MISRA Compliance:2020

Achieving compliance with MISRA Coding Guidelines

February 2020

First published February 2020 by HORIBA MIRA Limited
Watling Street
Nuneaton
Warwickshire
CV10 0TU
UK

www.misra.org.uk

ISBN 978-1-906400-26-2 PDF

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

MISRA Compliance:2020

Achieving compliance with MISRA Coding Guidelines

February 2020

MISRA Mission Statement

We provide world-leading, best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

MISRA is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety- and security-related electronic systems and other software-intensive applications. To this end, MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

Foreword

A lot of work has taken place within the MISRA C and MISRA C++ Working Groups since the initial release of MISRA Compliance in 2016, with one of the outcomes being that all future releases of MISRA Guidelines will mandate the use of MISRA Compliance.

Up to this point, the MISRA Guideline documents have all included content related to the various MISRA compliance activities. This update to MISRA Compliance enhances Section 2.2 (now titled "Framework") of Chapter 2 (now titled "The software development process"), completing the definition of what must be covered within the software development process when making a claim of MISRA compliance. This is mainly a "house-keeping" exercise, allowing the compliance-related content to be replaced by references to this document, ensuring consistency among the MISRA Guidelines whilst reducing the effort required in their maintenance.

Chris Tapp
Chairman, MISRA C++ Working Group

Acknowledgements

The MISRA consortium wishes to acknowledge the contribution made by the Japan Automobile Manufacturers Association to the preparation of this document.

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

Paul Burden	Independent Consultant (Formerly Programming Research Ltd)
Chris Tapp	LDRA Ltd / Keylevel Consultants Ltd

The MISRA consortium also wishes to acknowledge contributions from the following members of the MISRA C Working Group during the development and review process:

Andrew Banks	LDRA Ltd / Intuitive Consulting
Liz Whiting	LDRA Ltd

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process:

David Ward	HORIBA MIRA Ltd

Contents

1 Introduction

The MISRA language documents [1] and [2] ("The Guidelines") are compilations of guidelines for coding in the C [3] and C++ [4] languages. They are widely used in the development of critical software systems when the requirements of a quality standard must be met. Many software projects specify that code quality should be assured by meeting the requirements of The Guidelines. However, the meaning of the phrase "MISRA compliant" needs to be carefully defined.

In order for a claim of MISRA compliance to have meaning, it is necessary to establish:

- Use of a disciplined software development process;
- Exactly which guidelines are being applied;
- The effectiveness of the enforcement methods;
- The extent of any deviations from the guidelines;
- The status of any software components developed outside of the project.

The Guidelines recognize that, in some situations, it is unreasonable or even impossible to comply with a coding guideline and that it is necessary to deviate from its requirements. The freedom to deviate does not necessarily compromise claims of compliance, but it does carry with it great responsibility. In the absence of a disciplined development process, it is easy for that freedom to be abused. At best, that will undermine the credibility of any claims of MISRA compliance; at worst, it will compromise code quality, safety or security. It is therefore important to emphasize that a credible claim of compliance with The Guidelines can only be made when code is developed under a process which meets the principles laid out in this document.

The guidance given in this document supersedes the compliance, deviation and process requirements published previously in the various MISRA Guidelines.

Note: The various MISRA Guideline documents have been refined and revised over a number of years. This document uses examples and extracts from a number of them, but the issues discussed are equally relevant to all of The Guidelines.

2 The software development process

2.1 The development contract

A decision to adopt and apply MISRA Guidelines should be taken at the outset of a project. The application of MISRA Guidelines will typically be one requirement among many of a contractual agreement between two parties, the organization commissioning the project (the *acquirer*) and the organization developing the code (the *supplier*). These parties may be different commercial entities or they may simply be different departments within the same organization. Both will need to be actively involved in the task of assuring compliance with MISRA Guidelines. MISRA Compliance is not a rigidly defined concept and acceptance criteria will be a matter for negotiation. However there are some clear principles which must be observed in order for the concept of compliance to have credibility.

2.2 Framework

MISRA Guidelines are intended to be used within the framework of a documented software development process. They are of greatest benefit when a process is in place to ensure that, for example:

1. The software requirements, including any safety or security requirements, are complete, unambiguous and correct;
2. The design specifications reaching the coding phase are correct, consistent with the requirements and do not contain any other functionality;
3. The object modules produced by the compiler behave as specified in the corresponding designs;
4. The object modules have been tested, individually and together, to identify and eliminate errors.

Compliance with MISRA Guidelines must be an integral component of the code development phase and compliance requirements need to be satisfied before code is submitted for review or unit testing. A project that attempts to check for compliance late in its lifecycle is likely to spend a considerable amount of time in re-coding, re-reviewing and re-testing, and it is easy for this rework to introduce defects by accident. Code shall be checked for compliance as it is developed and not as part of a "tick-box" exercise to be completed as part of the final delivery phase.

If a project is building on code that already has a proven track record, then the benefits of gaining compliance by re-factoring existing code may be outweighed by the risks of introducing a defect. In such cases a judgement needs to be made based on the net benefit likely to be obtained.

Whilst this document does not define a complete software development process, there are some process activities that must be included in order to demonstrate the adoption of best practice when claiming MISRA compliance:

- Training;
- Style guide;
- Metrics;
- Tool management;
- Run-time behaviour.

All decisions made on these issues, including the reasons for those decisions, need to be documented, and appropriate records should be kept for any activities performed. Such documentation may then be included in a safety justification, if required. Appendix A provides a checklist which may be helpful in ensuring that documentation is produced for these items.

A full discussion of the requirements for software development processes is outside the scope of this document. Further information may be found in standards such as ISO/IEC/IEEE 12207 [5], IEC 61508 [6], ISO 26262 [7], EN 50128 [8], IEC 62304 [9] and DO-178 [10].

2.3 Training

In order to ensure an appropriate level of skill and competence on the part of those who produce the source code, formal training should be provided for:

- The use of the chosen programming language for embedded applications;
- The use of the chosen programming language for high-integrity, safety-related or security-related systems.

Since compilers and static analysis tools are complex pieces of software, consideration should also be given to providing training in their use. In the case of static analysis tools, it might be possible to obtain training in their use specifically in relation to the enforcement of MISRA Guidelines.

2.4 Style guide

An organization should enforce an in-house style guide to provide guidance on issues that do not directly affect the correctness of the code, but rather define a "house style" for the appearance of the source code. These issues are subjective and typically include:

- Code layout and use of indenting;
- Layout of braces "{ }" and block structures;
- Naming conventions;
- Use of comments;
- Inclusion of company name, copyright notice and other standard file header information.

Enforcement of the style guide requirements is outside the scope of this document.

See [14] for further information on style guides.

2.5 Metrics

The use of metrics is recommended by many software process standards as a means to identify code that may require additional review and testing effort; they can be used to prevent unwieldy and untestable code from being written by looking for values outside of established norms. However, the nature of the metrics being collected, and their corresponding thresholds, will be determined by the industry, organization and/or the nature of the project. This document, therefore, does not offer any guidance on software metrics.

For details of possible source code metrics see "Software Metrics: A Rigorous and Practical Approach" by Fenton and Bieman [15] and the MISRA report on Software Metrics [16].

Note: With a planned approach to development, the extra effort expended to achieve a balance between code size and cyclomatic complexity metric measures can be more than offset by the reduction in the time required to achieve high statement coverage during testing. See [16], [17].

