C/C++ Coding Standard Recommendations for IEC 61508

Version V1, Revision R2

Feb 23, 2011

CONFIDENTIAL INFORMATION

Table of Contents

1. C/C++ Source Code Standard

1.1 Introduction

1.1.1 Purpose and Scope

1.1.1.1 Scope

This source code standard applies to all safety related software that is developed and written in C or C++. Safety related software is normally subject to meeting the IEC 61508 standard (Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems). By conforming to this source code standard, and running a static analysis tool such as PC LINT to enforce the standard, you will be in compliance with IEC 61508 for the C and C++ languages.

1.1.1.2 Purpose

What is the purpose of a coding standard? In the perspective of safety-related software there are two valid answers. A coding standard's primary purpose is to reduce programming errors. A secondary purpose could be to achieve IEC 61508 certification for safety-related software. This IEC standard provides specific guidance on what must be included in a coding standard to provide an adequate level of protection for safety-related software in a specific context of programming language, development process, development toolset and required safety integrity level. The overall purpose is to reduce the number of software defects within a specified development environment. The safety integrity level, SIL, provides an assessment on the techniques and measures needed to reduce software defects for the application.

How does a coding standard help? By identifying certain coding practices as preferred and forbidding other particular types of coding practices, errors can be avoided, made less likely to occur, or made easier to detect. There are several contexts where this applies.

C/C++ programming languages have specific areas where some language features may not be fully specified. This means the code may be interpreted differently depending on the context or the compiler used to generate the runtime code. The interpretation may be very different than what the programmer intended, resulting in errors. There are also other features for which the operation is dependent on factors that are not visible at the level of scope where the code is written and reviewed. The programmer may be completely unaware of the hidden error! These are language related "features" which, if avoided, will reduce program errors. Some of these errors may not show up until long after release of the product. An additional benefit of avoiding context/compiler-specific language features is improved portability of the code.

Another class of problem is where the language may allow a programmer to be less than specific and the compiler interprets what the programmer intended without indicating potential ambiguities or discrepancies. The language may also allow the programmer to perform actions that, in general, do not appear to have a valid reason but the compiler trusts the programmer and implements the specified actions, even though it may be a mistake. This is particularly common in C and also in C++ to a lesser extent.

A third type of situation is where humans, due to the subtleties of the differences, often confuse various legal and fully defined sequences. By explicitly forbidding selected legal sequences within particular contexts, automated tools can detect that either the rule has been violated or a coding mistake has been made that will result in improper operation in some circumstances.

1.1.2 Standards used

This coding standard is based on the following standards.

IEC 61508:2010; Parts 3 and 7, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems [1]

ISO/IEC 9899:1999, Programming languages – C [2]

ISO/IEC 14882:1998, Programming languages – C++ [3]

2. Background

The following describes some background on how particular elements of this coding standard were determined. The coding standard has three sources of input information:

- Specific requirements for a coding standard from IEC 61508
- The result of a gap analysis between the IEC 61508 language selection requirements and the C and C++ programming language standards
- Knowledge of typical coding errors made by C/C++ programmers identified based on published industry knowledge and internal knowledge of best practices

2.1 IEC 61508 Requirements

The following is an abbreviated summary of relevant requirements from IEC 61508:

- 7.4.4.12 states that safety related software development "shall be used according to a suitable programming language coding standard."
- 7.4.4.13 states, "The coding standard shall specify good programming practice, proscribe unsafe language features (for example undefined language features, unstructured designs, etc.) and specify procedures for source documentation. As a minimum, the following information should be contained in the source documentation:
    - o legal entity (for example company, author(s), etc.);
    - o description;
    - o inputs and outputs;

    - configuration management history".

- 7.4.6 note states, "The source code shall: be readable, understandable and testable; satisfy the specified requirements for software module design; satisfy the specified requirements of the coding standard; and satisfy all relevant requirements specified during safety planning"

The first requirement 7.4.4.12 does not provide information relevant to the creation of a coding standard but does show that standard coding standard is required.

