Introduction to MISRA-C Rules

Motor Industry Software Reliability Association

Eng. Ahmed Esmail , Automotive Embedded Software Engineer

Table of Contents

1. What is MISRA?
2. Why are MISRA Rules Important?
3. Overview of MISRA Rules
4. C Problems.
5. Common MISRA Guidelines
9. Examples of MISRA Rules
7. Implementing MISRA in Embedded Systems
8. Benefits and Challenges
9. Case Studies
10. Conclusion

Introduction

What is MISRA?

➢The **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation's Guidelines for the Use of the C Language in Critical Systems describe a subset of C intended for developing safety-critical systems.

➢C is arguably the most popular high-level programming language for embedded systems, but when it comes to developing code for safety-critical systems, the language has many drawbacks. There are several unspecified, implementation-defined, and undefined aspects of the C language that make it unsuited for use when developing safety-critical systems. The MISRA C guidelines are intended to help you to overcame these weaknesses in the C language.

Introduction

Why are MISRA Rules Important?

- Enhanced Safety and Reliability.
- Compliance with Industry Standards.
- Reduced Defects and Vulnerabilities.
- Improved Code Readability and Maintainability.
- Cost Savings.
- Customer Confidence.

➢Overall, MISRA rules play a crucial role in ensuring the safety, reliability, and quality of software in safety-critical systems.

Overview of MISRA Rules

➢ An overview of MISRA rules involves understanding the purpose, scope, and classification of these guidelines.

- Purpose: To promote best practices in software development for embedded systems.
- Scope: Cover various aspects of software development.
  - Coding conventions.
  - Language features.
  - Program structure.
  - Documentation.

They address common sources of errors and vulnerabilities in embedded software and provide recommendations for mitigating risks associated with these issues.

- Classification
  - Mandatory Rules: Must be followed without exception to ensure compliance with MISRA standards
  - Advisory Rules: Provide additional guidance and recommendations for improving code quality

C problems

- The programmer makes mistakes
- The programmer misunderstands the language
- The compiler doesn’t do what the programmer expects
- The compiler contains errors
- Run-time errors

MISRA Guidelines

1. Naming Conventions.
   - Use meaningful and descriptive names.
   - Follow a consistent naming convention.
2. Data Types.
   - Ensure consistent use of signed and unsigned types.
3. Control Flow.
   - Ensure that all code paths have clear entry and exit points.
4. Memory Management.
5. Error Handling.
6. Concurrency and Synchronization.
7. Documentation.

Common MISRA-C Guidelines

MISRA- C Rules Groups

1. Environment rules
2. Language extensions
3. Documentation
4. Character sets
5. Identifiers
6. Types
7. Constants
8. Declarations and definitions
9. Initialization
10. Arithmetic type conversions
11. Pointer type conversions
12. Expressions
13. Control statement expressions
14. Control flow
15. Switch statements
16. Functions
17. Pointers and arrays
18. Structures and unions
19. Preprocessing directives
20. Standard libraries
21. Runtime failures

Common MISRA-C Guidelines

Presentation of Rules

Rule<number>(<category>): <requirement text>

[<source ref>]

<number> Every rule has a unique number. This number consists of a rule group prefix and a group member suffix.

<category> is one of “required” or “advisory”.

<requirement text> The rule itself.

<source ref> This indicates the primary source(s) which led to this item or group of items, where applicable.

➢ There have been three releases of the MISRA C standard.

- MISRA C:1998 (It was written for C90. There are 127 coding rules)
- MISRA C:2004 (It was written for C90. There are 142 coding rules)
- MISRA C:2012 (It was written for C99. There are 143 coding rules)

Examples of MISRA-C Rules

Examples of MISRA rules provide specific guidelines for writing code that adheres to the MISRA standards.

