



Coding Standards and MISRA C:2012 Overview



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability

LDRA Standards Experience & Pedigree

Professor Mike Hennell

- Member of SC-205 / WG-71 (DO-178C) formal methods subgroup
- Member of MISRA C committee and MISRA C++ committee
- Member of the working group drafting a proposed secureC annex for the C language definition (SC 22 / WG14)

Bill St Clair

- Member of SC-205 / WG-71 (DO-178C) Object Oriented Technology subgroup

Dr Clive Pygott

- Member of ISO software vulnerabilities working group (SC 22 / WG 23)
- Member of MISRA C++ committee
- Member of the working group drafting a proposed secureC annex for the C language definition (SC 22 / WG14)

Liz Whiting

- Member of MISRA C committee language definition (WG14)

Chris Tapp

- Chairman of MISRA C++ committee
- Member of MISRA C committee language definition (WG14)

Agenda

- Part 1:
 - What is a coding standard?
 - Why use a coding standard?
 - Which coding standard?
- Part 2:
 - MISRA C:2012
 - Checking Compliance
 - Questions & Answers

What is a Coding Standard?



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability

Coding Standards

- Roughly 80% of software defects when using the C or C++ language, are attributable to the incorrect usage of 20% of the language constructs
- If the usage of the language can be restricted to avoid this subset that is known to be problematic, then the quality of the ensuing software is going to greatly increase

If you don't want defects in your code,
then don't put them there!

Keep It Simple

- It is easy to write code that is difficult to read
- Prevent developers from writing “*clever*” code

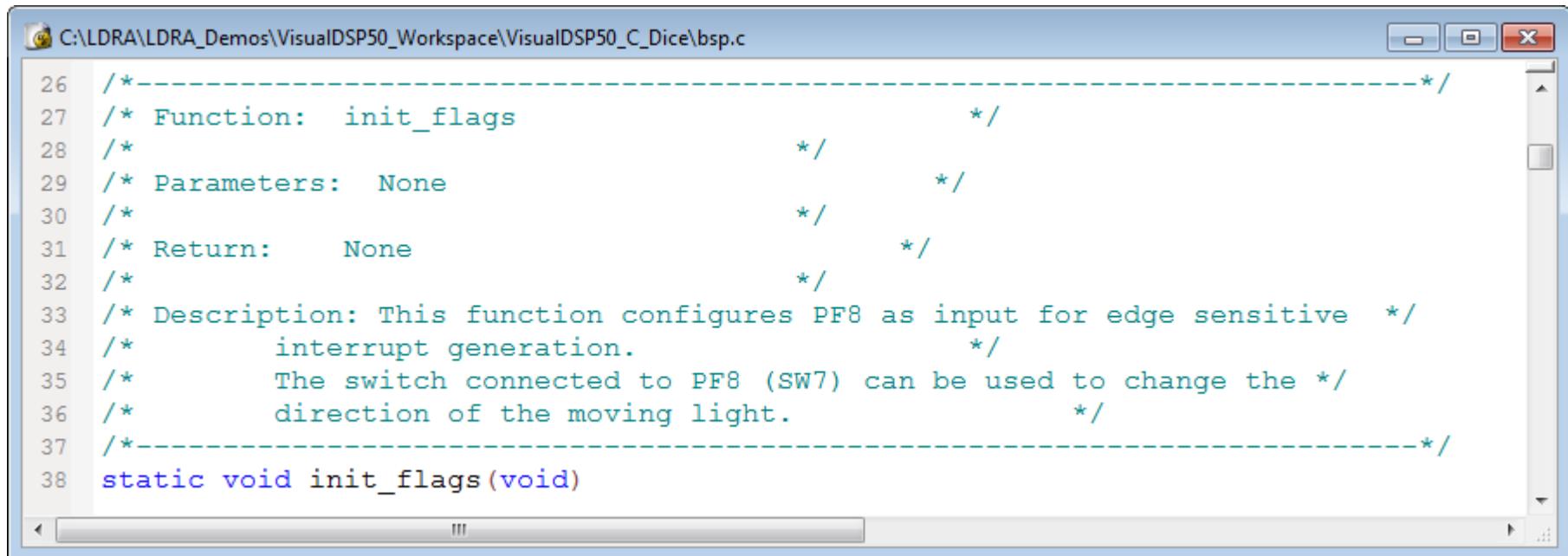
```
void clever_copy (char* to, const char* from, const int count)
{
    int n = (count + 7) / 8;
    switch (count % 8) {
        case 0: do { *to = *from++; }
        case 7:   *to = *from++;
        case 6:   *to = *from++;
        case 5:   *to = *from++;
        case 4:   *to = *from++;
        case 3:   *to = *from++;
        case 2:   *to = *from++;
        case 1:   *to = *from++;
    } while (--n > 0);
}

void copy (char* to, const char* from, const int count)
{
    int n = count;
    do
    {
        *to = *from++;
    } while (--n > 0);
}
```

- Write code that is easy to understand, easy to maintain and easy to test

Style Guidelines

- Tools can be used to verify coding standards, but there is always a need for some manual code review



The screenshot shows a Windows application window with a title bar "C:\LDRA\LDRA_Demos\VisualDSP50_Workspace\VisualDSP50_C_Dice\bsp.c". The main area contains C code for a function named "init_flags". The code uses multi-line comments (/* ... */) to document the function's purpose, parameters, return value, and description. The code is as follows:

```
26  /*-----*/
27  /* Function:  init_flags */
28  /*
29  * Parameters:  None
30  */
31  /* Return:      None
32  */
33  /* Description: This function configures PF8 as input for edge sensitive */
34  /*               interrupt generation. */
35  /*               The switch connected to PF8 (SW7) can be used to change the */
36  /*               direction of the moving light. */
37  /*-----*/
38  static void init_flags(void)
```

- Style guidelines ensure that the code is always written in the same style, facilitating code reviews

Configure Standard

- Coding Standards can be made up of:
 - Common sense rules
 - Don't mix signed and **unsigned** types
 - Reduced language subset
 - Ensure that "goto" or "malloc" is not used
 - Style guidelines
 - Ensure that the "tab" character is not used
 - Naming conventions
 - Ensure that all public functions start with <filename>_
 - Quality & complexity metrics
 - Ensure that all functions have a low cyclomatic complexity

Configure Standard: Example

LDRA Report File Editor - C:\LDRA_Toolsuite\c\creport.dat*

File Edit View Help

Static	Complexity	Data Flow	Cross Ref	Info Flow	Qual Report	Qualsys	LCSAJ	Hungarian Notation	User Def	Section		
Rule Number	Default Strength	Description			MISRA-AC	MISRA	MISRA-C 2004	MISRA C:2012	NETRINO	Standard	ACME Standard	
184	C	LDRA Report File Editor - C:\LDRA_Toolsuite\c\creport.dat*										
185	O	LDRA Report File Editor - C:\LDRA_Toolsuite\c\creport.dat*										
186	O	LDRA Report File Editor - C:\LDRA_Toolsuite\c\creport.dat*										
187	O	LDRA Report File Editor - C:\LDRA_Toolsuite\c\creport.dat*										
188	C	1	M	Cyclomatic complexity greater than ***.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
189	C	2	O	Procedure is not reducible in terms of intervals.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
190												
191												
Report												

File Edit View Help

Static	Complexity	Data Flow	Cross Ref	Info Flow	Qual Report	Qualsys	LCSAJ	Hungarian Notation	User Def	Section 3	Section
Rule Number	Default Strength	Description			MISRA-AC	MISRA	MISRA-C 2004	MISRA C:2012	NETRINO	Standard	ACME Standard
1	M	Global Variable does not conform to style g_<name>.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	M	Enum Element does not conform to style e_<name>.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
6	M	Pointer does not conform to style p_<name>.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7	M	Enum Name does not conform to style E<name>.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				
8	M	Global Func Name does not conform to style <file>_<name>.			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
9	M	Global Var Name does not conform to style <file>_<name>.			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>				

Report File (PC): Section 1 read (25 Models)(27 Modifiers)...Section 2 read (825 Rules)...Section 3 read...Section 4 read...Penalty File (PC): read

Automation

- Use of tools for automation of standards can greatly improve the efficiency of their application
- Over time engineers will quickly tend to alter their habits and write compliant code
- Coding Standards should be adopted from the outset of the project
- If a coding standard is used with existing code that has a proven track record, then the benefits of using the coding standard may be outweighed by the risk of introducing defects in making the code compliant!