Requirement 7.4.4.13 specifies the three areas of required content:

- Specify good programming practice which can be interpreted to be the industry best practice information
- Proscribe unsafe language features (such as the undefined, unspecified, etc.) which will be addressed via the coding standard.
- Requirements for minimum level of source documentation

The last requirement 7.4.6 is not a direct requirement for the coding standard but it is a requirement on the resulting code. Therefore the coding standard and any corresponding static source code analysis tool should insure as much coverage as possible of the portions of these areas relevant to coding.

2.2 C/C++ Programming languages compared to IEC 61508 language selection requirements

2.2.1 IEC 61508 language selection requirements

IEC 61508 Part 3 (Software Requirements) provides clearly defined requirements for the software life cycle for "safety-related software" which applies to any software forming part of a safety-related system or used to develop a safety-related system within the scope of IEC 61508-1 and IEC 61508-2. Specific requirements exist for the language selection for development of safety-related software.

The following is an abbreviated summary of the relevant requirements for language selection:

- 7.4.1.3 states, "The third objective of the requirement is to select a suitable set of tools, including languages and compilers, run-time interfaces, user interfaces, and data formats and representations for the required safety integrity level, over the whole safety lifecycle of the software which assists verification, validation, assessment and modification"
- 7.4.2.4 states the design method chosen shall possess features that "facilitate software modification such as modularity, information hiding and encapsulation"
- 7.4.4.2 is essentially a repeat of 7.4.1.3 covering the select of a suitable set of tools but more specifically spells out more detailed categories for use of tools and specifies the need for these tools to support the entire product lifecycle
- 7.4.4.3 states "the selection of off-line support tools shall be justified".

- 7.4.4.10 states the software shall have a translator assessed for fitness for purpose, shall use only defined language features, shall match the characteristics of the application, shall contain features that facilitate the detection of mistakes, and shall support features that match the design method.
- 7.4.4.11 states that when the 7.4.4.10 requirement cannot be met by the chosen language a further justification is needed which details “the fitness of the language and any additional measures which address any identified shortcomings of the language”

2.2.2 Gap Analysis

The C language can satisfy the first two bullets of section 2.1 and C++ clearly provides additional stronger support of information hiding and encapsulation. The major weakness of C++ in this context is that it allows a programmer to do things that either intentionally or unintentionally violate best practice. Examples include creating classes that may not be well encapsulated, such that data can be accessed via other routes, and that may not be clear to someone performing subsequent software modifications. This can provide a false feeling of security and actually lead to problems.

It is the last two bullets, requirements 7.4.4.10 and 7.4.4.11, in combination with ISO/IEC 9899:1999, Programming languages – C [2] and ISO/IEC 14882:1998, Programming languages – C++ [3], that represent the major limiting factors for language selection, the restrictions on its subsequent usage, and the primary basis of the gap analysis. These IEC Programming language specifications identify four specific types of behavior issues for both C and C++ that need to be addressed for IEC 61508:

- Unspecified behavior
- Undefined behavior
- Implementation-defined behavior
- Locale-specific behavior

These four types of behavior are more fully explained in the following subsections 2.2.2.1 through 2.2.2.4. Annex G of ISO/IEC 9899:1999, Programming languages – C provides easy consolidation of the relevant information for C. For C++, the references for these behaviors are spread out within ISO/IEC 14882:1998, Programming languages – C++ without an easy to reference annex but use of the Index provides a systematic method for identification of the relevant issues for these categories.

2.2.2.1 Unspecified behavior

Unspecified behavior refers to language constructs that must compile successfully, but allow the compiler writer some freedom as to what the construct does. There also is no requirement for the compiler to even be internally consistent in its behaviors. An example relevant to both C and C++ is the 'order of evaluation'.

This is particularly important to know so that the programmer truly understands how the code will be interpreted and executed. It is also a key factor in successful reuse of existing code for a new project since code containing these features may not act in the same manner if a different target system or a different compiler is used.

