JPL Institutional Coding Standard for the C Programming Language

[version edited for external distribution:
does not include material copyrighted by MIRA Ltd (i.e., LOC-5&6)
and material copyrighted by the ISO (i.e., Appendix A)]
Cleared for external distribution on 03/04/09, CL#09-0763.

Version: 1.0

Date: March 3, 2009

Jet Propulsion Laboratory
California Institute of Technology

Table of Contents

Version History

DATE	SECTIONS CHANGED	REASON FOR CHANGE	REVISION
2008-04-04	All	Document created	0.1
2008-05-12	Rule 13	Added guidance for the use of extern declarations, to avoid a known problem.	0.2
2009-03-04	Revision for external distribution	Copyrighted material omitted: LOC-5 and LOC6, and Appendix A	1.0

Acknowledgement

The research described in this document was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Rule Summary

1 Language Compliance	
1	Do not stray outside the language definition.
2	Compile with all warnings enabled; use static source code analyzers.
2 Predictable Execution	
3	Use verifiable loop bounds for all loops meant to be terminating.
4	Do not use direct or indirect recursion.
5	Do not use dynamic memory allocation after task initialization.
*6	Use IPC messages for task communication.
7	Do not use task delays for task synchronization.
*8	Explicitly transfer write-permission (ownership) for shared data objects.
9	Place restrictions on the use of semaphores and locks.
10	Use memory protection, safety margins, barrier patterns.
11	Do not use goto, setjmp or longjmp.
12	Do not use selective value assignments to elements of an enum list.
3 Defensive Coding	
13	Declare data objects at smallest possible level of scope.
14	Check the return value of non-void functions, or explicitly cast to (void).
15	Check the validity of values passed to functions.
16	Use static and dynamic assertions as sanity checks.
*17	Use U32, I16, etc instead of predefined C data types such as int, short, etc.
18	Make the order of evaluation in compound expressions explicit.
19	Do not use expressions with side effects.
4 Code Clarity	
20	Make only very limited use of the C pre-processor.
21	Do not define macros within a function or a block.
22	Do not undefine or redefine macros.
23	Place #else, #elif, and #endif in the same file as the matching #if or #ifdef.
*24	Place no more than one statement or declaration per line of text.
*25	Use short functions with a limited number of parameters.
*26	Use no more than two levels of indirection per declaration.
*27	Use no more than two levels of dereferencing per object reference.
*28	Do not hide dereference operations inside macros or typedefs.
*29	Do not use non-constant function pointers.
30	Do not cast function pointers into other types.
31	Do not place code or declarations before an #include directive.
5 – MISRA *shall* compliance	
73 rules	All MISRA *shall* rules not already covered at Levels 1-4.
6 – MISRA *should* compliance	
*16 rules	All MISRA *should* rules not already covered at Levels 1-4.

*) All rules are *shall* rules, except those marked with an asterix.

Introduction

Considerable efforts have been invested by many different organizations in the past on the development of coding standards for the C programming language. The intent of this standard is not to duplicate the earlier work but to collect the best available insights in a form that can help us improve the safety and reliability of our code. By conforming to a single institutional standard, rather than maintaining a multitude of project and mission specific standards, we can achieve greater consistency of code quality at JPL.

Two earlier efforts have most influenced the contents of this standard. The first is the MISRA-C coding guideline from 2004,[1] which was originally defined for the development of embedded C code in automobiles, but is today used broadly for safety critical applications. The second source is the set of coding rules known as the "Power of Ten."[2] Neither of these two sources, though, addresses software risks that are related to the use of multi-threaded software. This standard aims to fill that void.

This rules included in this standard, and the tools and processes that are used to verify code compliance, should be reviewed for possible revision no more than once per year and no less than once per five years.

Many software experts both inside and outside JPL have contributed to the creation of this document with proposals for good coding rules, and critiques of those contained in earlier standards. Their contributions (which do not necessarily imply the endorsement of this document) are gratefully acknowledged here.