Why use a Coding Standard?



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability

Why Use A Coding Standard?

- Simplest way to avoid numerous potential problems
- Mandated by many industrial standards

DO-178C Objectives

- ▷ ✘ Table A-1 - Software Planning Process - Unfulfilled
- ▷ ✘ Table A-2 - Software Development Process - Unfulfilled - 1 asset
- ▷ ✘ Table A-3 - Verification of Outputs of Software Requirements Process - Unfulfilled
- ▷ ✘ Table A-4 - Verification of Outputs of Software Design Process - Unfulfilled
- ▶ ✘ Table A-5 - Verification of Outputs of Software Coding and Process - Unfulfilled
 - ✿ Table A-5 1 - Source Code complies with low-level requirements - Unfulfilled
 - ✿ Table A-5 2 - Source Code complies with software architecture - Unfulfilled
 - ✿ Table A-5 3 - Source Code is verifiable - Unfulfilled
 - ✿ Table A-5 4 - Source Code conforms to standards - Unfulfilled
 - ✿ Table A-5 5 - Source Code is traceable to low-level requirements - Unfulfilled
 - ✿ Table A-5 6 - Source Code is accurate and consistent - Unfulfilled
 - ✿ Table A-5 7 - Output of software integration process is complete and correct - Unfulfilled
 - ✿ Table A-5 8 - Parameter Data Item File is correct and complete - Unfulfilled
 - ✿ Table A-5 9 - Verification of Parameter Data Item File is achieved - Unfulfilled
- ▷ ✘ Table A-5 3 - Source Code is verifiable - Unfulfilled
- ▷ ✘ Table A-5 4 - Source Code conforms to standards - Unfulfilled
- ▷ ✘ Table A-5 5 - Source Code is traceable to low-level requirements - Unfulfilled
- ▷ ✘ Table A-5 6 - Source Code is accurate and consistent - Unfulfilled
- ▷ ✘ Table A-5 7 - Output of software integration process is complete and correct - Unfulfilled
- ▷ ✘ Table A-5 8 - Parameter Data Item File is correct and complete - Unfulfilled
- ▷ ✘ Table A-5 9 - Verification of Parameter Data Item File is achieved - Unfulfilled
- ▷ ✘ Table A-6 - Testing of Outputs of Integration Process - Unfulfilled
- ▷ ✘ Table A-7 - Verification of Verification Process Results - Unfulfilled
- ▷ ✘ Table A-8 - Software Configuration Management Process - Unfulfilled
- ▷ ✘ Table A-9 - Software Quality Assurance Process - Unfulfilled
- ▷ ✘ Table A-10 - Certification Liaison Process - Unfulfilled

IEC 61508 Objectives

- ✖ IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3 : Software requirements
 - ▷ ✖ Table 1 - Software safety lifecycle - overview - Unfulfilled
 - ◀ ✖ Annex A - Guide to the selection of techniques and measures - Unfulfilled
 - ▷ ✖ Table A.1 - Software safety requirements specification - Unfulfilled
 - ▷ ✖ Table A.2 - Software safety analysis - Unfulfilled

- ◀ ✖ Table B.1 - Design and coding standards - Unfulfilled

- ✖ Table B.1 1 - Use of coding standard to reduce likelihood of errors - Unfulfilled
- ✖ Table B.1 2 - No dynamic objects - Unfulfilled
- ✖ Table B.1 3a - No dynamic variables - Unfulfilled
- ✖ Table B.1 3b - Online checking of the installation of dynamic variables - Unfulfilled
- ✖ Table B.1 4 - Limited use of interrupts - Unfulfilled
- ✖ Table B.1 5 - Limited use of pointers - Unfulfilled
- ✖ Table B.1 6 - Limited use of recursion - Unfulfilled
- ✖ Table B.1 7 - No unstructured control flow in programs in higher level languages - Unfulfilled
- ✖ Table B.1 8 - No automatic type conversion - Unfulfilled

- ✖ Table B.1 6 - Limited use of recursion - Unfulfilled
- ✖ Table B.1 7 - No unstructured control flow in programs in higher level languages - Unfulfilled
- ✖ Table B.1 8 - No automatic type conversion - Unfulfilled

- ▷ ✖ Table B.2 - Dynamic analysis and testing - Unfulfilled
- ▷ ✖ Table B.3 - Functional and black-box testing - Unfulfilled
- ▷ ✖ Table B.4 - Failure analysis - Unfulfilled
- ▷ ✖ Table B.5 - Modelling - Unfulfilled
- ▷ ✖ Table B.6 - Performance testing - Unfulfilled
- ▷ ✖ Table B.7 - Semi-formal methods - Unfulfilled
- ▷ ✖ Table B.8 - Static analysis - Unfulfilled
- ▷ ✖ Table B.9 - Modular approach - Unfulfilled

ISO 26262 Objectives

- ISO 26262 - Road Vehicles Safety Standard - Unfulfilled
 - Part 6: - Product Development : Software Level - Unfulfilled
 - Section 5 - Initiation of product development at the software level - Unfulfilled
 - Table 1 - Topics to be covered by modelling and coding guidelines - Unfulfilled
 - 1a - Enforcement of low complexity - Unfulfilled
 - 1b - Use of language subsets - Unfulfilled
 - 1c - Enforcement of strong typing - Unfulfilled
 - 1d - Use of defensive implementation techniques - Unfulfilled
 - 1e - Use of established design principles - Unfulfilled
 - 1f - Use of unambiguous graphical representation - Unfulfilled
 - 1g - Use of style guides - Unfulfilled
 - 1h - Use of naming conventions - Unfulfilled
 - 1i - Appropriate screening properties - Unfulfilled
 - 1g - Restricted use of interrupts - Unfulfilled
 - Table 4 - Mechanisms for error detection at the software architectural level - Unfulfilled
 - Table 5 - Mechanisms for error handling at the software architectural level - Unfulfilled
 - Table 6 - Methods for the verification of the software architectural design - Unfulfilled
 - Section 8 - Software unit design and implementation - Unfulfilled
 - Section 9 - Software unit testing - Unfulfilled
 - Section 10 - Software integration and testing - Unfulfilled
 - Section 11 - Verification of software safety requirements - Unfulfilled

Section 5 - Initiation of product development at the software level - Unfulfilled

- Table 1 - Topics to be covered by modelling and coding guidelines - Unfulfilled
 - 1a - Enforcement of low complexity - Unfulfilled
 - 1b - Use of language subsets - Unfulfilled
 - 1c - Enforcement of strong typing - Unfulfilled
 - 1d - Use of defensive implementation techniques - Unfulfilled
 - 1e - Use of established design principles - Unfulfilled
 - 1f - Use of unambiguous graphical representation - Unfulfilled
 - 1g - Use of style guides - Unfulfilled
 - 1h - Use of naming conventions - Unfulfilled
 - 1i - Appropriate screening properties - Unfulfilled
 - 1g - Restricted use of interrupts - Unfulfilled