2.2.2.2 Undefined behavior

Undefined behavior refers to situations where a programmer can basically violate strict use of the language but no errors are required to be generated. An example is passing invalid parameters to functions based on the type of data being passed or an incorrect number of parameters in the C language.

This is important for the last part of requirement 7.4.4.10 which states, “contain features that facilitate the detection of programming mistakes”. These are opportunities to catch potential programming mistakes where the programmer either mistyped something or perhaps did not fully understand the situation and therefore is not properly using other program elements as intended by their design.

2.2.2.3 Implementation-defined behavior

Implementation-defined behavior refers to situations where the interpretation needs to be defined and documented by the compiler manufacturer, such that they are consistent within a particular compiler but may vary between compilers making the code non-portable. An example of this is the behavior of the integer division and remainder operators “/” and “%” when applied to one positive and one negative integer.

Similar to the unspecified behaviors, these are potentially important issues with portability of code and therefore an issue when considering the reuse of existing code on a new project with safety-related software.

2.2.2.4 Locale-specific behavior

Locale-specific behaviors are built into the language, resulting in operation which may vary with international requirements (such as using ',' character instead of the '.' character to indicate a decimal point, or a time/date format).

The relevant issue: this is an independent factor that is capable of modifying the behavior of the code which is independent of the visible code and perhaps unknown to the programmer and test groups. Any locale-specific environments must be defined in the coding standard.

2.3 Knowledge of typical Coding Errors Made by C/C++ Programmers

Beyond the undefined behaviors described in the previous section, C and C++ program code tends to result in particular types of software bugs where the language is well defined, but frequently misunderstood [4]. Much of the information about common software bugs comes from actual industry experience with the languages and represents “industry best practice information”. Although this somewhat overlaps the area of undefined behaviors which can be addressed by prohibiting their use, additional measures can go even further in terms of error prevention. By recognizing particular programming patterns that have been shown to lead to coding errors, such measures can prevent problems even where the language is fully defined and explicit. Simple examples are areas where the addition or deletion of a single character from valid source code can result in a second valid expression with dramatically different meaning such as “=” versus “==”, “>=”, etc. In this case an additional restriction against using the assignment operator within logical expressions can result in both more readable code and an easy means for automated detection.

Another area is the use of pointers and dynamic memory allocation, which often represent an inordinately high number of software bugs that are often difficult to detect and remove. They are especially dangerous as they can result in total system failure. There is often a long time element involved in detection such that they can make it through verification and validation activities. Again, restrictions in use of technically valid programming practices can go a long way in both preventing and detecting errors of this type.

2.4 Principles Utilized

Based upon the known weaknesses of the C / C++ programming languages and the availability and capability of commercial tools, the following broad principles were selected to best mitigate the weaknesses.

2.4.1 Require programmers to be explicit

Many potential issues arise from code that may be ambiguous or otherwise misunderstood by either human reviewers or language compilers. For humans, details such as the precedence of operations are not necessarily logical and self-evident. The confusion can be further compounded by inconsistencies between various programming languages. For compilers, ambiguity stems from a different source: the language standard. If there are areas of the language syntax that are not completely and unambiguously defined, the various compiler vendors may create products which interpret the code in perfectly reasonable but different ways, such as the order of execution of multiple statements within a complex C/C++ expression. Wherever ambiguity for either humans or language compilers exists, it is necessary for the code to be written in a way to remove these ambiguities. For the precedence of operations example, explicit use of parentheses to group the expression will reduce ambiguity without any assumptions on precedence of operators. For a sequence of execution example, clarity is enhanced by placing individual expressions, which represent complete operations, on different lines of source code in the proper order such that the sequence of operation is clearly indicated to both humans and the compilers.