Here are some examples of MISRA rules:

1. Local variable and function parameter names should comply with a naming convention.

Noncompliant Code Example

```
void doSomething(int my_param) {
  int LOCAL;
  ...
}
```

Compliant Solution

```
void doSomething(int myParam) {
  int local;
  ...
}
```

With the default regular expression ^[a-z][a-zA-Z0-9]*$

Examples of MISRA-C Rules count.

2. Control structures should use curly braces.

Noncompliant Code Example

```
if (condition)  // Noncompliant
  executeSomething();
```

Compliant Solution

```
if (condition) {
  executeSomething();
}
```

Examples of MISRA-C Rules count.

3. Switch cases should end with an unconditional "break" statement.

Noncompliant Code Example

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:  // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ?
    doSomething();
  default:
    doSomethingElse();
    break;
}
```

Examples of MISRA-C Rules count.

3. Switch cases should end with an unconditional "break" statement.

Compliant Solution

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:
    doSomething();
    break;
  default:
    doSomethingElse();
    break;
}
```

Examples of MISRA-C Rules count.

4. Empty statements should be removed.

Noncompliant Code Example

```
void doSomething() {
  ;
}
```

Compliant Solution

```
void doSomething() {
}
```

Examples of MISRA-C Rules count.

5. Magic numbers should not be used.

Noncompliant Code Example

```
void doSomething() {
  for(int i = 0; i < 42; i++) {                 // Noncompliant - 42 is a magic number
    // ...
  }

  if (var == 42) {                              // Noncompliant - magic number
    // ...
  }
}
```

Examples of MISRA-C Rules count.

5. Magic numbers should not be used.

Compliant Solution

```
#define STATUS_OK 42

void doSomething() {
  int maxIterations = 42;                    // Compliant - in a declaration
  for(int i = 0; i < maxIterations ; i++){   // Compliant
    // ...
  }

  if (var == 0) {                            // Compliant - 0 is excluded
    // ...
  }

  if (var == STATUS_OK) {                    // Compliant - number comes from a macro
    // ...
  }
}
```

Next you will find the roles for each group of the MISRA-C

Group 1: Environment

No	Rule	Type	Category
1.1	All code shall conform to ISO 9899:1990 Programming languages – C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	Environment	Required
1.2	No reliance shall be placed on undefined or unspecified behavior.	Environment	Required
1.3	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.	Environment	Required
1.4	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	Environment	Required
1.5	Floating-point implementations should comply with a defined floating-point standard.	Environment	Advisory

Group 2: Language extensions

No	Rule	Type	Category
2.1	Assembler language shall be encapsulated and isolated.	Language extensions	Required
2.2	Source code shall only use /* ... */ style comments.	Language extensions	Required
2.3	The character sequence /* shall not be used within a comment.	Language extensions	Required
2.4	Sections of code should not be commented out.	Language extensions	Advisory

Group 3: Documentation

No	Rule	Type	Category
3.1	All usage of implementation-defined behavior shall be documented.	Documentation	Required
3.2	The character set and the corresponding encoding shall be documented.	Documentation	Required
3.3	The implementation of integer division in the chosen compiler should be determined, documented, and taken into account.	Documentation	Advisory
3.4	All uses of the #pragma directive shall be documented and explained.	Documentation	Required
3.5	If it is being relied upon, the implementation-defined behavior and packing of bitfields shall be documented.	Documentation	Required
3.6	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	Documentation	Required

Group 4: Character sets

No	Rule	Type	Category
4.1	Only those escape sequences that are defined in the ISO C standard shall be used.	Character sets	Required
4.2	Trigraphs shall not be used.	Character sets	Required

Group 5: Identifiers

No	Rule	Type	Category
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	Identifiers	Required
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Identifiers	Required
5.3	A typedef name shall be a unique identifier.	Identifiers	Required
5.4	A tag name shall be a unique identifier.	Identifiers	Required
5.5	No object or function identifier with static storage duration should be reused.	Identifiers	Advisory
5.6	No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names.	Identifiers	Advisory
5.7	No identifier name should be reused.	Identifiers	Advisory