The use of tools to collect metrics data is highly recommended. Many of the static analysis tools that may be used to enforce MISRA Guidelines also have the capability to produce metrics data.

2.6 Tool management

2.6.1 Selecting the compiler

In this document, the term "compiler", referred to as "the implementation" by the ISO standards [3], [4], means the compiler itself as well as any associated tools such as a linker, library manager and executable file format conversion tools.

The compiler selected for the project should meet the requirements of a conforming free-standing implementation for the chosen version of the language. It may exceed these requirements, for example by providing all the features of the language (a free-standing implementation need only provide a well-defined subset), or it might provide extensions as permitted by the language standard.

Ideally, confirmation that the compiler is indeed conforming should be supplied by the compiler developer, for example by providing details of the conformance tests that were run and the results that were obtained.

Sometimes, there may be a limited choice of compilers whose quality may not be known. If it proves difficult to obtain information from the compiler developers, the following steps could be taken to assist in the selection process:

- Checking that the compiler developer follows a suitable software development process;
- Reading reviews of the compiler and user experience reports;
- Reviewing the size of the user base and types of applications developed using the compiler;
- Performing and documenting independent validation testing, e.g. by using a conformance test suite or by compiling existing applications.

2.6.2 Selecting static analysis tools

When choosing a static analysis tool, it is clearly desirable that the tool enforces as many guidelines as possible. To this end, it is essential that the tool is capable of performing checks across the whole program, and not only within a single source file.

It is theoretically possible to check that source code complies with MISRA Guidelines by means of inspection alone. However, this is likely to be extremely time-consuming and error prone, and any realistic process for checking code against them will therefore involve the use of at least one static analysis tool.

All of the factors that apply to compiler selection apply also to the selection of static analysis tools, although the validation of analysis tools is a little different from that of compilers. An ideal static analysis tool would:

- Detect all violations of The Guidelines;
- Produce no "false positives", i.e. would only report genuine violations and would not report non-violations or possible violations.

For the reasons explained in Section 3.5, it is not, and never will be, possible to produce a static analysis tool that meets this ideal behaviour. The ability to detect the maximum number of violations possible, while minimizing the number of false positive messages, is therefore an important factor in choosing a tool.

There is a wide range of tools available with execution times ranging from seconds to days. Broadly speaking, tools that consume less time are more likely to produce false positives than those that consume large amounts of time. Consideration should also be given to the balance between analysis time and analysis precision during tool selection.

Analysis tools vary in their emphasis. Some might be general purpose, whereas others might focus on performing a thorough analysis of a subset of potential issues. Thus it might be necessary to use more than one tool in order to maximize the coverage of issues.

2.6.3 Tool validation

The compiler and the static analysis tool are generally seen as "trusted" processes, meaning that there is a certain level of reliance on the output of the tools. Measures must therefore be taken to ensure that this trust is not misplaced. Ideally, this should be achieved by the tool supplier running appropriate validation tests. Note that, while it is possible to use a validation suite to test a compiler for an embedded target, no formal validation scheme exists at the time of publication of this document. In addition, the tools should have been developed to a quality system capable of meeting the requirements of ISO 9001 [11] as assessed using ISO/IEC 90003 [12].

It should be possible for the tool supplier to show records of verification and validation activities together with change records that show a controlled development of the software. The tool supplier should have a mechanism for:

- Recording faults reported by the users;
- Notifying existing users of known faults;
- Correcting faults in future releases.

The size of the existing user base together with an inspection of the faults reported over the previous 6 to 12 months will give an indication of the stability of the tool.

It is often not possible to obtain this level of assurance from tool suppliers and, in these cases, the onus is on the developer to ensure that the tools are of adequate quality.

Some possible approaches the developer could adopt to gain confidence in the tools are:

- Perform some form of documented validation testing;
- Assess the software development process of the tool supplier;
- Review the performance of the tool to date.

The validation test could be performed by creating code examples to exercise the tools. For compilers this could consist of known good code from a previous application. For a static analysis tool, a set of code files should be written, each containing a violation of one guideline and together covering as many of The Guidelines as possible. For each test file, the static analysis tool should then find the non-conformant code. Although such tests would necessarily be limited, they would establish a basic level of tool performance.

It should be noted that validation testing of the compiler must be performed for the same set of compiler options, linker options and source library versions used when compiling the product code.

Tool vendors are often willing to assist in tool verification and may have a set of test files that can be used to allow independent evaluation of tool behaviour to be undertaken.

The tool developer may maintain a list of defects that are known to affect the tool along with any workarounds that are available. Knowledge of the contents of this list would clearly be advantageous before starting a project using that tool. If such a list is not available from the tool developer then a local list should be maintained whenever a defect or suspected defect is discovered and reported to the tool developer.

Note: some process standards, including IEC 61508 [6], ISO 26262 [7] and DO-178 [10], require the qualification of compilers and static analysis tools in some situations.

2.6.4 Understanding and configuring the compiler

The compiler may provide various options that control its behaviour. It is important to understand the effect of these options as their use, or non-use, might affect:

- The conformance of the compiler against applicable standards;
- The availability of language extensions;
- The availability of conditional language features;
- The resources, especially processing time and memory space, required by the program;
- The likelihood that a defect in the compiler will be exposed, such as might occur when complex highly-optimizing code transformations are performed.

If the compiler provides a choice between language version, it must be configured for the variant being used on the project. Similarly, if the compiler provides a choice of targets, it must be configured for the correct variant of the selected target.

It is important to understand the way in which the compiler implements those features of the language that are termed "implementation-defined" in the applicable standard. It is also important to understand the behaviour of any language extensions that the compiler may provide.

Reliance on language features identified as "conditional" within the applicable standard should be identified to ensure that they are provided by the compiler.

The compiler's optimization options should be reviewed and selected carefully in order to ensure that an appropriate balance between execution speed and code size has been obtained. Using more aggressive optimizations may increase the risk of exposing defects in the compiler.

Even if the compiler is not being used to ensure compliance with The Guidelines, it may produce messages during the translation process that indicate the presence of potential defects. If the compiler provides control over the number and nature of the messages it produces, these should be reviewed and an appropriate level selected.

2.6.5 Understanding and configuring the static analysis tool

The documentation for each static analysis tool being used on a project should be reviewed in order to understand:

- Which version(s) of the language are supported;
- How to configure the analyser to match the compiler's implementation-defined behaviour, e.g. sizes of the integer types;

- Which of The Guidelines the analyser is capable of checking (Section 3.3);
- Whether it is possible to configure the analyser to handle any language extensions that will be used;
- Whether it is possible to adjust the analyser's behaviour in order to achieve a different balance between analysis time and analysis precision.

If the static analyser provides a choice between language version, it must be configured for the variant being used on the project.

While a compiler is usually specific to a particular processor, or family of processors, static analysis tools tend to be general purpose. It is therefore important to configure each static analysis tool to reflect the implementation decisions made by the compiler. For example, static analysers need to know the sizes of the integer types.

If possible, the static analyser should be configured to support any language extensions that are supported by the compiler. If this is not possible then an alternative procedure will need to be put in place for checking that the code conforms to the extended language.

It may also be necessary to configure a static analyser in order to allow it to check certain aspects of The Guidelines. For example, if a project using the C language is not using `_Bool` to represent Boolean data, an analyser needs to be made aware of how the type is implemented in order to check some MISRA C guidelines.