People that have contributed in the preparations for this standard, starting in 2004, include Brian Kernighan (Princeton University), Dennis Ritchie (Bell Labs), Doug McIlroy (Dartmouth), Eddie Benowitz, Scott Burleigh, Tim Canham, Benjamin Cichy, Ken Clark, Micah Clark, Len Day, Robert Denise, Will Duquette, Dan Dvorak, Dan Eldred, Ed Gamble, Peter Gluck, Kim Gostelow, Chris Grasso, Alex Groce, Dave Hecox, Gerard Holzmann, Joe Hutcherson, Rajeev Joshi, Roger Klemm, Frank Kuykendall, Mary Lam, Steve Larson, Todd Litwin, Tom Lockhart, Lloyd Manglapus, Kenny Meyer, Alex Murray, Al Niessner, Bob Rasmussen, Len Reder, Glenn Reeves, Kirk Reinholtz, Mike Roche, Nicolas Rouquette, Steve Scandore, Marcel Schoppers, Dave Smyth, Ken Starr, Igor Uchenik, Dave Wagner, Garth Watney, Steve Watson, Matt Wette, Jesse Wright. Unless otherwise noted, all those above are employees of the Jet Propulsion Laboratory, California Institute of Technology, in Pasadena, California.

[1] MISRA-C 2004, Guidelines for the use of the C language in critical systems. MIRA Ltd. 2004, ISBN 0 9524156 4 X PDF, www.misra-c.com.
[2] The Power of Ten: Rules for Developing Safety-Critical Code, IEEE Computer, June 2006, pp. 93-95.

Scope

The coding rules defined here primarily target the development of *mission critical flight software* written in the *C programming language*. This means that the rules are focused on *embedded software* applications, which generally operate under stricter resource constraints than, e.g., ground software.

For conciseness, the scope of this standard is further restricted as much as possible to the definition of *coding* rules that can reduce the risk of software failures. General project and mission specific requirements that concern the *context* in which software is developed (e.g., process related requirements) but not the code itself, fall outside the current scope. Such additional requirements should be defined and documented separately in accordance with applicable controlling documents from JPL Rules.[3]

The following are some specific examples of process, project or mission specific requirements that fall outside the scope of this standard:

> File and directory organization, naming conventions, formatting, commenting and annotation, the format of file headers (e.g., to document copyright, ownership, and change history), conventions for the use of telemetry channels or event reporting, the development environment (choice of computers, operating systems, compilers, static analyzers, version control systems, build scripts or makefiles, software test requirements, etc).

With few exceptions, general principles of software architecture also fall outside the current scope. A good example of architectural and structuring principles for software systems can be found in the ARINC 653-1 standard for safety critical avionics software.[4]

Conventions

The use of the verbs *shall* and *should* have the following meaning in this document.

- *Shall* indicates a requirement that must be followed, with compliance verified.
- *Should* indicates a preference that must be addressed, but with deviations allowed, provided that an adequate justification is given for each deviation.

An effort is made to limit *shall* rules to cases for which compliance can effectively be verified (e.g., with tool-based checks). If a deviation from a *shall* rule is sought, substantial supporting evidence must be provided in a written waiver request. Such

---

[3] E.g., Software Development Standard Processes, Rev 1, D-74352 and Software Development, Rev 6, D-57653

[4] ARINC Specification, Avionics Application Software Standard Interface, Release 653-1 from 16 October 2003, and Release 653P1-2 from 7 March 2006. Airlines Electronic Engineering Committee, Aeronautical Radio Inc., Maryland, USA.

waiver requests must be evaluated by a team of software experts from across JPL, not associated with the project seeking the waiver.[5]

For each rule given, the most closely related rule in the MISRA-C:2004 standard or the Power of Ten rule-set is quoted.

