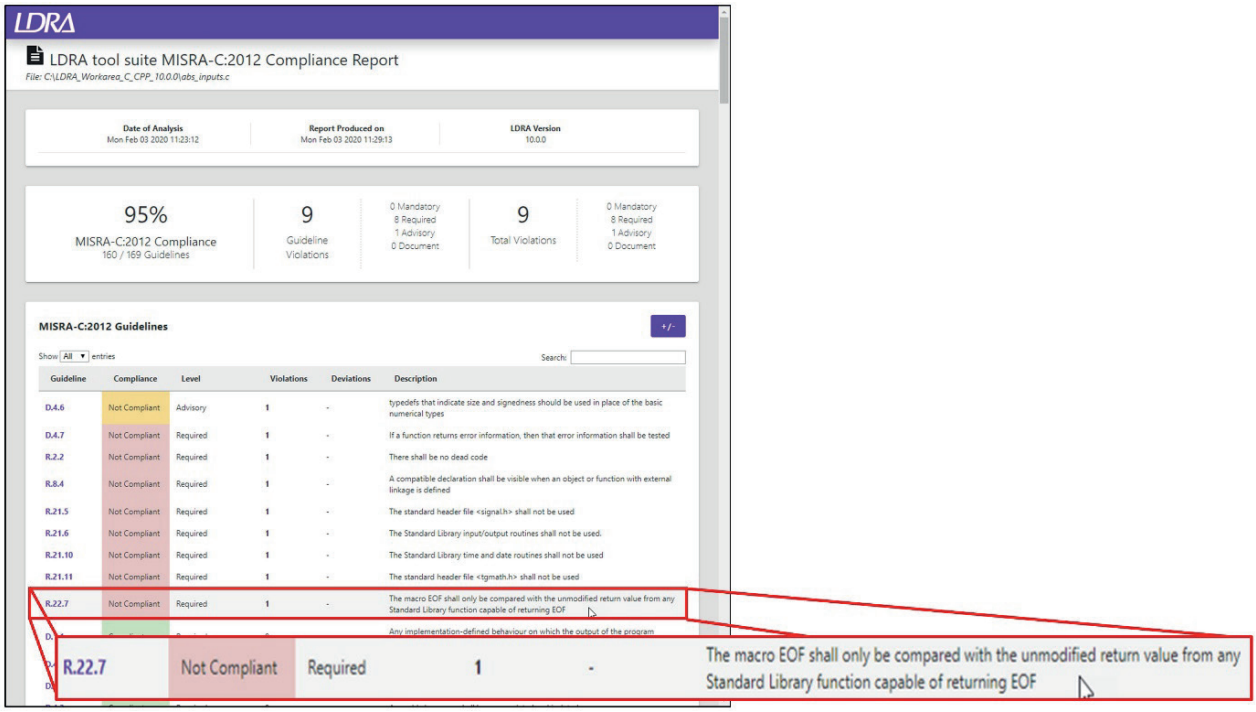

Figure 7: The TBvision component of the LDRA tool reports the Rule 22.7 (EOF comparison with char) violation in the source code

Summary and conclusions

In any high level language there are hundreds of instructions and constructs. Some of them are very easy to get wrong (especially in C and C++) so the coding standards were introduced to disallow the use of those error-prone functions and hence make the resulting code more likely to be error free. First released in 1998, MISRA C is one such coding standard.

The third version, 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 have refined and enhances the core principles that underpinned the document upon its release in 2013, and still underpin it today. The result is a document that has embraced both the changing demands of connected systems and the ongoing revisions and updates to the C language itself.

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, a position that is reflected in support for the standard in LDRA's static analysis tools.

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, Suite 228,
Lewisville, Texas 75056
United States
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit No B-3, 3rd Floor Tower B,
Golden Enclave, HAL Airport Road,
Bengaluru
560017
India
Tel: +91 80 4080 8707
e-mail: india@ldra.com

MISRA C Tech Briefing v4.0 03/20