Embedded C refers to the use of the C programming language in the development of embedded systems. These systems typically involve hardware with limited resources, such as microcontrollers. The rules for Embedded C programming focus on writing efficient, reliable, and maintainable code suitable for the constrained environments of embedded systems. Here are some general rules and best practices for Embedded C:

General Rules and Best Practices

Avoid Dynamic Memory Allocation

Rule: Refrain from using dynamic memory allocation (malloc, free).

Reason: Embedded systems often have limited memory, and dynamic memory allocation can lead to fragmentation and unpredictable behavior.

Use Fixed-Size Data Types

Rule: Prefer fixed-size data types (e.g., uint8_t, int16_t) over generic types (int, long).

Reason: Ensures consistency across different platforms and compilers.

Minimize Interrupt Handling Code

Rule: Keep interrupt service routines (ISRs) short and fast.

Reason: Long ISRs can delay other critical tasks and lead to missed interrupts.

Avoid Using Standard Library Functions

Rule: Be cautious with standard library functions like printf, scanf, etc.

Reason: These functions can be resource-intensive and may not be suitable for embedded environments.

Limit Use of Global Variables

Rule: Minimize the use of global variables.

Reason: Global variables can lead to code that is difficult to maintain and debug.

Use Volatile Keyword for Shared Variables

Rule: Use volatile for variables shared between ISRs and main code.

Reason: Ensures the compiler does not optimize away necessary reads and writes.

Check for Hardware-Specific Constraints

Rule: Be mindful of hardware-specific constraints and features.

Reason: Embedded systems interact directly with hardware, requiring careful handling of peripheral registers and memory-mapped I/O.

Coding Standards and Guidelines

1. MISRA C Compliance

Rule: Follow MISRA C guidelines.

Reason: MISRA C provides a comprehensive set of rules for writing safe, portable, and reliable code in embedded systems.

2. Code Readability and Maintainability

Rule: Write clear and maintainable code with proper comments and documentation.

Reason: Improves code readability and ease of maintenance.

3. Function Design

Rule: Keep functions small and focused on a single task.

Reason: Enhances readability, testing, and reusability.

4. Error Handling

Rule: Implement robust error handling.

Reason: Ensures the system can handle unexpected situations gracefully.

5. Testing and Validation

Rule: Thoroughly test and validate the code, especially in real hardware environments.

Reason: Ensures the system behaves correctly under all conditions.

Performance and Optimization

Optimize for Speed and Memory

Rule: Write code with performance and memory constraints in mind.

Reason: Embedded systems often have limited resources.

Inline Functions

Rule: Use inline functions for small, frequently called functions.

Reason: Reduces function call overhead.

Loop Unrolling

Rule: Consider loop unrolling for performance-critical loops.

Reason: Can reduce the overhead of loop control code.

Portability and Scalability

Avoid Hardware-Specific Code

Rule: Abstract hardware-specific code to separate modules.

Reason: Enhances portability and ease of modification.

Use Preprocessor Directives Wisely

Rule: Use preprocessor directives for conditional compilation.

Reason: Helps in managing different hardware configurations and debugging.

Scalable Architecture

Rule: Design the software architecture to be scalable and modular.

Reason: Facilitates easier updates and feature additions.

Safety and Reliability

Watchdog Timers

Rule: Implement watchdog timers to reset the system in case of failure.

Reason: Ensures the system can recover from unexpected states.

Redundancy

Rule: Implement redundancy for critical tasks.

Reason: Increases system reliability.

Fail-Safe Mechanisms

Rule: Design fail-safe mechanisms to handle critical failures.

Reason: Ensures the system can fail gracefully.

By following these rules and guidelines, developers can create efficient, reliable, and maintainable embedded C applications suitable for the constrained environments of embedded systems.