Where an analysis tool has capabilities to perform checks beyond those required to enforce The Guidelines, it is recommended that the extra checks are used. For example, it may be possible to enable checks to enforce aspects of the style and metrics process activities identified in Section 2.4 and Section 2.5, above.

2.6.6 Run-time behaviour

The software development process should document the steps that will be taken to avoid run-time errors, and to demonstrate that they have been avoided. For example, it should include descriptions of the processes by which it is demonstrated and recorded that:

- The execution environment provides sufficient resources, especially processing time and stack space, for the program;
- Run-time errors, such as arithmetic overflow, are absent from areas of the program: for example by virtue of code that checks the ranges of inputs to a calculation.

3 Fundamental elements of compliance

3.1 Guideline classification

The MISRA C:2012 Guidelines introduced a system of classification under which a *guideline* is described as either a *rule* or a *directive*. Earlier editions of MISRA Guidelines used no such distinction, and, in fact, consist almost entirely of *rules*.

A *rule* is a *guideline* which imposes requirements on the source code which are complete, objective, unambiguous and independent of any process, design-documentation or functional requirement. Analysis tools are capable of checking compliance with *rules*, subject to the limitations described later in Section 3.5.

A *directive* is a *guideline* which is not defined with reference to the source code alone. Analysis tools may be able to assist in checking compliance, but a *directive* will also refer to, or impose requirements on processes, documentation or functional requirements. The requirements of a *directive* may also introduce a degree of subjective judgement and different tools will therefore sometimes place different interpretations on what constitutes a non-compliance.

The majority of *guidelines* within MISRA Guidelines are classified as *rules*.

3.2 Analysis scope

The *analysis scope* of each MISRA *rule* is described as either "Single Translation Unit" or "System". Many *rules* can be checked by examination of each translation unit in isolation. Some *rules* can only be fully checked by analysing the source code in the entire system.

For example, a *rule* such as "Every *switch* statement shall have a default value", is a Single Translation Unit rule. It can be reliably verified by analysing the source code in each translation unit in isolation.

On the other hand, a *rule* such as "An identifier with external linkage shall have exactly one external definition" is a System rule. It can only be verified with certainty by analysis of the entire body of source code.

3.3 The guideline enforcement plan

The task of ensuring that effective policies are in place to implement *guideline* enforcement is fundamental to the task of achieving compliance. For most *guidelines*, the easiest, most reliable and most cost-effective means of detecting *guideline violations* will be to use an analysis tool or tools, the compiler, or a combination of these. A manual review process may be required where a *guideline* cannot be completely checked by a tool.

A *guideline enforcement plan* (*GEP*) listing each *guideline* within The Guidelines shall be produced to indicate how compliance with the *guidelines* is to be checked. The *supplier* will make the *GEP* available to the *acquirer* so that the suitability and robustness of the checking that has been undertaken can be assessed. An example is provided below:

Guideline	Compilers		Analysis tools		Manual review
	'A'	'B'	'A'	'B'	
Dir 1.1					Procedure x
Dir 2.1	no errors	no errors			
...					
Rule 4.1			message 38		
Rule 4.2				warning 97	
Rule 5.1	warning 347				
...					
Rule 12.1				message 79	
Rule 12.2			message 432		Procedure y
Rule 12.3			message 103		
Rule 12.4				message 27	

Guideline enforcement plan example fragment

The following information shall be recorded in support of the *guideline enforcement plan*:

1. For any tool identified within the plan:
   1.1 The tool version;
   1.2 Data and/or configuration files used by the tool;
   1.3 Any options used when invoking the tool;
   1.4 Evidence proving that the tool is able to detect *violations* of the *guidelines* for which it is to check compliance.
2. Details of any manual process identified within the plan.

This information shall be made available to the *acquirer* on request.

3.4 Investigating messages

The messages produced by the compliance checking process, or by the translation process, fall into one of the following categories:

1. Correct diagnosis of a *violation*;
2. Diagnosis of a possible *violation*;
3. False diagnosis of a *violation*;
4. Diagnosis of a genuine issue that is not a *violation*.

The preferred remedy for any messages in category (1) is to correct the source code in order to make it compliant with The Guidelines. If it is undesirable or impossible to render the code compliant, then *guideline violations* may need to be authorized as shown in Section 4.

Any messages in the other categories should be investigated. Sometimes, the easiest and quickest solution will be to modify the source code to eliminate the message. However, this may not always be possible or desirable, in which case a record of the investigation should be kept. The purpose of the record is to:

- Category (2) — Explain why the code is compliant despite diagnosis of a possible *violation*;
- Category (3) — Explain and, if possible, obtain the tool developer's agreement that the tool diagnosis is incorrect;
- Category (4) — Justify why the message can reasonably be disregarded.

All records of such investigations should be reviewed and approved by an appropriately qualified technical authority.

3.5 Decidability

As discussed in Section 3.1, a *rule* is defined as a *guideline* for which compliance is dependent entirely on the source code and not on any design considerations or external documentation. *Rules* are therefore *guidelines* which are amenable to enforcement using static analysis and the role of an analysis tool will be to answer the compliance question "Does this code comply with this rule?". Unfortunately there are some *rules* for which an answer cannot be provided in all circumstances and *rules* have to be classified as either *decidable* or *undecidable*.

Decidable rules

A *rule* is *decidable* if it is always possible to answer the compliance question with an unequivocal "Yes" or "No" answer. *Decidable rules* are particularly effective because, providing the analysis tool is configured correctly and is free of defects, it is always possible to verify compliance with certainty.

Undecidable rules

A *rule* is *undecidable* if an analysis tool cannot guarantee to respond to the compliance question with a "Yes" or a "No" in every situation. There may be situations when an analysis tool can respond with a "Yes" or a "No"; but there will also be some situations where the only possible response is "Possibly" (i.e. compliance is uncertain).

A review of the theory of computation, on which the decidability classification is based, is beyond the scope of this document, but a *rule* is likely to be *undecidable* if detecting violations depends on run-time properties such as:

- The value that an object holds;
- Whether control reaches a particular point in the program.

Most *undecidable rules* need to be checked on a system-wide basis because, in the general case, information about the behaviour of other translation units will be needed. Consider, for example, the requirement to check a *rule* such as "The value of an object with automatic storage duration shall not be read before it has been set", as shown in the following:

```
extern void f ( uint16_t * p );

uint16_t g ( void )
{
  uint16_t x;    /* x is not given a value                                  */

  f ( &x );      /* f might modify the object pointed to by its parameter */

  return x;      /* x may or may not be unset                               */
}
```

It will not be possible to verify compliance in the function `g` without examining the behaviour of the function `f`, which may be defined in another translation unit.

Complying with undecidable rules

When a *rule* is *undecidable*, no analysis tool, however sophisticated, can guarantee to respond unequivocally to the question "Does this code comply with this rule?" in every situation. Depending on the nature of the code, a tool may be able to report "Yes" or "No", but in many cases it may only be able to report "Possibly". (Of course, the actual response will be expressed in different ways by different tools and the absence of a diagnostic does not necessarily imply a response of "Yes").