Levels of Compliance

This standard defines six levels of compliance (LOC), ranging from the most general to the most specific. Compliance with this standard can be certified for each level separately, preferably with the help of tool-based compliance checkers. It is also possible to certify compliance at different LOC levels for different parts of a large code base. For newly written code, achieving full compliance with this standard – at least through level 4, is not expected to have a measurable impact on schedule or cost. This trade-off can be different for heritage code, developed before this standard went into effect. For existing code, the amount of effort needed to achieve compliance will increase with each new level. Schedule and cost considerations, weighed against mission risk, should determine which level is appropriate. Levels of compliance certification for each project or mission should be defined in the project's Software Management Plan (SMP).

The number of rules defined at each LOC is summarized in the following Table. The name of each segment is meant to be suggestive of its approximate purpose.

Level of Compliance	Rules Defined at Level	Cumulative Number of Rules Required for Full Compliance
LOC-1 Language Compliance	2	2
LOC-2 Predictable Execution	10	12
LOC-3 Defensive Coding	7	19
LOC-4 Code Clarity	12	31
LOC-5 MISRA-shall rules	73	104
LOC-6 MISRA-should rules	16	120

The rules defined at LOC-1 through LOC-4 correspond to the following MISRA-C and Power of Ten rules.

Level of Compliance	MISRA-C:2004 Rules	Power of Ten Rules
LOC-1 Language Compliance	1.1, 1.2, 2.3, 21.1	1, 10
LOC-2 Predictable Execution	9.3, 14.4, 16.2, 20.4	2, 3
LOC-3 Defensive Coding	6.3, 8.7, 8.10, 12.2, 13.1, 16.10, 20.3	5, 6, 7
LOC-4 Code Clarity	11.1, 16.1, 17.5, 19.1, , 19.4 19.5, 19.6, 19.12, 19.13, 19.17	4, 8, 9

[5] That is, it will not be sufficient for the cognizant engineer or the project or mission lead to approve a waiver from a *shall* rule. Because these rules are part of a JPL Institutional standard, an independent institutional approval process must be followed for significant deviations.

LOC-1: Language Compliance

Rule 1 (language)
All C code *shall* conform to the ISO/IEC 9899-1999(E) standard for the C programming language, with no reliance on undefined or unspecified behavior. [MISRA-C:2004 Rule 1.1, 1.2]

The purpose of this rule is to make sure that all mission critical code can be compiled with any language compliant compiler, can be analyzed by a broad range of tools, and can be understood, debugged, tested, and maintained by any competent C programmer. It ensures that there is no hidden reliance on compiler or platform specific behavior that may jeopardize portability or code reuse. The rule prohibits straying outside the language definition, and forbids reliance of undefined or unspecified behavior. This rule also prohibits the use of `#pragma` directives, which are by definition implementation defined and outside the language proper. The `#error` directive is part of the language, and its use is supported. The closely related `#warning` directive is not defined in the language standard, but its use is allowed if supported by the compiler (but note Rule 2).

The C language standard explicitly recognizes the existence of undefined and unspecified behavior. A list of formally unspecified, undefined and implementation dependent behavior in C, as contained in the ISO/IEC standard definition, is given in Appendix A.

Rule 2 (routine checking)
All code *shall* always be compiled with all compiler warnings enabled at the highest warning level available, with no errors or warnings resulting. All code *shall* further be verified with a JPL approved state-of-the-art static source code analyzer, with no errors or warnings resulting. [MISRA-C:2004 Rule 21.1]

This rule should be considered routine practice, even for *non*-critical code development. Given compliance with Rule 1, this means that the code should compile without errors or warnings issued with the standard gcc compiler, using a command line with minimally the following option flags:

```
gcc –Wall –pedantic –std=iso9899:1999 source.c
```

A suggested broader set of gcc compiler flags includes also:

```
-Wtraditional
-Wshadow
-Wpointer-arith
-Wcast-qual
-Wcast-align
-Wstrict-prototypes
-Wmissing-prototypes
-Wconversion
```

The rule of zero warnings applies even in cases where the compiler or the static analyzer gives an erroneous warning. If the compiler or the static analyzer gets confused, the code causing the confusion should be rewritten so that it becomes more clearly valid. Many developers have been caught in the assumption that a tool warning was false, only to realize much later that the message was in fact valid for less obvious reasons. The JPL recommended static analyzers are fast, and produce sparse and accurate messages.