Group 6: Types

No	Rule	Type	Category
6.1	The plain char type shall be used only for the storage and use of character values.	Types	Required
6.2	signed and unsigned char type shall be used only for the storage and use of numeric values.	Types	Required
6.3	`typedef`s that indicate size and signedness should be used in place of the basic types.	Types	Advisory
6.4	Bitfields shall only be defined to be of type unsigned int or signed int.	Types	Required
6.5	Bitfields of signed type shall be at least 2 bits long.	Types	Required

Group 7: Constants

No	Rule	Type	Category
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	Constants	Required

Group 8: Declarations and definitions

No	Rule	Type	Category
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	Declarations and definitions	Required
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.	Declarations and definitions	Required
8.3	For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical.	Declarations and definitions	Required
8.4	If objects or functions are declared more than once, their types shall be compatible.	Declarations and definitions	Required
8.5	There shall be no definitions of objects or functions in a header file.	Declarations and definitions	Required
8.6	Functions shall be declared at file scope.	Declarations and definitions	Required

Group 8: cont..

8.7	Objects shall be defined at block scope if they are only accessed from within a single function.	Declarations and definitions	Required
8.8	An external object or function shall be declared in one and only one file.	Declarations and definitions	Required
8.9	An identifier with external linkage shall have exactly one external definition.	Declarations and definitions	Required
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	Declarations and definitions	Required
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	Declarations and definitions	Required
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.	Declarations and definitions	Required

Group 9: Initialization

No	Rule	Type	Category
9.1	All automatic variables shall have been assigned a value before being used.	Initialization	Required
9.2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	Initialization	Required
9.3	In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Initialization	Required

Group 10: Arithmetic type conversions

No	Rule	Type	Category
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.	Arithmetic type conversions	Required
10.2	The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.	Arithmetic type conversions	Required

Group 10: cont..

10.3	The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression.	Arithmetic type conversions	Required
10.4	The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.	Arithmetic type conversions	Required
10.5	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	Arithmetic type conversions	Required
10.6	A U suffix shall be applied to all constants of unsigned type.	Arithmetic type conversions	Required

Group 11: Pointer type conversions

No	Rule	Type	Category
11.1	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	Pointer type conversions	Required
11.2	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type, or a pointer to void.	Pointer type conversions	Required
11.3	A cast should not be performed between a pointer type and an integral type.	Pointer type conversions	Advisory
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	Pointer type conversions	Advisory
11.5	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	Pointer type conversions	Required

Group 12: Expressions

No	Rule	Type	Category
12.1	Limited dependence should be placed on the C operator precedence rules in expressions.	Expressions	Advisory
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	Expressions	Required
12.3	The sizeof operator shall not be used on expressions that contain side effects.	Expressions	Required
12.4	The right-hand operand of a logical && or || operator shall not contain side effects.	Expressions	Required
12.5	The operands of a logical && or || shall be primary expressions.	Expressions	Required
12.6	The operands of logical operators (&&, ||, and !) should be effectively boolean. Expressions that are effectively boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).	Expressions	Advisory
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.	Expressions	Required

Group 12: cont..

12.8	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	Expressions	Required
12.9	Trigraphs shall not be used.The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	Expressions	Required
12.10	The comma operator shall not be used.	Expressions	Required
12.11	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	Expressions	Advisory
12.12	The underlying bit representations of floating-point values shall not be used.	Expressions	Required
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	Expressions	Advisory

Group 13: Control statement expressions

No	Rule	Type	Category
13.1	Assignment operators shall not be used in expressions that yield a boolean value.	Control statement expressions	Required
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively boolean.	Control statement expressions	Advisory
13.3	Floating-point expressions shall not be tested for equality or inequality.	Control statement expressions	Required
13.4	The controlling expression of a for statement shall not contain any objects of floating type.	Control statement expressions	Required
13.5	The three expressions of a for statement shall be concerned only with loop control.	Control statement expressions	Required
13.6	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	Control statement expressions	Required
13.7	Boolean operations whose results are invariant shall not be permitted.	Control statement expressions	Required

Group 14: Control flow

No	Rule	Type	Category
14.1	There shall be no unreachable code.	Control flow	Required
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change.	Control flow	Required
14.3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character.	Control flow	Required
14.4	The goto statement shall not be used.	Control flow	Required
14.5	The continue statement shall not be used.	Control flow	Required
14.6	For any iteration statement, there shall be at most one break statement used for loop termination.	Control flow	Required

Group 14: cont..