The nature of undecidability is such that the task of verifying compliance to an *undecidable rule* by static analysis alone is often impossible. Even in coding situations where the task is theoretically provable, the necessary analysis may be very complex and may require computing resources which are unmanageable. Tools vary widely in their capability to diagnose non-compliance to *undecidable rules*. What is important is the degree to which a tool is successful in distinguishing and reporting definite *violations* as well as possible *violations*, thereby managing to avoid both false positives and false negatives.

Particular attention should be paid to the process for reviewing compliance to *undecidable rules*. The fact that static analysis cannot ensure compliance in the general case does not mean that compliance cannot be guaranteed for any code. If an analysis tool is reliable in identifying possible *violations*, it may be feasible to eliminate such uncertainties by adopting conservative and defensive coding techniques at the development stage.

4 Deviations

4.1 The role of deviations

The notion of MISRA compliant code can readily be compromised unless there is a clear understanding between *supplier* and *acquirer* as to how the notion of compliance is to be interpreted. Compliance cannot be claimed for an organization, only for a project, and it is important to establish agreed processes and acceptance criteria at the outset of the project.

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*.

4.2 Deviation records

A *deviation record* should include the following information:

1. The *guideline(s)* being *violated*;
2. A concise description of the circumstances in which a *violation* is acceptable;
3. The reason why the *deviation* is required (see Section 4.4);
4. Background information to explain the context and the language issues;
5. A set of requirements to include any risk assessment procedures and precautions which must be observed.

In addition, it is important to be able to identify where the *deviation* is being applied. A *deviation* may be associated with a single *violation* or with a number of similar *violations* which conform to a common use case. Where there are multiple *violations* to consider, the *deviation record* may consist of a body of common documentation supported by a register of locations where the *deviation* is being applied. This register may be in the form of a detailed list of file names and line numbers, a more general description of the context in which the deviation is applied (e.g. in specific files or in the expansion of a specific macro) or define a unique identifier that can be used to tag one or more locations within the code. However, the ability to identify every instance where the *guideline* is *violated* will be essential in order to support a robust review process. It should also be noted that in the case of some *guidelines*, a single *violation* may be associated with two or more locations in the code. This would occur, for example, where conflicting declarations of the same entity are identified.

An example *deviation record* is provided in Appendix B.

4.3 Deviation permits

A principle which has emerged from the use of MISRA Guidelines over several years, is that many *deviations* are consistent with well-known use cases which occur widely. New use cases may be identified during the code development phase, but in practice the majority of *deviations* will conform to use cases which have been identified in the past. It is therefore possible for the documentation of many *deviations* to be simplified by allowing them to refer to an approved deviation use case, as defined in a *deviation permit*.

A *deviation permit* is not the same as a *deviation record* but it can supply much of the material which is required in a *deviation record*. A *deviation permit* defines a use case under which a *violation* may be justified and specifies the documentation and process requirements which must be supplied in the *deviation record*.

The effort associated with compiling and reviewing *deviations* can be substantial and it is a task that requires the input of experienced and well qualified staff. This effort can be significantly reduced by developing a repository of approved *deviation permits*, a task which can be scheduled in advance of any code development. The use of *deviation permits* makes it possible to:

- Focus greater control over the generation of any new *deviations*;
- Reduce disruption during the development process;
- Reduce the effort associated with creating and reviewing *deviations*.

A further advantage to be gained by such advance planning is that the *deviation permit* repository can be the subject of negotiated agreement between *supplier* and *acquirer* at an early stage in the project, thereby reducing the likelihood of later disagreement when the code is reviewed by the *acquirer*.

Deviation permits may originate from various sources:

- Public *deviation permits*, published by MISRA;
- Acquirer *deviation permits*, produced by an *acquirer*;
- Supplier *deviation permits*, produced by a *supplier*.

MISRA publishes public *deviation permits* in separate documents for the various versions of the MISRA Guidelines. These address *guidelines* where *violations* can most reasonably be justified and provide examples of best practice which may be followed when additional *deviation permits* are drafted.

Note: Publication of common use cases as *deviation permits* by MISRA does not imply that they are acceptable within a particular project and their use in support of a *deviation* must be subjected to the same balances and measures as for any other *deviation*.

An example *deviation permit* is provided in Appendix C.

4.4 Justifying a deviation

Deviations must not be permitted:

- Simply to satisfy the convenience of the developer;
- When a reasonable alternative coding strategy would make the need for a *violation* unnecessary;
- Without considering the wider consequences of a particular *violation* on other *guidelines*;
- Without the support of a suitable process;
- Without the consent of a designated technical authority.

In addition, a proposed *deviation* should only be approved if it can be justified on the basis of one or more of the following reasons:

Reason 1: Code quality

The Guidelines address code quality with a particular emphasis on issues of safety and security. ISO/IEC 25010 [13] formally defines a more extensive list of characteristics and sub-characteristics of software quality in the form of a “Product Quality Model”, as shown in the following table:

Characteristic	Sub-characteristic
Functional suitability	Functional completeness, Functional correctness, Functional appropriateness
Performance efficiency	Time behaviour, Resource utilization, Capacity
Compatibility	Co-existence, Interoperability
Usability	Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics, Accessibility
Reliability	Maturity, Availability, Fault tolerance, Recoverability
Security	Confidentiality, Integrity, Non-repudiation, Accountability, Authenticity
Maintainability	Modularity, Reusability, Analysability, Modifiability, Testability
Portability	Adaptability, Installability, Replaceability

The ISO/IEC 25010 Product Quality Model

These characteristics are described fully in Section 4.5 of ISO/IEC 25010 [13]. They encompass many different aspects of software quality and only some are associated with coding practices. The key MISRA objectives of safety and security are most closely aligned with the "Functional suitability" and "Reliability" characteristics. However, a *guideline* addressing an issue of safety or security can sometimes have a negative impact on characteristics such as "Maintainability", "Portability" or "Performance efficiency". Similarly, it can be impossible to implement some defensive coding measures (in accordance with the "Fault tolerance" characteristic) when complying with *guidelines* which prohibit invariant expressions and infeasible code.

In such situations it may be reasonable to introduce a *violation* of a *guideline* in order to improve code quality with respect to the characteristic that has been compromised, but only if safety and security are maintained and the result can be justified on the grounds that better overall quality is achieved. As with any *deviation*, the rationale behind the *guideline* must be clearly understood and the risks and merits of a non-compliant approach must be carefully weighed.

Reason 2: Access to hardware

Compiler-specific extensions to the language are often necessary in embedded software systems where low-level access to hardware functionality cannot be provided by the standard language syntax. As with any code which has *implementation-defined behaviour*, it is important for the purposes of maintainability that it should be encapsulated and isolated as much as possible.

Reason 3: Adopted code integration

It is possible for two individual translation units to be compliant with a *guideline* when viewed in isolation, but for *violations* of that *guideline* to emerge as a result of combining the translation units in the same system.

The significance of *adopted code* is described in Section 6. When introducing such code into a system, it may be impossible, impractical, or simply unwise to resolve the problem by rewriting existing code and, in those circumstances, providing suitable precautions are observed, it may be appropriate for a *deviation* to be authorized.

Reason 4: Non-compliant adopted code

When *adopted code* has not been developed with any intention to make it compliant with MISRA Guidelines, it will be meaningless to associate a *deviation* with any of the reasons listed above. The *violation* may have to be justified simply on the basis that the code is adopted and that it can be demonstrated that the *violation* does not compromise safety or security (see Section 6).