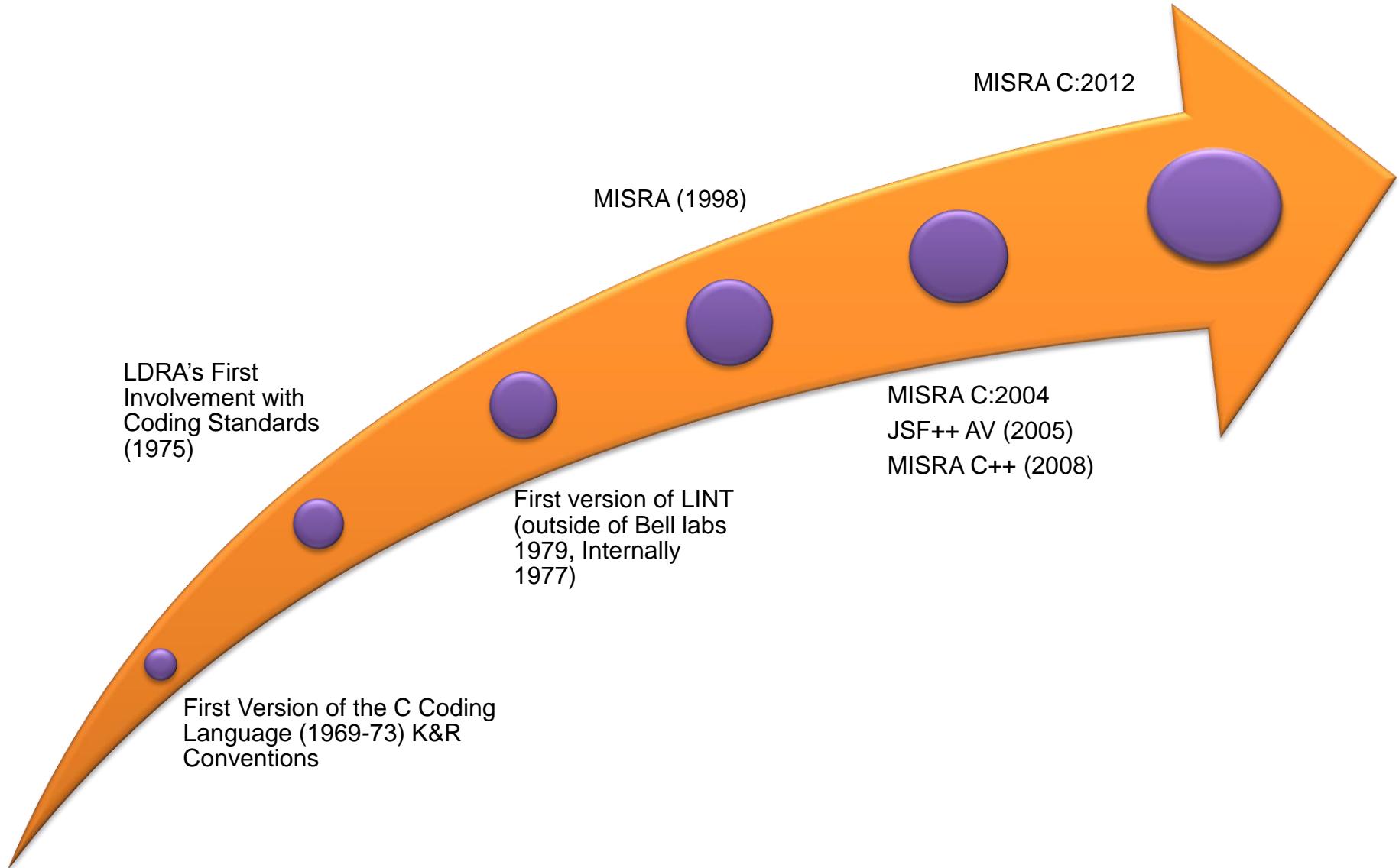
- Table 4 - Mechanisms for error detection at the software architectural level - Unfulfilled
- Table 5 - Mechanisms for error handling at the software architectural level - Unfulfilled
- Table 6 - Methods for the verification of the software architectural design - Unfulfilled
- Section 8 - Software unit design and implementation - Unfulfilled
- Section 9 - Software unit testing - Unfulfilled
- Section 10 - Software integration and testing - Unfulfilled
- Section 11 - Verification of software safety requirements - Unfulfilled

Which Coding Standard?



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability

Progression of Coding Standards

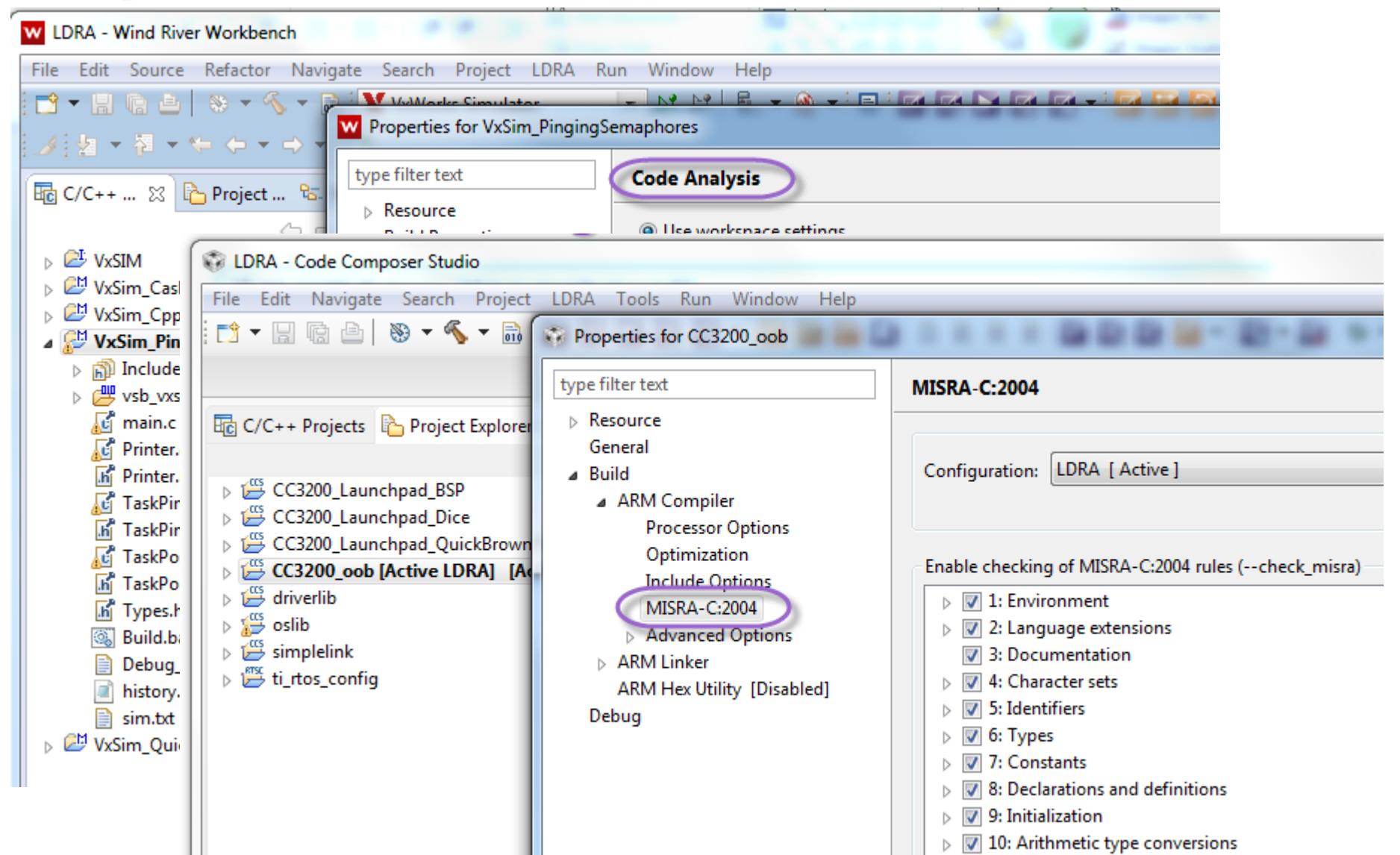


Which “C” Coding Standard?