2.4.2 Use strong data typing

Strong data typing is fundamental to detection of many types of faults due to non-explicit code. This feature is built into most languages but C and C++ are somewhat lax in detection and enforcement of this feature. The language does detect the inconsistencies but in place of annunciating this to the programmer it applies assumptions and decides how it believes the programmer intended to convert between the data types. However, some compilers and especially C/C++ static source code analysis tools can be set to enable a stronger level of checking, producing an explicit list of these data type coding inconsistencies.

In order to overcome this weakness of C/C++ the coding standard will specify the code to always use the proper consistent data type or explicitly request the type of conversion desired, therefore any mismatch will indicate an error.

2.4.3 Trust the programmer when explicit expressions are used

When the code is explicit and non-ambiguous, the standard and process will start with the assumption that the programmer understands the code created and intended the operations explicitly documented by the code. If however the programmer is not explicit, the assumption is that the ambiguities were not recognized and need to be pointed out to the programmer to insure the intended functionality is implemented. Code reviews might more easily verify whether the code will perform the intended functionality, given that the code itself is clear and non-ambiguous.

For example, explicit conversion or casting by the programmer to the expected data type is considered to be correct since it indicates the programmer intended to mix the data types and is explicitly controlling how this is done. The one exception occurs when the explicit casting itself would result in a confusing or undefined situation such as can happen with the casting of pointers.

2.4.4 Ban language features or take other actions to mitigate issues not correctable by explicit programming

Requiring the programmer to be fully explicit takes care of many of the “undefined” issues but it cannot fully address many of the “unspecified” or “implementation-specific” issues that are related to the differences within and between compilers. In these cases there are only two choices and both are recognized by IEC 61508. These features may either 1) be banned from use or 2) be defined by a systematic method in the development process to verify consistency across the identified set of tools used. This is again a prime area of concern that especially needs to be addressed in reuse of existing code.

2.4.5 Ban explicit language features that lead to or hide coding errors

C/C++ allows for very complex expressions to be utilized within one line of code that may be fully explicit but yet easy to misinterpret. The addition or removal of a single character can change one valid expression into another valid expression that is completely different. A simple example for this type of potential problem is the use of the assignment operator “=” within a logical expression consisting of mostly Boolean terms and comparison operators such as “==” or “<=”. In this case, it is clear that the addition of an extra “=” character can change an assignment into a comparison or the elimination of a “=”, “>”, or other characters can change a logical comparison into an assignment statement.

By banning the use of the assignment operator within Boolean expressions, the occurrence of an “=” is either a programming error where the programmer forgot a character or a violation of the source coding standard. To a lesser extent, this also falls into the category where humans may misunderstand the intent and introduce bugs in the maintenance phase of the product.

Best practices in this area also include the detection, correction or elimination of code constructs that, if eliminated, would have no effect on the operation of the program. Examples include flow control branches that will never be executed or others that are always executed since even though there appears to be a decision point, the decision is predetermined within the existing code independent of what happens at runtime. A specific example is a branch based on the test variable being less than zero when the variable type is an unsigned integer that can never be negative; this most likely indicates either a coding or design logic error. There is a very good chance that the programmer intended for these decisions to be active and was not aware of the actual operation. This potentially subtle “bug” is difficult to detect.

2.4.6 Use automated static code analysis for verification and expert programmer guidance

A static code analysis tool can detect non-compliances to the coding standard. This type of tool can be effectively utilized at multiple points within the development process. First, it can provide informal and automated compliance checking for the programmer at the time of coding. Second it can be used for independent verification of official compliance to the coding standard, which is a formal use of the tool where it is only used to verify compliance to the standard and nothing more.

Using the static code analysis tool at the time of writing the code has additional benefits to the programmer. It can also detect many less concrete questionable patterns in the code that have been recognized as typical of coding errors. Some tools can also be quite effective at identifying situations where there may be potential memory leaks, such as reassigning a pointer pointing to dynamically allocated memory to something new without freeing the original memory block. Using the static source code analysis tool in this way provides not only quick feedback to the programmer on meeting the official coding standard but also provides a quick check list of where hard to find bugs may be hiding. It is a tool to help the programmer write better code quickly and become a better programmer with private direct feedback before anyone else even sees the code.