Similar considerations may arise when *adopted code* has been written to be compliant with a different version of The Guidelines.

5 The guideline re-categorization plan

Successive versions of The Guidelines have all presented a system of *guideline* categorization. Earlier versions drew a simple distinction between those categorized as Required and those categorized as Advisory. Subsequently, MISRA C:2012 introduced the Mandatory category. These categories define the policy to be followed in determining whether a *guideline* may be *violated* or not and whether a *deviation* is required:

- Mandatory Guidelines — *Guidelines* for which *violation* is never permitted;
- Required Guidelines — *Guidelines* which can only be *violated* when supported by a *deviation* defining a set of clear restrictions, requirements and precautions;
- Advisory Guidelines — Recommendations to be followed as far as is reasonably practical. *Violations* are identified but are not required to be supported by a *deviation*.

5.1 Re-categorization

At the outset of a development project, the *acquirer* and *supplier* shall agree a *guideline re-categorization plan* (*GRP*) to determine how The Guidelines are to be applied. The *GRP* reflects the following issues:

1. It is recognized that in some projects there may be some Advisory *guidelines* which are to be ignored altogether. An additional category, "Disapplied", is therefore introduced to describe this condition. Of course, any decision to disapply a *guideline* should not be taken lightly and the rationale shall therefore be documented in the *GRP*.
2. A principle that has been observed during the evolution of the various versions of The Guidelines is that there are only a small number of Required *guidelines* for which a compelling justification for *deviation* ever arises. This means that, in practice, it is quite possible to treat a high proportion of the Required *guidelines* as Mandatory. Any such re-categorization can only be introduced in the light of experience and some careful investigation; but this discipline can be used very effectively to curb the proliferation of *violations*. A similar approach can be applied to Advisory *guidelines*. If an Advisory *guideline* is considered to be of significant importance, it may be helpful to re-categorize it as a Required *guideline* or even as a Mandatory *guideline*.

A *GRP* can therefore be used to supersede the original system of categorization defined in The Guidelines with a system which differs in the following ways:

- Some Required *guidelines* may be re-categorized as Mandatory;
- Some Advisory *guidelines* may be re-categorized as Mandatory, Required or Disapplied.

MISRA category	*Revised category*			
	Mandatory	Required	Advisory	Disapplied
Mandatory	Permitted			
Required	Permitted	Permitted		
Advisory	Permitted	Permitted	Permitted	Permitted

Notes:

1. A Mandatory *guideline* may not be re-categorized in any way;
2. A Required *guideline* may not be re-categorized as Advisory or Disapplied;

3. *Violations* of guidelines which remain categorized as Advisory do not require a supporting *deviation* but do need to be identified;

4. *Violations* of *guidelines* re-categorized as Disapplied are disregarded altogether.

A partial example of a *GRP* is shown below:

Guideline	*MISRA category*	*Revised category*
Dir 1.1	Required	Mandatory
Dir 2.1	Required	Required
...		
Rule 4.1	Required	Required
Rule 4.2	Advisory	Disapplied
Rule 5.1	Required	Mandatory
...		
Rule 12.1	Advisory	Mandatory
Rule 12.2	Required	Required
Rule 12.3	Advisory	Advisory
Rule 12.4	Advisory	Required

Guideline re-categorization plan example fragment

Note: A project may wish to establish more than one *GRP* for different aspects of a project but in doing so it must be recognized that some *guidelines* have to be applied across the entire system and therefore should be categorized consistently.

6 Adopted Code

6.1 The nature of adopted code

Compliance with MISRA Guidelines is made more complicated when it is necessary to interface with code which is derived from outside the scope of the current project. In many situations, it will not be possible or practicable to modify this code to bring it into compliance, even if it is supplied in source form. This type of code will be referred to as *adopted code*, to distinguish it from code developed within the scope of the current project, which will be referred to as *native code*.

Adopted code is typically derived from sources such as:

- The Standard Library — The library code and header files specified by The Language Standard and provided with the compiler;
- Device driver files — Code either included with the compiler or supplied by a semiconductor manufacturer providing an interface to device peripherals;
- Middleware — Operating systems, protocol stacks, development tools, etc.;
- Third Party Libraries — Mathematical operation libraries, graphical libraries, etc.;
- Automatically generated code — Code generated by modelling tools, UML tools, etc.;
- Legacy code — Code developed for other projects or previous versions of the current project.

The impact of *adopted code* on the integrity of a system must never be overlooked or assumed to be benign. There are two distinct issues to consider:

1. Has the *adopted code* been developed to a verifiably adequate level of safety and security?
2. Has the *adopted code* been developed to be compliant with MISRA Guidelines?

These two issues are not equivalent. Compliance with MISRA Guidelines is neither a necessary nor a sufficient condition to ensure the quality of *adopted code*.

Applying MISRA Guidelines to a project in the presence of *adopted code* introduces a number of difficulties. If *adopted code* has not been developed to exactly the same compliance criteria as the *native code*, claims of MISRA Compliance will inevitably be compromised. In practice, *adopted code* may have been written without MISRA compliance as an objective, but even if it has, it may have been written to comply with a different version of The Guidelines or with different compliance criteria (e.g. a different *GRP*).

6.2 System wide analysis scope

The fact that some *guidelines* have to be applied at system wide analysis scope means that two translation units which are both compliant within themselves (i.e. when viewed in isolation), may not be compliant when combined in the same system. *Guideline violations* can emerge whenever two components are merged into a single system even when both have been developed to comply with the same version of The Guidelines and identical compliance criteria. If the components are both *native code*, then it may be possible to resolve the *violations* with appropriate modification to the code. However, when *adopted code* is involved this may be impractical and it may be necessary to raise a *deviation* in accordance with Reason 3 or Reason 4 (see Section 4.4).

6.3 Adopted binary code

In many projects *adopted code* will only be available in binary form and this will inevitably restrict the scope for verifying not just compliance to MISRA Guidelines but also the inherent quality of the code. Organizations who supply *adopted code* in binary form (*adopted binary code*) will frequently wish to withhold source code in order to protect their intellectual property.

The *supplier* may be willing to issue a statement of MISRA compliance when *adopted binary code* has been developed to comply with MISRA Guidelines. Alternatively, organizations that are collaborating closely may:

- Agree upon a common procedure and tools that each will apply to their own source code;
- Supply each other with stub versions of source code to permit cross-organizational checks to be made.

However, without access to the entire body of source code, the *acquirer* will still be restricted in their ability to verify compliance with those *guidelines* which apply at system level.

When source code is unavailable for analysis, other avenues need to be explored in order to ensure that quality standards are maintained, such as, for example, the use of static or dynamic analysis of the binary code.

Note: some process standards may include wider requirements for managing multi-organization developments, for example Part 8 Clause 5 "Interfaces in distributed development" of ISO 26262 [7].

6.4 Adopted source code

The presence of *guideline violations* in *adopted code* will not necessarily indicate that the code is of poor quality as some *guidelines* can be violated with impunity providing certain conditions are fulfilled. However, *violations* cannot simply be ignored. Every *guideline violation* needs to be reviewed to ensure that integrity has not been compromised and that it is covered by an approved *deviation*.

Problems may arise when:

- Required or Advisory *guidelines* re-categorized as Mandatory are *violated*;
- Advisory *guidelines* re-categorized as Required are *violated* and the use case is not covered by a *deviation*.