- MISRA C
 - 1998, 2004, 2012
- NETRINO
- CERT C Secure Coding Standard
- HIS (Hersteller Initiative Software)
- ...
- Company Specific Standard



Eclipse IDEs



Compilers

MPLAB X IDE v2.26 - MPLABX_PIC32MX_Test : default

File Edit View Navigate Source Refactor Run Debug Team Tools Window Help

Main.c

```

1 #include <p32xxxx.h>
2 #include <sys/appio.h>
3
4 static unsigned int ldra_message = 0U;
5 static unsigned int ldra_iter = 0U;
6
7 /* write four characters at a time to the APP IO Window */
8 void ldra_write (void) {
9     /* Wait until ready */
10    while (_DDPSTATbits.APOFUL) {
11        Nop();
12    }
13    /* Write 4 characters at a time */
14    _APPO = ldra_message;

```

ldra_writ... MPLABX_PIC32MX_Test

- ldra_exit()
- ldra_iter
- ldra_message
- ldra_port_write(const char* pMsg)
- ldra_write()
- main()
- p32xxxx.h
- sys/appio.h

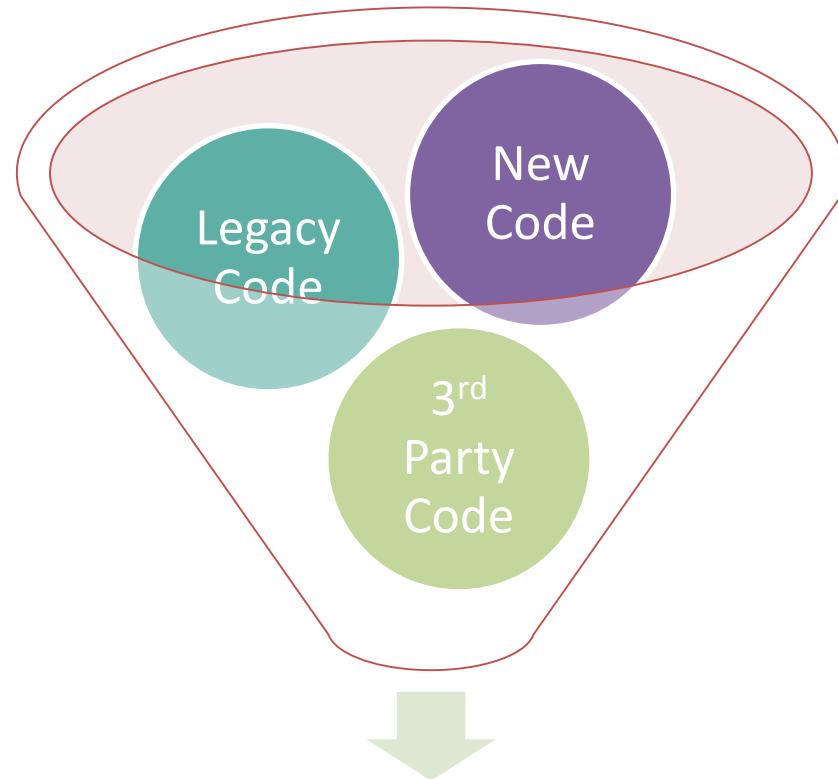
Output PIC AppIO Window LDRA Window

Refresh Violations View Code Review L:/LDRA_Demos/MPLABX_Workspace/MPLABX_PIC32MX_Test.X/Main.c

Source	Line	Description	Standard	LDRA Rule	Severity
Main.c	4	Basic type declaration used.	MISRA-C:1998 13;MISRA-C:2004 8.1;SEC:...	90 S	Advisory
Main.c	5	Basic type declaration used.	MISRA-C:1998 13;MISRA-C:2004 8.1;SEC:...	90 S	Advisory
Main.c	5	Scope of variable could be reduced. : ldra_iter	MISRA-C:1998 22;MISRA-C:2004 8.1;SEC:...	25 D	Advisory
Main.c	11	Function call with no prior declaration. : Nop	MISRA-C:2004 8.1;SEC:...	496 S	Mandatory
Main.c	10	Potentially infinite loop found.	DERA 142;EADS-C 133;...	28 D	Required
Main.c	10	Expression is not Boolean.	MISRA-C:1998 35,36,49;MISRA-C:2004 8.1;SEC:...	114 S	Required
Main.c	14	Signed/unsigned conversion without cast. : (int and unsigned int):...	MISRA-C:1998 43,48;MISRA-C:2004 8.1;SEC:...	434 S	Required
Main.c	8	No prototype for non-static function. : ldra_write	MISRA-C:2012 R.8.4;	106 D	Required

Which “C” Coding Standard?

- New Code
 - MISRA C:2012
 - + Style Guidelines
 - + Naming Conventions
 - + Complexity Guidelines
- 3rd Party Code
 - Custom Standard
- Legacy Code
 - Run-Time Error Standard



Final Code

3rd Party Code

HAL Code Generator tool

(ACTIVE) Rational Rhapsody 8.1

Help

Rational.

IBM.

Search: SXF Go Scope: All topics

Search Results

the Simplified C++ Execution Framework (SXF). Enabling the generation of...

Enabling the generation of MISRA C++ compliant code

About this task. Note: For a framework that is highly compliant with MISRA/C++, it is recommended to use the Simplified C++ Execution Framework (SXF).

Enabling the generation of MISRA C compliant code

Designing applications with the microcontroller profile. What are the major differences between OXF and SXF in Rational Rhapsody IBM Rational Rhapsody Forum...

Getting started: Designing safety-critical applications with Rational ...

Developing Safety-critical applications in C Simplified C++ execution framework (SXF) Enabling the generation of MISRA C compliant code Enabling the...

Project settings

SXFC++ controls the code generation settings for the Rational Rhapsody in C++ SXF (Simplified Execution Framework) for Safety Critical Software development.

Frameworks and operating systems

A simplified C++ execution framework

Rational Rhapsody 8.1.0 > Reference > Frameworks and operating systems

Simplified C++ execution framework (SXF)

A simplified C++ execution framework (SXF) is provided to facilitate the development of applications that must satisfy safety-critical standards.

The following table compares the SXF C++ framework with Rational® Rhapsody®'s standard OXF C++ framework.

Table 1. Comparison of SXF and OXF

SXF	OXF
Static architecture	Dynamic allocation
MISRA C++ 2008 compliant with modeling checks	Not validated for MISRA
No animation/tracing	Animation/Tracing
Only Real Time mode	Real Time/Simulated Time modes
No containers (can be added)	Containers
Static memory manager (only BaseNumberOfInstances)	Static memory manager
Flat statecharts	Flat or reusable statecharts
No Multi-core	Multi-core
No Interfaces	Interface-based
No Ports	Ports
Windriver Workbench 653 Adapter or Microsoft Visual Studio 2008 or 2010 (for host) Support	Multiple operating systems support

Creating a project that uses the SXF framework

To create a project that uses the SXF framework:

1. Select File > New from the main menu.
2. In the New Project window, for Project Settings, select SafetyCriticalForC++Developers.
3. Before generating code, apply the SXF stereotype to the configuration that you are using to generate the code.

MISRA C:2012



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability

What is MISRA C?

It is a language subset



Prevents undefined and unspecified behaviour



Controls implementation-defined behaviour



Gives predictable behaviour

What is Unspecified Behaviour?

Code

```
int      a = 5;  
short   *b = &a;  
  
*b = 6;  
  
printf( a );
```

Result

Compiler is allowed to assume that modification of a *short* will not affect an *int* **even when they are the same size**, so `printf(a)` could print “5” or “6”

Another Example:

Code

```
uint16_t a;  
uint16_t b;  
uint32_t c;  
uint32_t x;  
  
x = a + b + c;
```

Result

Depends on the size of *int* used by the compiler. If it is 16-bit, then there may be loss of Most Significant Bits for $a + b$

Array Bounds Exceeded

Code

```
int32_t a[ 10 ];  
uint32_t i;  
  
for ( i = 0; i <= 10; ++i )  
{  
    a[ i ] = 0;  
}
```

Result

Depending on the runtime environment (OS, etc.), this will result in an exception or overwrite unrelated memory.