Depending on the design process used, a static source code tool can also verify the detailed design is compliant to the high level design specified. More details are provided in the related exida.com document “IEC 61508 Certification of Safety-Related Software Products with C/C++” [R7].

3. Coding Standard

There are two primary areas required of a coding standard for IEC 61508 compliant development of safety-related software. First, there are minimum requirements for documentation of the source code. Second, there is a language subset that has been defined to reduce the probability of introducing programming faults and increase the probability of detecting any remaining faults

3.1 General requirements for source documentation

Source documentation, which needs to contain at a minimum:

- legal entity (for example company, author(s), copyright, etc.)
- description

- inputs and outputs
- configuration management history

3.2 Rules for language subset of C/C++

The following rules were developed to specifically address the known weaknesses of the C and C++ programming languages based on the principles presented in section 2.4:

1. The code shall not contain any syntax errors.
2. All functions, classes, data types, macros, variables and constants shall be uniquely and fully declared once (and only once) and subsequently implemented, initialized and used fully consistent with their declaration meaning:
   a. functions shall have a complete prototype that explicitly specifies a return type (or void), and a definite number of parameters that are fully and uniquely defined including name and data type,
   b. all classes shall:
      i. have a constructor and destructor, which shall be virtual if the class is a base class;
      ii. have a specified access level for every member of the class;
      iii. only permit private and protected data to be accessed by the class's member functions, and
      iv. only have member functions which conform to the function requirements of part a.,
   c. all identifier names shall be unique across all namespaces within the first 31 characters,
   d. all macros shall be unique, complete and unambiguous independent of their end application, and
   e. all numerical constants shall be unique, complete and fully compatible with the intended data type.
3. All arithmetic expressions and assignment statements shall be:
   a. non ambiguous and explicit in terms of precedence, and
   b. internally consistent in data types with explicit casting to indicate programmer intent.
4. All logical expressions shall:
   a. be fully explicit in terms of precedence,
   b. be fully consistent in use of data types (i.e.: not compare signed and unsigned numbers),
   c. not include assignment statements,
   d. not include function calls that have side effects, and

e. not depend on exact precision of floating point numbers (i.e.: do not perform “==” or “!=” comparisons on floating point numbers)

5. All pointer expressions shall:

a. only be used after initialized (non-NULL),

b. in a manner consistent with preserving strong data typing & non ambiguous operations

c. only be used if pointer data type and value range are checked before access, and

d. not be used for:

i. pointer arithmetic,

ii. operations involving mixing of different types, or

iii. pointing to objects with a different level of scope or persistence (i.e.: autovariables from outside the function, private or protected data functions within a class from outside the class)

6. The code shall not use non-structured flow control such as jump statements including: break outside of a case in a switch statement, goto, setjmp, longjmp, or continue.

7. All switch statements shall:

a. be fully consistent in data types,

b. not use Boolean data type,

c. contain a default case,

d. contain at least one case, and

e. terminate all non-empty cases with a break.

8. Bit fields, bit representation and bit-wise operators shall:

a. only be used/applied to unsigned integer data types and

b. must be subsequently utilized to be fully consistent with their defined type and storage capacity.

9. Implementation defined or unspecified C or C++ features such as but not limited to those examples shown below shall only be used in conjunction with an approved process which insures all compliers, linkers and other tools used have been tested and verified for consistent and unambiguous implementation of the feature across the entire defined set of software tools utilized:

a. interfaces to other languages, including assembly,

b. character values other than the set defined by the C++ Programming standard. The set shall not include character types being used to represent numeric information other than character symbols,

c. the use of the characters ‘/’, ‘\’ “ ‘ “, or ‘*’ in the names of header files, and

d. other C/C++ implementation defined or unspecified features per the standard documents.