LOC-2: Predictable Execution

Rule 3 (loop bounds)

All loops *shall* have a statically determinable upper-bound on the maximum number of loop iterations. It *shall* be possible for a static compliance checking tool to affirm the existence of the bound. An exception is allowed for the use of a single non-terminating loop per task or thread where requests are received and processed. Such a server loop *shall* be annotated with the C comment: `/* @non-terminating@ */`. [Power of Ten Rule 2]

Rule 4 (recursion)

There *shall* be no direct or indirect use of recursive function calls. [MISRA-C:2004 Rule 16.2; Power of Ten Rule 1]

The presence of statically verifiable loop bounds and the absence of recursion prevent runaway code, and help to secure predictable performance for all tasks. The absence of recursion also simplifies the task of deriving reliable bounds on stack use. The two rules combined secure a strictly acyclic function call graph and control-flow structure, which in turn enhances the capabilities for static checking tools to catch a broad range of coding defects.

One way to enforce secure loop bounds is to add an explicit upper-bound to all loops that can have a variable number of iterations (e.g., code that traverses a linked list). When the upper-bound is exceeded an assertion failure and error exit can be triggered. For standard for-loops, the loop bound requirement can be satisfied by making sure that the loop variables are not referenced or modified inside the body of the loop.

Rule 5 (heap memory)

There *shall* be no use of dynamic memory allocation after task initialization. [MISRA-C:2004 Rule 20.4; Power of Ten Rule 3]

Specifically, this rule disallows the use of malloc(), sbrk(), alloca(), and similar routines, after task initialization.

This rule is common for safety and mission critical software and appears in most coding guidelines. The reason is simple: memory allocators and garbage collectors often have unpredictable behavior that can significantly impact performance. A notable class of coding errors stems from mishandling memory allocation and free routines: forgetting to free memory or continuing to use memory after it was freed, attempting to allocate more memory than physically available, overstepping boundaries on allocated memory, using stray pointers into dynamically allocated memory, etc. Forcing all applications to live within a fixed, pre-allocated, area of memory can eliminate many of these problems and make it simpler to verify safe memory use.

> Rule 6 (inter-process communication)
> An IPC mechanism *should* be used for all task communication. Callbacks *should* be avoided. No task *should* directly execute code or access data that belongs to a different task. All IPC messages *shall* be received at a *single* point in a task.

Communication and data exchanges between different tasks (modules) in the system are best performed through a disciplined use of IPC (inter-process communication) messaging. IPC messages should then contain only data, preferably no data pointers, and *never* any function pointers. Each task or module should maintain its own data structures, and not allow direct access to local data by other tasks. This style of software architecture is based on principles of software modularity, data hiding, and the separation of concerns that can avoid the need for the often more error-prone use of semaphores, interrupt masking and data locking to achieve task synchronization.

> Rule 7 (thread safety)
> Task synchronization *shall not* be performed through the use of task delays.

Specifically the use of task delays has been the cause of race conditions that have jeopardized the safety of spacecraft. The use of a task delay for task synchronization requires a guess of how long certain actions will take. If the guess is wrong, havoc, including deadlock, can be the result.

> Rule 8 (access to shared data)
> Data objects in shared memory *should* have a single owning task. Only the owner of a data object *should* be able to modify the object. Ownership *should* be passed between tasks explicitly, preferably via IPC messages.

Ownership equals write-permission, but non-ownership generally will not exclude read-access to a shared object. Note that this rule does not prevent the use of system-wide library modules that are not associated with any one task, but it does place a restriction on how tasks use such modules. Generally, if a shared object does not have a single owning task, access to that object has to be regulated with the use of locks or semaphores, to avoid access conflicts that can lead to data corruption.