Since *adopted code* cannot be modified, it may be necessary to adopt an alternative, less stringent *GRP*. There will then be distinct *GRP*'s, for *native code* and for the *adopted code*. In practice, if there are several distinct units of *adopted code*, it may be appropriate to introduce additional *GRP*'s. The fact that an *adopted code GRP* is more permissive than a *native code GRP* does not imply that the commitment to quality, safety and security is any less stringent for *adopted code*. Having a *guideline* categorized as Mandatory in the *native code GRP* and Required in the *adopted code* simply means that there will be additional *violations* to be reviewed.

Note: Code which does not comply with a *guideline* originally categorized as Mandatory in MISRA Guidelines can never be classified as compliant.

The primary purpose of a *GRP* is to apply constraints to code development by specifying:

1. *Guidelines* which must never be violated;
2. *Guidelines* which must not be *violated* without a formal *deviation*.

When code changes are not an option, as in *adopted code*, the role of the *GRP* is not so much to influence the development of code as to document the extent to which the *adopted code* is compliant and provide assurance that all *guideline violations* have been adequately reviewed. Some *guidelines* which have been re-categorized as Mandatory in *native code* will need to be designated as Required in *adopted code* if *deviations* are necessary.

6.5 Adopted header files

The use of different *GRP*'s for analysis of *native code* and *adopted code* becomes more complicated when *adopted code* header files are included within *native code* modules. A distinction has to be drawn between *violations* which are attributable to the header file and those which are intrinsically associated with the *native code*. Unfortunately, this distinction is complicated by the fact that the *violations* which are attributable to a header file are not always located within the header file itself. Sometimes a *violation* which is clearly attributable to the *adopted code* is actually located within the *native code*. This can occur, for example, when expanding a macro. Consider the following:

```
/* API.h */
#define NOT_NULL( a ) ( ( a ) != 0 )

/* Native.c */
#include API.h

void f ( char * p )
{
  if ( NOT_NULL( p ) )   /* Expansion violates MISRA C:2012 Rule 11.9 */
  {
    use( p );
  }
}
```

The macro `NOT_NULL` defined within the API header file is itself perfectly compliant, but when the macro is invoked with a pointer argument, a *violation* results within the *native code*. In order for the code to be compliant, the macro would need to be written as:

```
#define NOT_NULL( a ) ( ( a ) != NULL )
```

6.6 The Standard Library

Code forming the Standard Library is an integral part of a compiler's implementation and is likely to have been designed with efficiency as a key objective. It may rely on *implementation-defined* or *unspecified behaviours* such as:

1. Casting pointers to object types into pointers to other object types;
2. Pointer arithmetic;
3. Embedding assembly language statements in C.

As it is part of the implementation, and its functionality and interface are defined in The Standard, Standard Library code is not required to comply with MISRA Guidelines. Unless otherwise specified in the individual *guidelines*, the contents of standard header files, and any files that are included during processing of a standard header file, are not required to comply with MISRA Guidelines. However, *guidelines* that rely on the interface provided by standard header declarations and macros are still applicable. For example, the *guidelines* related to type checking of function arguments and return values apply to those functions specified in The Standard Library.

Note: Where a project decides it is of benefit, The Standard Library may be treated in exactly the same way as any other piece of *adopted code*.

7 Claiming MISRA compliance

A project cannot be described as "MISRA Compliant" unless development has been conducted in accordance with the process and principles described in this document. A range of complex issues have been described and these must be appropriately addressed and documented.

7.1 Staff competence

The success of any software development project depends critically on the availability of suitably competent staff. Specific skills are required when The Guidelines are adopted to ensure that the issues underlying the *guidelines* they contain are fully understood and therefore that any proposed *deviations* do not compromise a project's integrity. In many cases it is non-trivial to fully understand the implications of a *violation*, and there is often an interaction between the various *guidelines*. For example, the protection given by Guideline A may rely on behaviour enforced by Guideline B and any *deviation* from Guideline B needs to consider and prevent an unintended consequential *violation* of Guideline A.

Staff who are developing code should receive appropriate training so that they are competent in understanding both The Guidelines and the relevant language issues. The supplier shall maintain staff competence records to allow an *acquirer* to confirm that the staff responsible for the project's MISRA-related activities have the relevant skills and experience. Staff involved in the approval of *deviations*, within both the *supplier* and *acquirer* organizations, need to be especially knowledgeable and experienced.

7.2 The management process

The aim of the compliance process is to eliminate undisciplined development without incurring an unreasonable administrative overhead. The way in which development is managed will be the subject of negotiation between the *acquirer* and the *supplier*. At the outset they will need to reach agreement as to the process by which *deviations* will be managed in order for the notion of compliance to have a precise contractual meaning. The preparation of a realistic and well-designed *GRP* and the formulation of a comprehensive set of *deviation permits* are two ingredients which can make a significant contribution to both the effectiveness and the efficiency of the compliance process.

During the development phase, the processes by which *deviation permits* (where adopted) are compiled and *deviations* are authorized will be of critical importance if development is not to be unduly disrupted. These processes may or may not involve the *acquirer*, but even within the *supplier* organization a clear chain of responsibility must be established with the involvement of suitably competent staff.

When a development contract is negotiated, constraints may be introduced to limit the freedom which the *supplier* has for introducing *deviations*. These constraints are enforced by:

- Re-categorizing Required *guidelines* as Mandatory;
- Re-categorizing Advisory *guidelines* as Mandatory or Required;
- Restricting deviations on specific guidelines to use cases defined in approved *permits*.

These constraints will commonly be imposed by the *acquirer* as a means of exercising control over the development, but they may also be supplemented by additional restrictions introduced internally by the *supplier*.

Consider the two contrasting examples provided below:

Example 1

The *supplier* is given the responsibility to create their own *deviations* and *GRP*, and these are reviewed by the *acquirer* at agreed milestones during the project. This process:

- Delegates greater responsibility to the *supplier*;
- Incurs a greater risk of *deviations* being introduced which the *acquirer* later considers to be unreasonable;
- Defers and magnifies the review process for the *acquirer*;
- May require rework by the *supplier* if *deviations* are deemed unacceptable by the *acquirer*.

Example 2

The contract imposes a restrictive *GRP* in which the majority of *guidelines* are re-categorized as Mandatory and where any *violation* of a Required *guideline* must be covered by a *deviation* which complies with an approved *deviation permit*. These *deviation permits* will be agreed between *acquirer* and *supplier* at the start of the project, except in exceptional circumstances when they may be the subject of negotiated agreement during the development phase. This process:

- Allows the *acquirer* to exercise a high degree of control;
- Limits the scope for unreasonable *deviations*;
- Requires diligent preparation at the outset of the project;
- Potentially incurs greater disruption in development if approval for a new *deviation permit* has to be sought from the *acquirer*.

Both of these examples describe valid processes and both depend on the necessary review activities being administered responsibly and efficiently. The essential difference between the two approaches is the degree of freedom granted by the *acquirer* to the *supplier* in the introduction of *deviations*.

7.3 The guideline compliance summary

As discussed in Section 5.1, a *guideline re-categorization plan* is established at the beginning of a project as a statement of intent detailing how The Guidelines are to be applied to the project's *native code*. Additional *GRP*s may also have been established to accommodate project components consisting of *adopted code*.