10. The following error prevention guidelines should be used:

   a. trigraphs shall not be used,
   b. braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures,
   c. in an enumerator list, the '=' shall not be used to explicitly initialize members other than the first unless all items are explicitly initialized,
   d. the return values of functions shall not be ignored,
   e. the #include statement in a file shall only be preceded by other pre-processor instructions,
   f. all uses of #pragma shall be documented and justified,
   g. header files shall not be included more than once,
   h. unions shall not be used to access the subparts of larger data types,
   i. each object shall be declared on a separate line,
   j. functions, parameters ext. shall be declared const whenever possible,
   k. the comma operator shall not be used outside of the loop control expression of a for loop,
   l. there shall not be any unreachable code, and
   m. all non-NULL statements shall have side effects.
   n. dynamic memory allocation shall only be used if there is not enough memory available to statically allocate all variables, arrays, and objects
   o. if dynamic memory allocation is used, the return value of the memory allocator should be checked to ensure the allocation was successful. If it is not successful, an appropriate safe action should take place
   p. interrupts should only be used if they simplify the system
   q. software handling of interrupts should be inhibited during critical parts of the program
   r. Interrupt usage, masking, and all non-encapsulated data that can be modified by interrupt service routines should be thoroughly documented
   s. if recursion is used, there must be a clear criterion which makes predictable the depth of recursion

11. All code, including code libraries, utilized within the software project shall be compliant to these rules except as specifically justified, documented and approved through an approved process specified by Functional Safety Management plan.

3.2.1 Relationship of rules to types of modules

The rules defined above may not always apply to pre-existing code. **Table 1** summarizes when the rules must be applied and more details are given in this section.

Table 1 Summary of when rules of coding standard apply

Rules	**New Code**	**Existing Code (Proven in Use)**	**Existing Code (Not Proven in Use)**
1 and 2	Required[1]	Desired[2]	Required[1]
3 through 8	Required[1]	Not Required	Desired[2]
9	Required[1]	Not required as long as same versions of tools are used to compile and build the software	Desired[2]
10	Required[1]	Not Required	Not Required
[1] Required elements can be overridden with proper justification and management signoff. Justification for required elements involves an argument as to why the cited problem is not ambiguous in this environment. [2] Desired elements can be overridden with proper justification and peer review. Justification for desired elements involves an assessment as to whether fixing the problem would be more likely to inject a new problem than it would be to solve or prevent a problem.			

3.2.1.1 Use of rules for development of new code

Rules 1 and 2 are basic requirements for effective use of C or C++ within a controlled process and must be met for all safety-related software development. These are rules that are automatically enforced in many languages and address many of the undefined conditions of C and C++ (newer ANSI compliant C++ compilers may already enforce many of these rules).

Rules 3 through 8 are required for new development. These rules further address the need for code to be completely and unambiguously defined within a specific development environment by primarily addressing the “unspecified” issues. In addition to language specification issues and being non-ambiguous to the compiler these requirements also begin to address issues related to being explicit for human reviewers.

Rule 9 addresses implementation-defined language features which may become issues with code portability (very important for proven in use evaluations) and even programmer portability (if a programmer is only familiar with how one particular compiler operates). It also covers a few remaining unspecified behaviors that will most likely be handled by validation of the development tool set versus not using some highly desirable language features. Any subsection of rule 9 not followed in the development process needs to be formally documented including the other means taken to insure this does not create any safety-related problems.

Rule 10 provides highly recommended suggestions that significantly enhance the error prevention and error detection capabilities of the C and C++ languages. These rules should be followed whenever reasonable.

3.2.1.2 Use of rules for existing code (proven in use)

For code that can be proven in use, the following minimum conditions should be met:

1. Use static code analysis to demonstrate adherence to rules 1 and 2.
2. Use static code analysis to identify any use of features relevant to rule 9 only if a different compiler is being used than the version that was certified to be proven-in-use.