Myths and Legends

Does NOT

Say the rules must be followed all the time

Define style or metric guidelines

Stop developers writing code

Does

Require controlled deviations

Require style and metrics be applied

Force developers to think

MISRA C Versions

1998

- MISRA C first released in 1998 to target C90
- Produced by automotive organizations as part of a government-funded project

2004

- Produced by and for diverse industrial sectors
- As compatible as possible with the previous version but more precise / specific
- Introduced the concept of “underlying type”
- Exemplar suite launched in 2007

2012

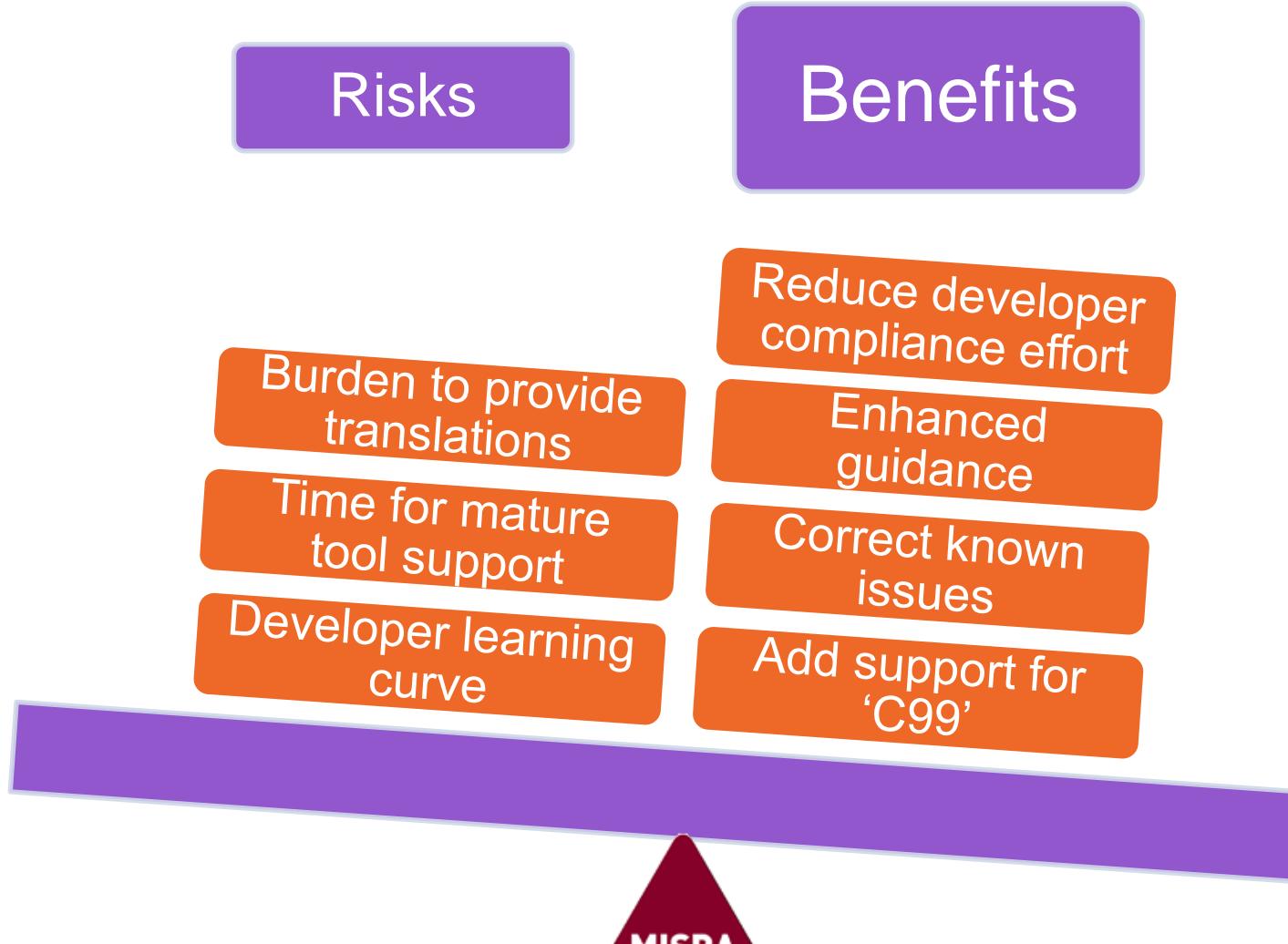
- Adds support for C99, retains support for C90 and corrects issues with 2004 version
- As compatible as possible with the previous version
- All guidelines include a detailed rationale and comprehensive examples
- Includes guidance on the applicability of guidelines to automatically generated code

C99: (ISO/IEC 9899:1999)

- Introduced new features such as:
 - Inline functions
 - New data types ex: _Bool, long long int
 - Variable length arrays
 - C++ style variable declarations
 - C++ style comments //
 - New library functions, ex: snprintf
 - New header files, ex: stdbool.h
 - Improved support for IEEE floating point
 - Macros with a variable number of arguments
- Many compilers have already implemented some if not all of these features



New Version Cost / Benefit Analysis



Objectives

Enhance

- C99 support
- Prefer decidable rules

Improve

- Correct issues with 2004 version
- Ensure rules have strong rationale

Expand

- Type safety of non numeric types
- State rule applicability to auto code

Guideline Considerations

Language

- Applies to C90 and/or C99?

Behaviour

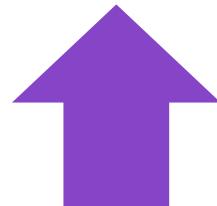
- How should undefined / unspecified behaviour be covered?
- One rule for each / specific guidelines?

Precision

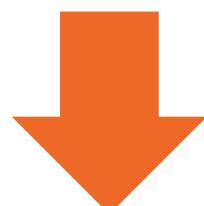
- Make the guideline target the danger as precisely as possible
- Try not to prevent reasonable use of the language

Undefined / Unspecified Behaviour

- Anything at all can happen since the standard imposes no requirements
- The program may:
 - Miraculously do what the programmer intended!
 - Execute incorrectly
 - Fail to compile



Many specific guidelines would be needed to cover all issues



Single rule would make it more difficult for tools to give meaningful diagnostics

Mixed Approach

General

- Undefined / unspecified behaviour for which no practical guidance can be given
- Only reasonable action is to avoid the behaviour

Specific

- Undefined / unspecified behaviours for which advice can be given for a specific compiler
- In this case, it may be possible to justify a deviation for use

Rule Decidability

Decidable

An ideal analysis tool can always determine compliance or non-compliance

Preferred as code review effort lower

Undecidable

Compliance cannot be determined, generally due to control flow variability

False-negatives are possible, so code review is essential

Tool behaviour will vary, leading to more false-positives

Rule 2.2 There shall be no *dead code*

[IEC 61508-7 Section C.5.10], [ISO 26262-6 Section 9.4.5], [DO-178C Section 6.4.4.3.c]

Category Required

Analysis Undecidable, System

Applies to C90, C99

Amplification

Any operation that is executed but whose removal would not affect program behaviour constitutes *dead code*. Operations that are introduced by language extensions are assumed always to have an effect on program behaviour.

Note: The behaviour of an embedded system is often determined not just by the nature of its actions, but also by the time at which they occur.

Note: *unreachable code* is not *dead code* as it cannot be executed.

Rationale

The presence of *dead code* may be indicative of an error in the program's logic. Since *dead code* may be removed by a compiler, its presence may cause confusion.

Exception

A cast to *void* is assumed to indicate a value that is intentionally not being *used*. The cast is therefore not *dead code* itself. It is treated as using its operand which is therefore also not *dead code*.

Enforcement

Headline

- Some headlines are wider / narrower than the behaviour they target to, makes it shorter / easier to understand

Amplification

- Normative text within an Amplification can be used to weaken / strengthen the headline to increase its precision

Exception

- Exceptions are sometimes given to allow restricted use of an otherwise prohibited behaviour

Enforcement

More Precise

- Some MISRA C:2004 guidance was much stronger than needed to protect against a particular issue, making it difficult to achieve what was needed

Tighter Specification

- MISRA C:2004 12.1 “*Limited dependence* should be placed on C’s operator precedence rules in expressions”
- More guidance for tools to be able to make consistent decisions

Guideline Classes

Directives

It is not possible to provide the full description necessary to perform a check for compliance

E.g., Dir 4.4
“Sections of code should not be commented out”

Rules

A complete description of the requirement can be provided

Each guideline is classified as *decidable* or *undecidable*

Dir 4.4 Sections of code should not be “commented out”

Category Advisory

Applies to C90, C99

Amplification

This rule applies to both // and /* ... */ styles of comment.

Rationale

Where it is required for sections of source code not to be compiled then this should be achieved by use of conditional compilation (e.g. #if or #ifdef constructs with a comment). Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.