At the conclusion of a project, a *guideline compliance summary* (*GCS*) shall be produced to record the final compliance level claimed by a project in its totality. The *GCS* includes an entry for each *guideline* within The Guidelines and records the level of compliance with it, as permitted by its *MISRA category*.

The following levels of compliance may be claimed for a *guideline*:

1. Compliant — There are no *violations* of the *guideline* within the project;
2. Deviations — There are *violations* of the *guideline* within the project which are all supported by *deviations*;
3. Violations — There are *violations* of the *guideline* within the project which are not supported by *deviations*;
4. Disapplied — No checks have been made for compliance with the *guideline*.

The compliance level that may be declared for each *guideline* depends on its *MISRA category*:

MISRA Category	Compliance levels that may be claimed within the *guideline compliance summary*			
Mandatory	Compliant			
Required	Compliant	Deviations		
Advisory	Compliant	Deviations	Violations	Disapplied

These levels allow the extent to which compliance has been achieved across the project to be reflected, capturing the worst-case enforcement permitted by the *GRP*s that have been applied.

Note: Where multiple *GRPs* are used within a project, it may be easier to maintain a *GCS* for each *GRP*. These can then be used to produce a combined *GCS* at the conclusion of the project.

Examples

1. A Required *guideline* is re-categorized as Mandatory within the *native code GRP* and is left as Required within an *adopted code GRP*. The *GCS* would claim a compliance level of Deviations as there were *deviations* in at least one component (the *adopted code*);

2. An Advisory *guideline* is re-categorized as Required within the *native code GRP* and as Disapplied within an *adopted code GRP*. The *GCS* would claim a compliance level of Disapplied as compliance was not checked in at least one component (the *adopted code*).

A partial example of a *GCS* is shown below:

Guideline	*MISRA Category*	Compliance
Dir 1.1	Required	Compliant
Dir 2.1	Required	Deviations
...		
Rule 4.1	Required	Deviations
Rule 4.2	Advisory	Disapplied
Rule 5.1	Required	Compliant
...		
Rule 12.1	Advisory	Compliant
Rule 12.2	Required	Deviations
Rule 12.3	Advisory	Violations
Rule 12.4	Advisory	Deviations

Guideline compliance summary example fragment

7.4 Project delivery

On completion of a project, the *supplier* shall make the following artefacts available to the *acquirer* to support a claim of compliance with The Guidelines:

1. The *guideline enforcement plan* and, if requested by the *acquirer*:

   1.1 The documentation listed in Section 3.3, demonstrating how compliance has been enforced;

   1.2 Documentary evidence proving which tool checks have been performed;

   1.3 Documentary evidence of any *violations* identified.

2. The *guideline compliance summary*, declaring the level of compliance which is being claimed;
3. Details of all approved *deviation permits* (if used);
4. *Deviation records* covering all *violations* of *guidelines* re-categorized as Required;

These documents will be supplied to the *acquirer* in the first instance, but may also be used to provide evidence to any other party who may subsequently use the developed code.

8 References

References to the following MISRA Guidelines should be understood to include all editions (including any amendments).

[1] MISRA C, *Guidelines for the use of the C language in critical systems*, HORIBA MIRA Limited.

[2] MISRA C++, *Guidelines for the use of the C++ language in critical systems*, HORIBA MIRA Limited.

References to the following publications apply to the latest edition of the referenced document (including any amendments).

[3] ISO/IEC 9899, *Programming languages — C*, International Organization for Standardization.

[4] ISO/IEC 14882, *Programming languages — C++*, International Organization for Standardization.

[5] ISO/IEC/IEEE 12207, *Systems and software engineering — Software life cycle processes*, International Organization for Standardization.

[6] IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, International Electrotechnical Commission.

[7] ISO 26262, *Road vehicles — Functional safety*, International Organization for Standardization.

[8] EN 50128, *Railway applications — Communications, signalling and processing systems — Software for railway control and protection*, European Committee for Electrotechnical Standardization.

[9] IEC 62304, *Medical device software — Software life cycle processes*, International Electrotechnical Commission.

[10] DO-178/ED-12, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/EUTOCAE, 2011

[11] ISO 9001 *Quality management systems — Requirements*, International Organization for Standardization.

[12] ISO/IEC 90003 *Software engineering — Guidelines for the application of ISO 9001 to computer software*, International Organization for Standardization.

[13] ISO/IEC/IEEE 25010, *Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, International Organization for Standardization.

References to the following publications apply only to the edition cited.

[14] Straker D., *C Style: Standards and Guidelines*, ISBN 0–13-116898-3, Prentice Hall 1991.

[15] Fenton N.E. and Bieman J., *Software Metrics: A Rigorous and Practical Approach*, 3rd Edition, ISBN 978-1-439838-22-8, CRC Press, 2014.

[16] MISRA Report 5, *Software Metrics*, Motor Industry Research Association, Nuneaton, February 1995.

[17] MISRA Report 6, *Verification and Validation*, Motor Industry Research Association, Nuneaton, February 1995.

Appendix A Process and tools checklist

This Appendix provides a checklist of the development process and tool use guidance that need to be followed in order to claim MISRA compliance, as described in Section 2.2 and Section 7.4.

Section	Guidance
2.3	Staff have been trained in the use of the programming language within embedded system
7.1	Staff have been trained in the use of The Guidelines
2.4	These is a process for enforcing a style guide
2.5	These is a process for enforcing code metrics
2.6.3	There is a process for dealing with deficiencies in the compiler's implementation
2.6.3	There is a process for dealing with deficiencies in the analysis tool's implementation
2.6.4	A choice has been made between possible versions of the programming language
2.6.4	The translator has been configured to accept the correct version of the programming language
2.6.4	The translator has been configured to generate an appropriate level of diagnostic information
2.6.4	The translator has been configured appropriately for the target machine
2.6.4	The translator's optimization level has been configured appropriately
2.6.5	The analysis tools have been configured to accept the correct version of the programming language
2.6.5	The analysis process can deal with any language extensions that have been used
2.6.5	The analysis tools have been configured for the implementation, for example to be aware of the sizes of the integer types
2.6.6	There is a process for ensuring that the program has sufficient resources, such as processing time and stack space
2.6.6	There is a process for demonstrating and recording the absence of run-time errors, for example in module designs
3.3	There is a GEP showing how compliance with each guideline is to be checked
3.4	There is a process for investigating and resolving any diagnostic messages produced by the translator
3.4	There is a process for investigating and resolving any diagnostic messages produced by the analysis tools
3.5	There is a process to manage undecidability issues
4	There is a deviation process for recording and approving deviations
5.1	There is a GRP showing how each guideline is to be enforced
7.3	There is a GCS showing the level of compliance which is being claimed

Appendix B Example deviation record

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

B.1 Summary

The rationale for MISRA C:2012 Rule 10.6 is that it avoids potential developer confusion regarding the type in which some arithmetic operations take place. Unfortunately, because of a "feature" of the compiler being used on this project, it is not possible to make the code comply with the rule without incurring serious run-time performance degradation. The impact of this is that the software's timing requirements cannot be satisfied in all cases and there is a significant risk that the vehicle's legislated hydrocarbon emissions target will not be met.

B.2 Detailed Description

The project makes use of several variable time-step integrators which accumulate a multiple-precision long-term sum. The quantity to be integrated and the integration time-step are both 16-bit unsigned quantities which need to be multiplied to give a 32-bit result which is then accumulated.