> Rule 9 (semaphores and locking)
> The use of semaphores or locks to access shared data *should* be avoided (cf. Rules 6 and 8). If used, nested use of semaphores or locks *should* be avoided. If such use is unavoidable, calls *shall* always occur in a single predetermined, and documented, order. *Unlock* operations *shall* always appear within the body of the same function that performs the matching *lock* operation.

Semaphore *acquire* and *release* operations, when used for locking, and interrupt *mask* and *unmask* operations, should always appear in pairs, within the same function, to

comply with the second part of Rule 9. Semaphore operations can also validly be used for "*producer-consumer*" synchronization. In those cases *acquire* and *release* operations may appear in different tasks. The use of nested semaphore or locking calls in more than one possible order can cause deadlock.

> Rule 10 (memory protection)
> Where available, i.e., when supported by the operating system, memory protection *shall* be used to the maximum extent possible. When not available, safety margins and barrier patterns *shall* be used to allow detection of access violations.

For instance, an area of memory above the stack limit allocated to each task should be reserved as a safety margin, and filled with a fixed and uncommon bit-pattern. A health task can detect stack overflow anomalies by at regular intervals checking the presence of the bit-pattern for each task. The same principle can be used to protect against buffer overflow, or access to memory outside allocated regions. Critical parameters should similarly be protected in memory by placing safety margins and barrier patterns around them, so that access violations and data corruption can be detected more easily.

> Rule 11 (simple control flow)
> The goto statement *shall not* be used. There *shall* be no calls to the functions setjmp or longjmp. [MISRA-C:2004, Rule 14.4, Power of Ten Rule 1]

Simpler control flow translates into stronger capabilities for both human and tool-based analysis and often results in improved code clarity. Mission critical code should not just be arguably, but trivially correct.

> Rule 12 (enum Initialization)
> In an enumerator list, the "=" construct *shall not* be used to explicitly initialize members other than the first, unless all items are explicitly initialized. [MISRA-C:2004, Rule 9.3]

LOC-3: Defensive Coding

Rule 13 (limited scope)

Data objects *shall* be declared at the smallest possible level of scope. No declaration in an inner scope *shall* hide a declaration in an outer scope. [MISRA-C:2004 Rule 8.7, 8.10; Power of Ten Rule 6]

This rule supports a well-known principle of data-hiding. If an object is not in scope, its value cannot be referenced or corrupted. Similarly, if an erroneous value of an object has to be diagnosed, the fewer the number of statements where the value could have been assigned; the easier it is to diagnose the problem. The rule discourages the re-use of variables for multiple, incompatible purposes, which complicates fault diagnosis.

The rule is consistent with the principle of preferring *pure* functions that do not touch global data, that avoid storing local state, and that do not modify data declared in the calling function indirectly. The use of distributed state information can significantly reduce code transparency, reduce the effectiveness of standard software test strategies, and complicate the debugging process if anomalies occur. Good programming practice is further to prefer the use of *immutable* data objects and references. This means that data objects should by preference be declared of C type `enum` or with the C qualifier `const`. Especially function parameters should be declared with the type qualifier `const` wherever possible.

Although their use is sometimes unavoidable, there is a hidden danger in the use of `extern` declarations in C. Without precautions, if we declare a global data object named `x` as type `A` (e.g., `int`) in one source file, and then place an extern declaration to the same object `x` in another source file, while accidentally using another type `B` (e.g., `double`), most current compilers (including gcc with all warnings enabled at the highest setting) and most current static analyzers, will *not* detect the type inconsistency. Clearly, if the two types have different size (as in our example of int and double) havoc will result (mitigated only partially by the use of barrier patterns, as recommended in Rule 10). The correct remedy for this significant flaw in current compiler technology is to:

> *Place all `extern` declarations in a header file. The header file must be included in every file that refers to the corresponding data object: both the source file in which the actual declaration appears and the files in which the object is used.*

If this rule is followed, the compiler will be able to flag all type inconsistencies reliably. Note the similarity in this treatment of `extern` declarations and the standard use of function prototypes (which follows very similar rules).