See also

Rule 3.1, Rule 3.2

Type Checking

Underlying
type

- MISRA C:2004 introduced “Underlying Type” to support stricter type enforcement
- Applicable to numeric types
- Limited (plain) char and Boolean support

Essential
type

- MISRA C:2012 introduces the “Essential Type Model”
- Enumeration and bit-field types included
- Boolean type support improved
- Un-suffixed non-negative integer constant expressions with signed type may now be used in some unsigned contexts
- Better support for (plain) char data type

MISRA C:2012 Highlights

MISRA C:2012

- Includes support for C99
- Gives a reasoned rationale for each rule
- Prefers the use of decidable rules, helping to reduce manual code review requirements
- Allows programmers to spend more time on coding and less on MISRA compliance

Checking Compliance



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability

Checking Compliance

- Check the code manually
 - Needs to be done on the “undecidable” rules
 - But don’t really want to do it on all the code!
- Use a lightweight tool, such as is often built into compilers
 - Fast (*Checks just a subset*)
 - Detects the easy to find defects
 - Tends to be “Optimistic” – **False Negatives**
- Use a heavyweight tool
 - Slow (*Deep analysis*)
 - Detects the easy and **hard** to find defects
 - Tends to be “Pessimistic” – **False Positives**

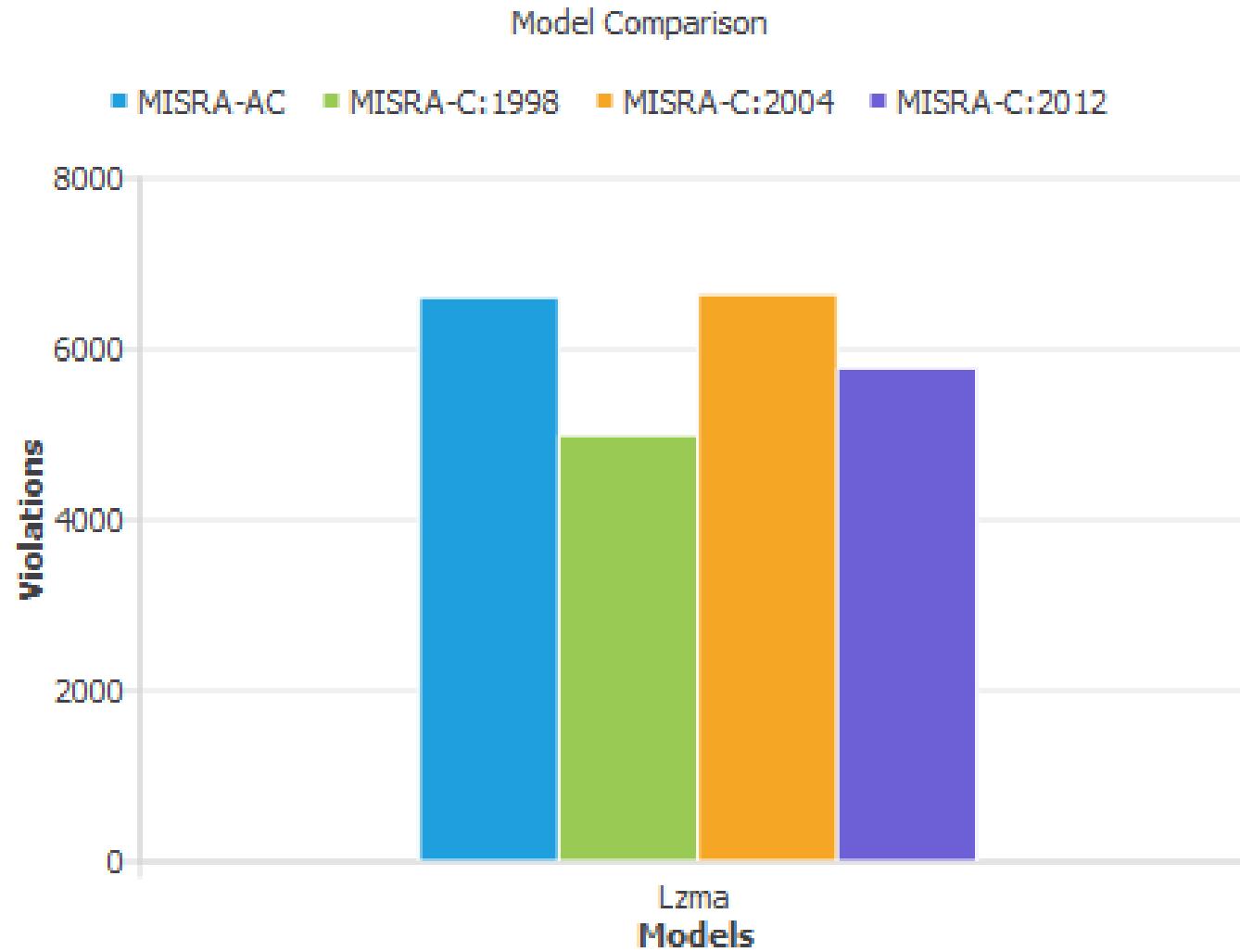


Compliance Document

LDRA TBbrowse - Minow.c cashregister.ojs.html

Number of Violations	LDRA Code	(M) Mandatory Standards	MISRA-C:2012 Code
0	36 S	Function has no return statement.	MISRA-C:2012 R.17.4
0	54 S	Sizeof operator with side effects.	MISRA-C:2012 R.13.6
0	66 S	Function with empty return expression.	MISRA-C:2012 R.17.4
0	407 S	free used on string.	MISRA-C:2012 R.22.2
0	480 S	String function params access same variable.	MISRA-C:2012 R.19.1
0	483 S	free parameter is not heap item.	MISRA-C:2012 R.22.2
0	484 S	Attempt to use already freed object.	MISRA-C:2012 R.22.2
0	496 S	Function call with no prior declaration.	MISRA-C:2012 R.17.3
0	545 S	Assignment of overlapping storage.	MISRA-C:2012 R.19.1
0	591 S	Inappropriate use of file pointer.	MISRA-C:2012 R.22.5
0	614 S	Use of static keyword in array parameter.	MISRA-C:2012 R.17.6
0	631 S	Declaration not reachable.	MISRA-C:2012 R.9.1
0	2 D	Function does not return a value on all paths.	MISRA-C:2012 R.17.4
0	48 D	Attempt to write to unopened file.	MISRA-C:2012 R.22.6
0	51 D	Attempt to read from freed memory.	MISRA-C:2012 R.22.2
0	53 D	Attempt to use uninitialized pointer.	MISRA-C:2012 R.9.1
0	69 D	Procedure contains UR data flow anomalies.	MISRA-C:2012 R.9.1
0	98 D	Attempt to write to file opened read only.	MISRA-C:2012 R.22.4

MISRA Standards Comparison



Code Review

TBvision 9.3.0 © 2013 LDRA Ltd

Source View Configure Website Links Help

File View Results View

Available Results Code Review : (By Violation)

	Number	Level of Violation	Phase Code	Standard Code
◆ #undef used.		Advisory	68 S	MISRA-C:2012 R.20.5
▷ Remainder of % op could be negative.	2	Document	584 S	MISRA-C:2012 D.1.1
▷ Cast on a constant value.	2	Required	203 S	MISRA-C:2012 R.11.8
▷ Value outside range of underlying type.	2	Required	488 S	MISRA-C:2012 R.10.3,R.10.4
▷ Function return value potentially unused.	5	Required	91 D	MISRA-C:2012 D.4.7,R.17.7
▷ Empty middle expression in for loop.	2	Required	429 S	MISRA-C:2012 R.14.2
▷ Call has execution order dependant side effects.	2	Required	1 Q	MISRA-C:2012 R.13.1,R.13.2,...
▷ Name reused in inner scope.	17	Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : limit		Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : i		Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : numPairs		Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : posState		Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : matchPrice		Required	131 S	MISRA-C:2012 R.5.3

Name reused in inner scope.

◆ **Name reused in inner scope. : limit**

◆ Name reused in inner scope. : normalMatchPrice	Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : i	Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : curAndLenPrice	Required	131 S	MISRA-C:2012 R.5.3
◆ Name reused in inner scope. : opt	Required	131 S	MISRA-C:2012 R.5.3

File ... Class ...