Since the C compiler in use on this project implements the *int* type in 32-bits, the code to compute the 32-bit product is:

```
extern uint16_t qty, time_step;

uint32_t  prod = ( uint32_t ) qty * ( uint32_t ) time_step;
```

Clearly, even though the operands of the multiplication operator are 32-bit, there is no possibility that the product is too large for 32-bits because both operands were zero-extended from 16-bits.

In accordance with our standard development procedures, all object modules are passed through a worst-case execution time analysis tool to ensure that each function meets its execution time budget as specified in the architectural design. The analyser highlighted that the code generated for these integrators was far in excess of the budget laid out for them. Investigation revealed that the reason for the excessive execution time was that the compiler was generating a call to a "shift-and-add" style of long multiplication routine. This is surprising because the processor is equipped with an *IMUL* instruction that is capable of multiplying two 16-bit unsigned integers to deliver a 32-bit result in a single clock cycle. Although the multiplication operands are 32-bit, the compiler has the means to know that the most significant 16-bits of these operands are 0 so it should be capable of selecting the *IMUL* instruction.

Experimentation with the compiler suggests that it will select the *IMUL* instruction provided that the operands of the multiplication are implicitly converted from 16-bit to 32-bit, i.e.

```
uint32_t  prod = qty * time_step;
```

This is particularly odd because the behaviour of the code as described by the C Standard is independent of whether the conversion is implicit or explicit. While the program will generate the same results regardless of whether *IMUL* or a library call is used for multiplication, the library call requires a worst case of 100 cycles to execute. The compiler vendor has confirmed in writing that this is the behaviour they expect under the circumstances.

B.3 Justification

Since this type of integrator is used in several functions in the project and is executed at least once every 100 microseconds on average, the performance of the library function is not acceptable. At the specified CPU internal frequency of 25 MHz this means that 4% of the time is spent just on these multiplications. This in itself is insignificant and can be contained in the headroom available in the overall timing budget. However, the design specifies that the integrator shall make its result available within a maximum of 10 microseconds in order to satisfy timing requirements of other functions. The failure to meet this requirement means that there is significant risk in achieving the emissions target, the commercial implications of which are in excess of $10m.

Our preferred solution to this problem is to write the integrators using implicit conversions. This would require deviation against MISRA C:2012 Rule 10.6 for instances of such an integrator. The code is functionally identical to that generated by the MISRA-compliant code but executes up to 100 times faster.

The following other options were considered:

- Increase the clock speed — to achieve the required performance would require a 10-fold increase in clock but the processor's maximum PLL frequency is 100 MHz;
- Change processor — not commercially viable given that hardware design validation is well underway; the additional costs to the project would be around $250,000 and there would be a timing impact too;
- Change compiler — there is no other commercially recognized compiler for this processor; there is an unsupported public domain compiler but it is not considered of suitable quality for this project;
- Recode the library routine — the library uses a base-2 long multiplication; it could be recoded to implement a base-65 536 long multiplication using 3 *IMUL* instructions but we are reluctant to make changes to the compiler vendor's code; we have sought their views on this approach and received the response that "they could not support us making changes to their library".

B.4 Scope

This deviation applies to all instances of variable time-step integrators within the project.

B.5 Risk assessment

There are no consequences associated with non-compliance with MISRA C:2012 Rule 10.6 in the circumstances described in this deviation record.

B.6 Risk management

There are no additional verification and validation requirements resulting from this deviation.

B.7 Actions to control reporting

The MMMC tool used to check compliance with this rule provides a facility whereby a diagnostic message can be suppressed within an expression. Since all integrators are of the form:

```
prod = qty * time_step;
```

a macro can be used to implement the integrator and suppress the warning. The following macro will be used to implement the multiplication and assignment of its result to the product term:

```
/* Violates Rule 10.6: See deviation R_00102 */
#define INTEG(prod, qty, time_step) \
   ( /* -mmmc-R10_6 */ (prod) = (qty) * (time_step) /* -mmmc-pop */ )
```

Although this macro could be implemented as a function, the overhead of the call and return is excessive given the simplicity of the operation being performed. A macro is therefore preferred to a function in this instance even though this means violating Dir 4.9.

Appendix C Example deviation permit

Rule 10.6 The value of a *composite expression* shall not be assigned to an object with wider *essential type*

Permit / Example / C:2012 / R.10.6.A.1

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)

Background

The assignment of a *composite expression* to a wider *essential type* is generally not permitted as it is unclear if the expression is expected to be evaluated in the narrower type of the operands or the wider type of the result.

The "ABC" compiler produces inefficient, slow code when two 16-bit operands are multiplied to produce a 32-bit result when either or both of the operands are cast to a 32-bit type as required to make the expression comply with Rule 10.6.

Performing such multiplications in the absence of the casts yields exactly the same result (due to the effects of integer promotion) with a significant reduction in execution time as a single instruction is used rather than a call to a shift-and-add style long multiplication routine.

Requirements

1. The wider *essential type* shall have a size of 32 bits (i.e. the same as the size of *int*);
2. The *essential type* of the operands shall have a size of 16 bits;
3. The *composite expression* shall have exactly two operands;
4. The *composite expression* shall only contain the arithmetic multiplication operator;
5. It is intended that the *composite expression* be evaluated in the wider type.

Appendix D Glossary

Acquirer

Organization or person that enters into an agreement to acquire or procure a product or service from a *supplier*.

Adopted code

Code that has been developed outside the scope of the current project which may or may not have been developed so as to comply with The Guidelines applied to the project.

Deviation

A *violation* which has been formally accepted and approved.

Deviation permit

The specification of a use case and a set of requirements which may be applied to justify a *deviation*.

Deviation record

The documentation used to justify the presence of a *violation*.

Directive

A *guideline* lacking a complete, unambiguous specification.

Guideline

A *directive* or *rule* that defines a language restriction used to implement part of *The Guidelines*.

Guideline enforcement plan (GEP)

A record of the methods used to provide enforcement of each *guideline*.

Guideline re-categorization plan (GRP)

A policy agreed between the *acquirer* and the *supplier* whereby the *MISRA category* assigned to each *guideline* within The Guidelines is reviewed and in some cases superseded by a more stringent category.

Guideline compliance summary (GCS)

A record of the level of compliance achieved by indicating where *guidelines* have been Disapplied, where *violations* are present and where *deviations* have been introduced.

The Guidelines

A generic term denoting one of the documents within the MISRA Guidelines that is used to enforce a language subset.

MISRA category

A classification (Mandatory, Required or Advisory) applied to every *guideline* within The Guidelines that establishes the conditions under which a *violation* may or may not be permitted.

MISRA Guidelines

A collective name for all editions of MISRA C [1] and MISRA C++ [2].

Native code

Code that has been developed within the scope of the current project which has been developed so as to comply with The Guidelines applied to the project.

Revised category

The classification (Mandatory, Required, Advisory or Disapplied) applied to every *guideline* within the *guideline re-categorization plan* as a result of *guideline* re-categorization.

Rule

A *guideline* having a complete, unambiguous specification.

The Standard

A generic term denoting the ISO language standard referenced by a MISRA coding guideline document.

Supplier

Organization or person that enters into an agreement with the *acquirer* for the supply of a product or service.

Violation

Code which does not conform to the restrictions specified by a *guideline*.