Rule 14 (checking return values)

The return value of non-void functions *shall* be checked or used by each calling function, or explicitly cast to (void) if irrelevant. [MISRA-C:2004 Rule 16.10; Power of Ten Rule 7]

Rule 15 (checking parameter values)
The validity of function parameters *shall* be checked at the start of each public function.[6] The validity of function parameters to other functions *shall* be checked by either the function called or by the calling function. [MISRA-C:2004 Rule 20.3; Power of Ten Rule 7]

This is consistent with the principle that the use of *total* functions is preferable over non-total functions. A total function is setup to handle *all* possible input values, not just those parameter values that are expected when the software functions normally.

Rule 16 (use of assertions)
Assertions *shall* be used to perform basic sanity checks throughout the code. All functions of more than 10 lines *should* have at least one assertion. [Power of Ten Rule 5]

Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must be side-effect free and can be defined as Boolean tests. When an assertion fails, an explicit recovery action should be taken, e.g., by returning an error condition to the caller of the function. No assertion should be used for which a static checking tool can prove that it can never fail or never hold.

Statistics for industrial coding efforts indicate that unit tests often find at least one defect per one hundred lines of code written. The odds of intercepting defects increase with a liberal use of assertions. Assertions can be used to verify pre- and post-conditions of functions, parameter values, expected function return values, and loop-invariants. Because assertions are side-effect free, they can be selectively disabled after testing in performance-critical code. A recommended use of assertions is to follow the following pattern:

```
if (!c_assert(p >= 0) == true) {
        return ERROR;
}
```

where the assertion is defined during testing as:

```
#define c_assert(e)    ((e) ? (true) : \
        tst_debugging("%s,%d: assertion '%s' failed\n", \
        __FILE__, __LINE__, #e), false)
```

In this definition, __FILE__ and __LINE__ are predefined by the macro preprocessor to produce the filename and line-number of the failing assertion. The syntax #e turns the assertion condition e into a string that is printed as part of the error message. Because in flight there is no convenient place to print an error message, the call to tst_debugging can be turned into a call to a different error-logging routine after testing. In flight, the

[6] A public function is a function that is used by multiple tasks, such as a library function. In a multi-threaded environment, library functions are typically re-entrant.

assertion then turns into a Boolean test that protects, and enables recovery, from anomalous behavior, automatically logging every violation encountered.

The examples above are for *dynamic* assertions that can provide protection against unexpected conditions encountered at runtime. An even stronger check can be provided by *static* assertions that can be evaluated by the compiler at the time code is compiled. A static assertion can be defined like the c_assert above, but can be used standalone (i.e., not in a conditional), for instance as follows:

```
c_assert( 1 / ( 4 – sizeof(void *));
```

This assertion will trigger a "division by zero" warning from the compiler when the code is compiled on 32-bit machines (thus triggering Rule 2). To check the opposite requirement, i.e., to make sure that we are executing on a 32-bit machine only, the following static assertion can be used:

```
c_assert( 1 / (sizeof(void *) & 4) );
```

This version will trigger the "division by zero" warning from the compiler when the code is compiled on machines that do *not* have a 32-bit wordsize.

Rule 17 (types)
Typedefs that indicate size and signedness *should* be used in place of the basic types. [MISRA-C:2004 Rule 6.3]

This rule appears in most coding standards for embedded software and is meant to enhance code transparency and secure type safety. Typical definitions include I32 for signed 32-bit integer variables, U16 for unsigned 16-bit integer variables, etc.

Rule 18
In compound expressions with multiple sub-expressions the intended order of evaluation *shall* be made explicit with parentheses. [cf. MISRA-C:2004 Rule 12.2]

Rule 19
The evaluation of a Boolean expression *shall* have no side effects. [MISRA-C:2004 Rule 13.1]

LOC-4: Code Clarity

Especially mission critical code should be written to be readily understandable by any competent developer, without requiring significant effort to reconstruct the thought processes and assumptions of the original developer. The rules in this section aim to secure compliance with this requirement.