MISRA Rule 5.3 : Violation

Rule 5.3 An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

```
static int MY_FAST_CALL LzmaDec_DecodeReal(CLzmaDec *p, SizeT limit, const Byte *bufLimit
{
    CLzmaProb *probs = p->probs;

    unsigned state = p->state;
    UInt32 rep0 = p->reps[0], rep1 = p->reps[1], rep2 = p->reps[2], rep3 = p->reps[3];
    unsigned pbMask = ((unsigned)1 << (p->prop.pb)) - 1;
    unsigned lpMask = ((unsigned)1 << (p->prop.lp)) - 1;
    unsigned lc = p->prop.lc;

    Byte *dic = p->dic;
    SizeT dicBufSize = p->dicBufSize;
    SizeT dicPos = p->dicPos;

    UInt32 processedPos = p->processedPos;
    UInt32 checkDicSize = p->checkDicSize;
    unsigned len = 0;

    const Byte *buf = p->buf;
    UInt32 range = p->range;
    UInt32 code = p->code;

    if (state < kNumberStates ? 0 : 11)
        prob = probs + RepLenCoder;
    }
    {
        unsigned limit, offset;
        CLzmaProb *probLen = prob + LenChoice;
        IF_BIT_0(probLen)
        {
            UPDATE_0(probLen);
            probLen = prob + LenLow + (posState << kLenNumLowBits);
            offset = 0;
            limit = (1 << kLenNumLowBits);
        }
    }
}
```

Code Review

TBvision 9.3.0 © 2013 LDRA Ltd

Source View Configure Website Links Help

File View Results View Available Results Code Review : (By Violation)

	Number	Level of Violation	Phase Code	Standard Code
Violations				
Function has no return statement.	2	Mandatory	36 S	MISRA-C:2012 R.17.4
Procedure contains UR data flow anomalies.	35	Mandatory	69 D	MISRA-C:2012 R.9.1
Function call with no prior declaration.	38	Mandatory	406 S	MISRA-C:2012 R.17.3
Attempt to use uninitialized pointer. : lookBuf		Mandatory	53 D	MISRA-C:2012 R.9.1
More than one break or goto statement in loop.	3	Advisory	409 S	MISRA-C:2012 R.15.4
Use of comma operator.	2	Advisory	53 S	MISRA-C:2012 R.12.3
Numeric overflow.	87	Advisory	493 S	MISRA-C:2012 R.12.4
Typedef name has no size indication.	2	Advisory	495 S	MISRA-C:2012 D.4.6
Comment possibly contains code.	65	Advisory	302 S	MISRA-C:2012 D.4.4
Attempt to change parameter passed by value.	50	Advisory	14 D	MISRA-C:2012 R.17.8
Logical conjunctions need brackets.	155	Advisory	49 S	MISRA-C:2012 R.12.1
Logical conjunctions need brackets.	160	All	076 S	MISRA-C:2012 R.18.4
			012 D.4.9	
			012 D.4.5	
			012 R.15.5	
Basic type declaration used.	190	Advisory	90 S	MISRA-C:2012 D.4.6
Pointer parameter should be declared const.	129	Advisory	62 D	MISRA-C:2012 R.8.13
Reference parameter to procedure is reassigned.	17	Advisory	149 S	MISRA-C:2012 R.17.8
Scope of variable could be reduced.	4	Advisory	25 D	MISRA-C:2012 R.8.9
More than one break or goto statement in loop.	260	All	620 S	MISRA-C:2012 R.15.4

Attempt to use uninitialized pointer. : lookBuf

File Vi... Class Vi...

MISRA Rule 9.1 : False Positive

Rule 9.1 The value of an object with automatic storage duration shall not be read before it has been set

C90 [Undefined 41], C99 [Undefined 10, 17]

Category Mandatory

Analysis Undecidable, System

Applies to C90, C99

The screenshot shows the LDRA TBbrowse interface. The title bar reads "LDRA TBbrowse - [7zStream.c]". The menu bar includes File, Edit, Configure, View, Window, and Help. Below the menu is a toolbar with various icons. The main window displays a C code snippet:

```
SRes LookInStream_LookRead(ILookInStream *stream, void *buf, size_t *size)
{
    const void *lookBuf;
    if (*size == 0)
        return SZ_OK;
    RINOK(stream->Look(stream, &lookBuf, size));
    memcpy(buf, lookBuf, *size);
    return stream->Skip(stream, *size);
}
```

A portion of the code is highlighted with a blue selection bar. A purple oval highlights the expression `&lookBuf` in the line `RINOK(stream->Look(stream, &lookBuf, size));`. The status bar at the bottom left says "For Help, press F1" and the bottom right says "Ln 42, Col 0".

Code Review

	Number	Level of Violation	Phase Code	Standard Code
Violations				
▷ Function has no return statement.	2	Mandatory	36 S	MISRA-C:2012 R.17.4
▷ Procedure contains UR data flow anomalies.	35	Mandatory	69 D	MISRA-C:2012 R.9.1
▷ Function call with no prior declaration.	38	Mandatory	496 S	MISRA-C:2012 R.17.3
◆ Attempt to use uninitialized pointer. : lookBuf		Mandatory	53 D	MISRA-C:2012 R.9.1
▷ More than one break or goto statement in loop.	3	Advisory	409 S	MISRA-C:2012 R.15.4
▷ Use of comma operator.	2	Advisory	53 S	MISRA-C:2012 R.12.3
▷ Numeric overflow.	87	Advisory	493 S	MISRA-C:2012 R.12.4
▷ Typedef name has no size indication.	2	Advisory	495 S	MISRA-C:2012 D.4.6
▷ Comment possibly contains code.	65	Advisory	302 S	MISRA-C:2012 D.4.4
◆ Comment possibly contains code.		Advisory	302 S	MISRA-C:2012 D.4.4
◆ Comment possibly contains code.		Advisory	302 S	MISRA-C:2012 D.4.4
◆ Comment possibly contains code.		Advisory	302 S	MISRA-C:2012 D.4.4
◆ Comment possibly contains code.		Advisory	302 S	MISRA-C:2012 D.4.4



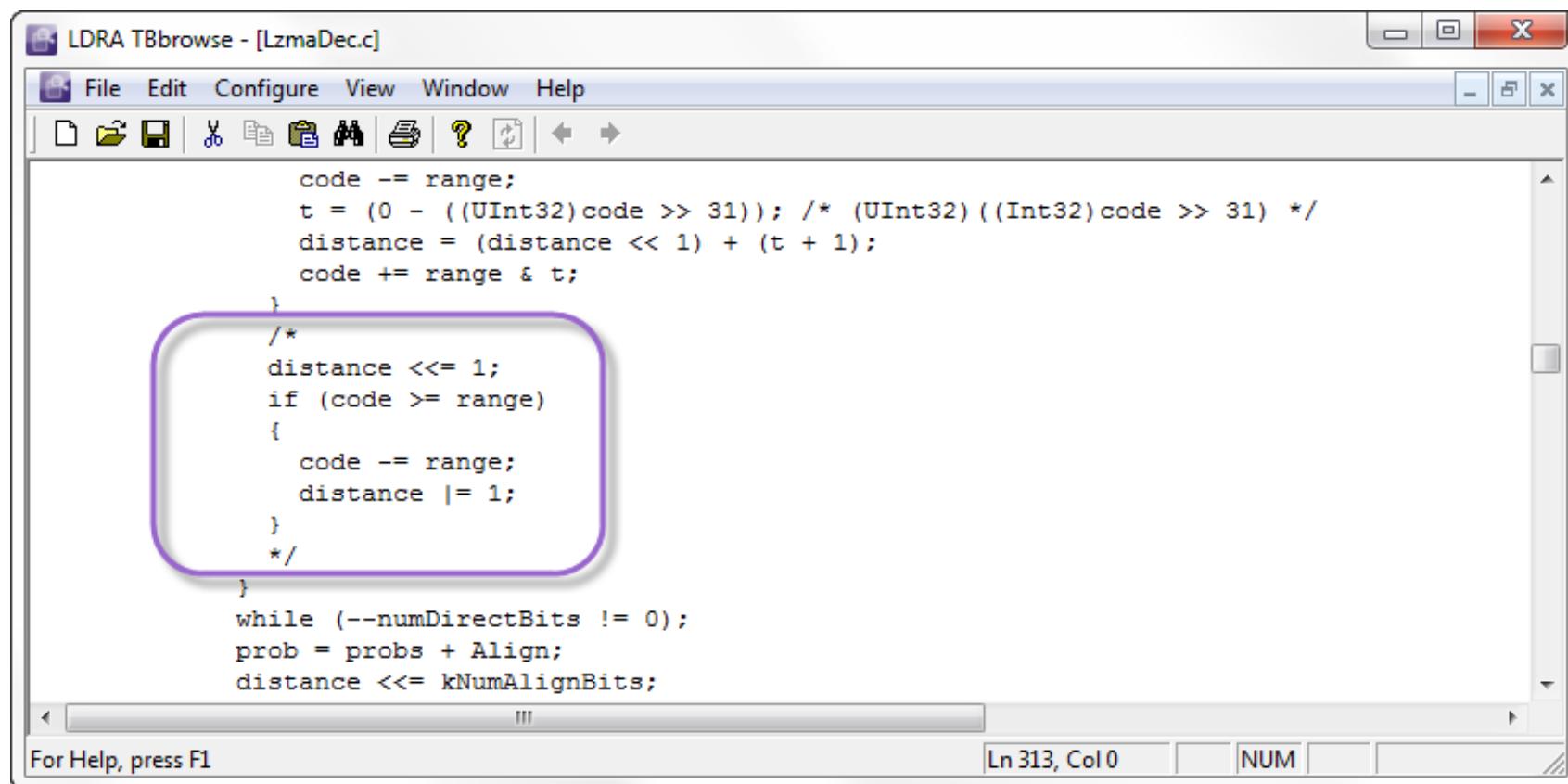
Comment possibly contains code.