14.7	A function shall have a single point of exit at the end of the function.	Control flow	Required
14.8	The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.	Control flow	Required
14.9	An if expression construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement or another if statement.	Control flow	Required
14.10	All if ... else if constructs shall be terminated with an else clause.	Control flow	Required

Group 15: Switch statements

No	Rule	Type	Category
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Switch statements	Required
15.2	An unconditional break statement shall terminate every non-empty switch clause.	Switch statements	Required
15.3	The final clause of a switch statement shall be the default clause.	Switch statements	Required
15.4	A switch expression shall not represent a value that is effectively boolean.	Switch statements	Required
15.5	Every switch statement shall have at least one case clause.	Switch statements	Required

Group 16: Functions

No	Rule	Type	Category
16.1	Functions shall not be defined with a variable number of arguments.	Functions	Required
16.2	Functions shall not call themselves, either directly or indirectly.	Functions	Required
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Functions	Required
16.4	The identifiers used in the declaration and definition of a function shall be identical.	Functions	Required
16.5	Functions with no parameters shall be declared and defined with the parameter list void.	Functions	Required
16.6	The number of arguments passed to a function shall match the number of parameters.	Functions	Required

Group 16: cont..

16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	Functions	Advisory
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Functions	Required
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Functions	Required
16.10	If a function returns error information, then that error information shall be tested.	Functions	Required

Group 17: Pointers and arrays

No	Rule	Type	Category
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointers and arrays	Required
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Pointers and arrays	Required
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Pointers and arrays	Required
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Pointers and arrays	Required
17.5	The declaration of objects should contain no more than two levels of pointer indirection.	Pointers and arrays	Advisory
17.6	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	Pointers and arrays	Required

Group 18: Structures and unions

No	Rule	Type	Category
18.1	All structure and union types shall be complete at the end of the translation unit.	Structures and unions	Required
18.2	An object shall not be assigned to an overlapping object.	Structures and unions	Required
18.3	An area of memory shall not be used for unrelated purposes.	Structures and unions	Required
18.4	Unions shall not be used.	Structures and unions	Required

Group 19: Preprocessing directives

No	Rule	Type	Category
19.1	`#include` statements in a file should only be preceded by other preprocessor directives or comments.	Preprocessing directives	Advisory
19.2	Non-standard characters should not occur in header file names in #include directives.	Preprocessing directives	Advisory
19.3	The #include directive shall be followed by either a <filename> or "filename" sequence.	Preprocessing directives	Required
19.4	C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Preprocessing directives	Required
19.5	Macros shall not be #define'd or #undef'd within a block.	Preprocessing directives	Required
19.6	#undef shall not be used.	Preprocessing	Required

Group 19: Preprocessing directives cont..

19.7	A function should be used in preference to a function-like macro.	Preprocessing directives	Advisory
19.8	A function-like macro shall not be invoked without all of its arguments.	Preprocessing directives	Required
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Preprocessing directives	Required
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Preprocessing directives	Required
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	Preprocessing directives	Required
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	Preprocessing directives	Required

Group 19: Preprocessing directives cont..