The purpose of these rules is that all code remains readily understandable and maintainable, also years after it is written, and especially when examined under time pressure and by anyone other than the original developer. Code does not just serve to communicate a developer's intent to a computer, but also to current and future colleagues that must be able to maintain, revise, or extend the code reliably. Code clarity cannot easily be captured in a comprehensive set of mechanically verifiable checks, so the specific rules included here serve primarily as examples of safe coding practice.

Rule 20 (preprocessor use)

Use of the C preprocessor *shall* be limited to file inclusion and simple macros. [Power of Ten Rule 8]

The C preprocessor is a powerful obfuscation tool that can destroy code clarity and befuddle both human- and tool-based checkers. The effect of constructs in unrestricted preprocessor code can be extremely hard to decipher, even with a formal language definition in hand. In new implementations of the C preprocessor, developers often have to resort to using earlier implementations as the referee for interpreting complex defining language in the C standard.

Specifically, the use of token pasting (cf. MISRA-C:2004 Rules 19.12 and 19.13), variable argument lists (ellipses) (cf. MISRA-C:2004 Rule 16.1), and recursive macro calls are excluded by this rule. All macros are required to expand into complete syntactic units (cf. MISRA-C:2004 Rule 19.4).

The use of conditional compilation directives (#ifdef, #if, #elif) should be limited to the standard boilerplate that avoids multiple inclusion of the same header file in large projects. (See also Rule 23.) There is rarely a justification for the use of other conditional compilation directives even in large software development efforts. Each such use should be justified in the code. Note that with just ten conditional compilation directives, there could be up to 2^{10} (i.e., 1024) possible versions of the code, each of which would have to be tested – causing a generally unaffordable increase in the required test effort.

Rule 21 (preprocessor use)

Macros *shall not* be #define'd within a function or a block. [MISRA-C:2004 Rule 19.5]

Rule 22 (preprocessor use)

#undef *shall not* be used. [MISRA-C:2004 Rule 19.6]

Rule 23 (preprocessor use)
All #else, #elif and #endif preprocessor directives *shall* reside in the same file as the #if or #ifdef directive to which they are related. [MISRA-C:2004 Rule 19.17]

Rule 24
There *should* be no more than one statement or variable declaration per line. A single exception is the C for-loop, where the three controlling expressions (initialization, loop bound, and increment) can be placed on a single line.

Rule 25
Functions *should* be no longer than 60 lines of text and define no more than 6 parameters. [Power of Ten Rule 4]

A function should not be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function. Long lists of function parameters similarly compromise code clarity and should be avoided.

Each function should be a logical unit in the code that is understandable and verifiable as a unit. It is much harder to understand a logical unit that spans multiple screens on a computer display or multiple pages when printed. Excessively long functions are often a sign of poorly structured code.

Rule 26
The declaration of an object *should* contain no more than two levels of indirection. [MISRA-C:2004 Rule 17.5]

Rule 27
Statements *should* contain no more than two levels of dereferencing per object. [Power of Ten Rule 9]

Rule 28
Pointer dereference operations *should not* be hidden in macro definitions or inside typedef declarations.

Rule 29
Non-constant pointers to functions *should not* be used.

Rule 30 (type conversion)
Conversions *shall not* be performed between a pointer to a function and any type other than an integral type. [MISRA-C:2004 Rule 11.1]

Pointers are easily misused, even by experienced programmers. They can make it hard to follow or analyze the flow of data in a program, especially by tool-based checkers. Function pointers especially can restrict the types of checks that can be performed by static analyzers and should only be used if there is a strong justification, and when alternate means are provided to maintain transparency of the flow of control. If function pointers are used, it can become difficult for tools to prove absence of recursion. In these cases alternate guarantees should be provided to make up for this loss in analytical capabilities.

> Rule 31 (preprocessor use)
> #include directives in a file *shall* only be preceded by other preprocessor directives or comments. [MISRA-C:2004 Rule 19.1]

The recommended file format is to structure the main standard components of a file in the following sequence: include files, macro definitions, typedefs (where not provided in system-wide include files), external declarations, file static declarations, followed by function declarations.