MISRA Dir 4.4 : Violation

Dir 4.4 Sections of code should not be “commented out”

Category Advisory

Applies to C90, C99



The screenshot shows a window titled "LDRA TBrowse - [LzmaDec.c]" displaying a C program. A purple rounded rectangle highlights a specific block of code within a conditional statement. The code is as follows:

```
    code -= range;
    t = (0 - ((UInt32)code >> 31)); /* (UInt32)((Int32)code >> 31) */
    distance = (distance << 1) + (t + 1);
    code += range & t;
}
/*
distance <= 1;
if (code >= range)
{
    code -= range;
    distance |= 1;
}
*/
}
while (--numDirectBits != 0);
prob = probs + Align;
distance <= kNumAlignBits;
```

The highlighted code block is the part between the opening brace of the first if-statement and the closing brace of the second if-statement, which contains the assignment `code -= range;` and the condition `if (code >= range)`.

MISRA Dir 4.4 : False Positive

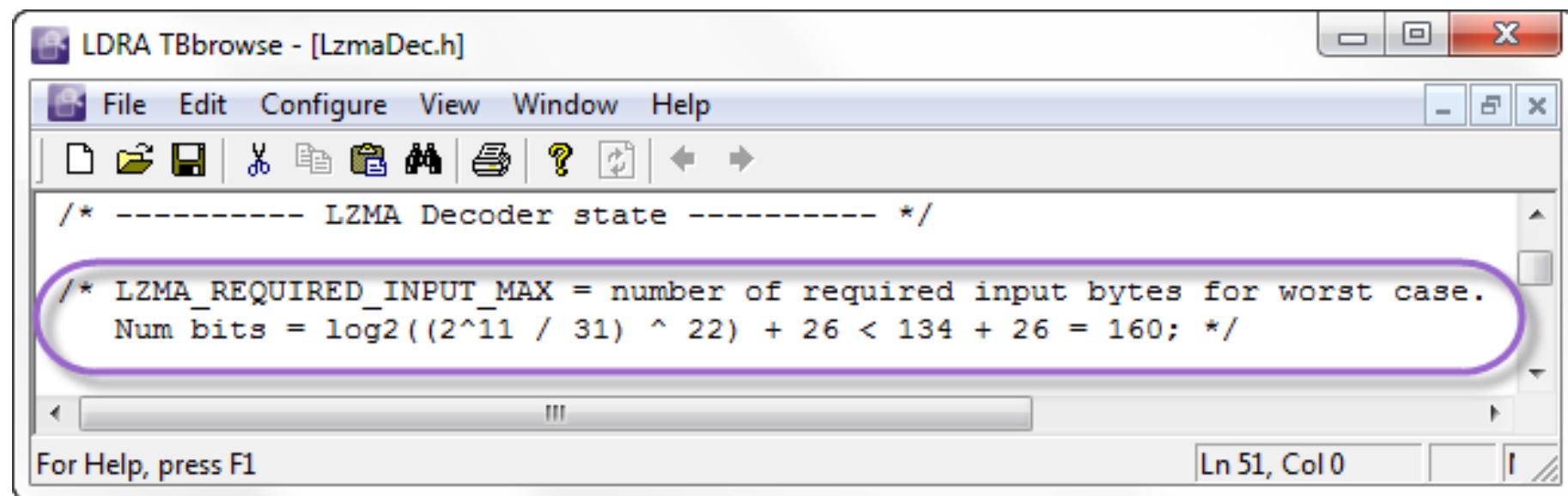
Dir 4.4 Sections of code should not be "commented out"

Category Advisory

Applies to C90, C99

Amplification

This rule applies to both // and /* ... */ styles of comment.



The screenshot shows the LDRA TBrowse interface. The title bar reads "LDRA TBrowse - [LzmaDec.h]". The menu bar includes File, Edit, Configure, View, Window, and Help. The toolbar contains icons for file operations like Open, Save, and Print. The main code editor window displays the following C code:

```
/* ----- LZMA Decoder state ----- */

/* LZMA_REQUIRED_INPUT_MAX = number of required input bytes for worst case.
   Num bits = log2((2^11 / 31) ^ 22) + 26 < 134 + 26 = 160; */
```

A purple oval highlights the second line of the code, which contains a complex mathematical calculation for determining the number of required input bytes. The status bar at the bottom left says "For Help, press F1" and the bottom right says "Ln 51, Col 0".

Deviation

Rule 18.6 The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

C90 [Undefined 9, 26], C99 [Undefined 8, 9, 40]

Category Required

Analysis Undecidable, System

Applies to C90, C99

```
/*
 * Get the product
 * If no product found return NULL_POINTER
 */
const struct Product*
{
    LDRA_uint32_t pIter = 0;
    const struct Product* theProduct = NULL_POINTER;

    while (pIter < MAX_PRODUCTS)
    {
        if (ProductList[pIter].barcode == bCode)
        {
            theProduct = &ProductList[pIter];
            pIter = MAX_PRODUCTS; /* found it so exit loop */
        }
        else
        {
            pIter++;
        }
    }
    /*LDRA_INSPECTED 71 S: ACSD_004: MISRA C:2012 Rule 18.6 : Returned pointer is ok */
    return theProduct;
}
```

Deviation Justification Document

ACME Coding Standard Deviations

Deviation	File	LDRA Rule	MISRA C:2012 Rule	Justification
ACSD_001	CashRegister.c	130 S	R.21.5 Required	Ideally <stdio.h> should not be used, but as this code is just used for demonstration purposes, this include was added so that the code can print something out when it runs.
ACSD_004	Productdatabase.c	71 S	R.18.6 Required	Undecidable rule: Returned pointer has been checked and is ok.
ACSD_005	Productdatabase.c	71 S	R.18.6 Required	Undecidable rule: Returned pointer has been checked and is ok.
ACSD_006	Productdatabase.c	71 S	R.18.6 Required	Undecidable rule: Returned pointer has been checked and is ok.

Summary

- Coding Standard
 - If you don't want errors in your code, don't put them there!
 - Mandated by numerous industrial standards
- MISRA C:2012
 - Generic coding standard
 - Extend by adding style, naming conventions, complexity guidelines, ... specific to the project
 - Support for C90 and C99
 - Rules / Directives
 - Decidable / Undecidable rules
 - Compliance checking



For further information:

www.Idra.com

info@Idra.com



@Idra_technology



LDRA Software Technology



LDRA Limited



Delivering Software Quality and Security through
Test, Analysis & Requirements Traceability