19.13	The # and ## preprocessor operators should not be used.	Preprocessing directives	Advisory
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	Preprocessing directives	Required
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Preprocessing directives	Required
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	Preprocessing directives	Required
19.17	All #else, #elif, and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	Preprocessing directives	Required

Group 20: Standard libraries

No	Rule	Type	Category
20.1	Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined.	Standard libraries	Required
20.2	The names of Standard Library macros, objects, and functions shall not be reused.	Standard libraries	Required
20.3	The validity of values passed to library functions shall be checked.	Standard libraries	Required
20.4	Dynamic heap memory allocation shall not be used.	Standard libraries	Required
20.5	The error indicator errno shall not be used.	Standard libraries	Required
20.6	The macro offsetof in the stddef.h library shall not be used.	Standard libraries	Required
20.7	The setjmp macro and the longjmp function shall not be used.	Standard libraries	Required

Group 20: Standard libraries cont..

20.8	The signal handling facilities of signal.h shall not be used.	Standard libraries	Required
20.9	The input/output library stdio.h shall not be used in production code.	Standard libraries	Required
20.10	The functions atof, atoi, and atol from the library stdlib.h shall not be used.	Standard libraries	Required
20.11	The functions abort, exit, getenv, and system from the library stdlib.h shall not be used.	Standard libraries	Required
20.12	The time handling functions of time.h shall not be used.	Standard libraries	Required

Group 21: Runtime failures

No	Rule	Type	Category
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: a. static analysis tools/techniques b. dynamic analysis tools/techniques c. explicit coding of checks to handle runtime faults.	Run time failures	Required

Implementing MISRA in Embedded Systems

➢ Implementing MISRA in embedded systems involves integrating MISRA guidelines into the software development process to ensure compliance with the standards.

1. Selecting the Appropriate MISRA Standard.
2. Understanding the MISRA Guidelines.
3. Setting up Development Tools.
4. Establishing Coding Standards.
5. Code Reviews and Inspections.
6. Static Code Analysis.
7. Testing and Verification.
8. Documentation and Traceability.
9. Training and Education.
10. Continuous Improvement.

➢ By following these steps, you can effectively implement MISRA guidelines in embedded systems development, leading to safer, more reliable, and maintainable software products.

Benefits and Challenges of MISRA-C

➢ Implementing MISRA guidelines in embedded systems development offers several benefits, but it also presents certain challenges. Let's explore both

Benefits

1. Enhanced Safety and Reliability.
2. Compliance with Industry Standards.
3. Reduced Defects and Vulnerabilities.
4. Improved Code Readability and Maintainability.
5. Customer Confidence.

Challenges

1. Learning Curve.
2. Tooling and Resources.
3. Impact on Productivity.
4. Flexibility vs. Rigidity.
5. Legacy Code and Integration.

➢ Despite these challenges, the benefits of implementing MISRA guidelines in embedded systems development often outweigh the drawbacks, leading to safer, more reliable, and higher-quality software products. With proper planning, training, and commitment, organizations can successfully navigate the challenges of adopting MISRA standards and reap the benefits of improved software quality and compliance.

Case Studies of MISRA-C

1. Automotive Industry - Ford Motor Company.
   - Ford Motor Company adopted MISRA C guidelines for its embedded software development in automotive systems.
   - Implementation of MISRA guidelines helped Ford minimize software defects and vulnerabilities, resulting in safer and more dependable vehicles on the road.
2. Aerospace Industry - Boeing.
3. Medical Devices - Medtronic.
4. Rail Transportation - Siemens Mobility.

➢ These case studies highlight the diverse applications of MISRA guidelines across different industries and demonstrate the significant impact of adhering to MISRA standards on the safety, reliability, and quality of embedded systems. By following MISRA guidelines, organizations can enhance the safety, reliability, and quality of their products, gain regulatory approvals, and build trust with customers and stakeholders.

Conclusion

➢In conclusion, the adoption of MISRA guidelines is essential for organizations developing embedded systems for safety-critical applications. By embracing these guidelines, organizations can build software that not only meets regulatory requirements but also instills confidence in its reliability, safety, and quality, ultimately contributing to the overall success and trustworthiness of their products in the marketplace.

THANK YOU

Eng. Ahmed Esmail,
Automotive Embedded Software Engineer