[Levels 5 and 6 omitted in this version for copyright restrictions – consult the original MISRA C guidelines for details.]

References

Primary documents:

Motor Industry Software Reliability Association (MISRA), *MISRA-C: 2004, Guidelines for the use of the C language in critical systems*, October, 2004.

"The Power of Ten -- Rules for Developing Safety Critical Code," *IEEE Computer*, June 2006, pp. 93-95.

International Standard, ISO/IEC 9899:1999 (E) – *Programming Languages – C*, Second Edition, 1999-12-01. Date of ISO approval 5/22/2000. Published by ANSI, New York, NY, 2002, 538 pgs.

The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc. 1978, 2nd Edition 1988.

Other relevant publications and standards:

European Space Agency (ESA) Board for Software Standardization and Control, *C and C++ Coding Standards*, March 30, 2000.

UK Ministry of Defence, Defence Standard 00-55. Requirements for safety related software in defence equipment. Part 2: Guidance, UK Ministry of Defence, Aug. 1997.

DOD-178B, Software Considerations in Airborne Systems and Equipment Certifications, RTCA, Washington D.C., 1992.

Steve Maguire, *Writing Solid Code*, Microsoft Press, 1993.

Andrew Koenig, *C Traps and Pitfalls*, Addison_Wesley, 1989, ISBN 0-201-17928-8.

Jerry Doland and Jon Vallett, *C Style Guide*, Tech. Report SEI-94-003, Software Eng. Branch, Code 552, Goddard Space Flight Center, Aug. 1994.

Les Hatton, *Safer C: Developing software for high-integrity and safety-critical systems*, McGraw-Hill, 1995.

Thomas Plum, *C Programming guidelines*, Plum Hall, 1989, ISBN 0-911537-07-4.

Robert C. Seacord, *Secure Coding in C and C++*, Addison-Wesley, 2005.

David Straker, *C-Style standards and guidelines*, Prentice Hall, 1992, ISBN 0-13-116898-3.

Other documents and standards consulted

Spencer's 10 commandments from 1991 (10 rules)
Nuclear Regulatory Commission from 1995 (22 rules)
Original MISRA rules from 1997 (127 rules)
Software System Safety Handbook from 1999 (34 rules)
European Space Agency coding rules from 2000 (ESA) (123 rules)
Goddard Flight Software Branch coding standard from 2000 (GSFC) (100 rules)
MRO coding rules from 2002 (LMA) (132 rules)
Hatton's ISO C subset proposal from 2003 (20 rules)
MSAP coding rules from 2005 (JPL/MSAP) (141 rules)
JSF AV Rules Rev. C (joint strike fighter air vehicle) from 2005 (154 rules)
SIM Realtime Control Subsystem Coding Rules from 2005 (JPL/SIM) (24 rules)
MSL Coding Rules from 2006 (JPL/MSL) (82 rules)
Rules suggested by JPL developers, 2007 (38 rules)

Appendix A
Unspecified, Undefined, and Implementation-Dependent Behavior in C

As a short synopsis of the basic definition of unspecified, undefined and implementation defined behavior – the following may suffice (based on a definition proposed by Clive Pygott in ISO SC22 in its study of language vulnerabilities:

- **Unspecified behaviour**: The compiler has to make a choice from a finite set of alternatives, but that choice is not in general predictable by the programmer.
  Example: the order in which the sub-expressions of a C expression are evaluated, or the order in which the actual parameters in a function call are evaluated.

- **Implementation defined behaviour:** The compiler has to make a choice that is clearly documented and available to the programmer.
  Example: the range of values that can be stored in C variables of type short, int, or long.

- **Undefined behaviour:** The definition of the language can give no indication of what behavior to expect from a program – it may be some form of catastrophic failure (a 'crash') or continued execution with some arbitrary data.

The following more detailed list is reproduced from Appendix J in ISO/IEC 9899-1999. All references contained in this list are to numbered sections in the ISO document.

[Remainder omitted, for copyright restrictions.]

Index