3.2.1.3 Use of rules for existing code not proven in use

Existing code with insufficient documentation history for proven in use should follow the same guidelines for new code except that compliance to rule 10 is not recommended. However, automated static source code analysis reports including rule 10 issues would be strongly recommended input to code reviews for this code to facilitate discovery of any potential hidden problems. In addition other selected features of rules 3 through 8 that are related to programming "style" may be excluded by written justification if the effort to fix "style" issues is likely to cause new safety-related issues such as adding explicit casting to existing code that is known to operate correctly but where the original programmer intent is not clearly known. Explicit use of parentheses for precedence of operation is another example. If such justifications are made however these sections need to be clearly identified and such sections cannot be modified in the future without bringing them up to the new code standard with full testing at all levels.

3.3 Rules for modular approach

Clause 7.4.2.4 of part 3 of IEC 61508 states the language chosen shall possess features that "facilitate software modification such as modularity, information hiding and encapsulation. In order to meet that clause the following rules should be applied to software module design:

a. A software module shall have a single, well-defined task or function to fulfill

b. Connections between software modules should be limited and strictly defined, coherence in one software module should be strong

c. Modules should be restricted to no more than 100 lines of uncommented source code.

d. Modules should be restricted to a cyclomatic complexity of 10 or less.

e. Functions should have a single entry and single exit only

f. Software modules should communicate with other software modules via their interfaces – where global common variables are used they should be well structured, access should be controlled, and their use should be justified in each instance.

g. All software module interfaces should be fully documented

h. Any software module's interface should contain only those parameters necessary for its function

Any code that is proven-in-use is not subject to these rules. However, if significant design changes are being made to proven-in-use modules, it is suggested that these rules be applied to the re-design. All new modules should meet these requirements. However, if there is a compelling reason to violate one of these rules for a new module, then that reason should be clearly documented.

3.4 Rules for Structured Programming

The goal of structured programming is to implement a program that can be analyzed well without executing the program. This will facilitate more problems being found in the code review/inspection stage of the process. Structured program builds on the modular approach and adds concepts that limit complexity. The following rules should be followed in order to meet the goals of structured programming:

a. follow the rules for a modular approach as defined in section 3.3
b. follow the rules for a C/C++ subset as defined in section 3.2
c. keep the number of possible paths through a software module small, and the relation between input and output parameters as simple as possible.
d. Avoid complicated branching
e. Where possible, relate loop constraints and branching to input parameters.
f. Avoid using complex calculations as the basis of branching and loop decisions

Any code that is proven-in-use is not subject to these rules. However, if significant design changes are being made to proven-in-use modules, it is suggested that these rules be applied to the re-design. All new modules should meet these requirements

4. References

[1] IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010, International Electrotechnical Commission, Geneva, Switzerland (2nd Edition)

[2] ISO/IEC 9899, Programming languages – C, 1999

[3] ISO/IEC 14882, Programming languages – C++, 1998

[4] Hatton, Les, Safer C: Developing Software for High-integrity and Safety-critical Systems, 1995, New York, NY, USA, McGraw-Hill Book Company.

[5] IEC 61508 Certification of Safety-Related Software Products with C/C++, Version V1, R1.1 August 28, 2001, exida.com LLC, Sellersville, PA, USA

[6] Straker, D., C - Style Standards & Guidelines, 1992, New York, NY, USA, Prentice Hall.

[7] Guidelines For The Use Of The C Language In Vehicle Based Software, MISRA, April 1998

[8] Nuclear Regulatory Commission Guidelines for Safe Use of Standard Programming Languages, previously located at http://www.nrc.gov/NRC/NUREGS/CR6463/index.htm

5. Notes

5.1 Side Effects

A "*side effect*" is a change in the status of a program often caused by the actions of a subprogram. Such status change cannot be anticipated easily by a casual reading of the subprogram's header or a statement that calls upon the subprogram. Side effects are usually undesirable because their presence complicates understanding of the totality of a program's actions. Side effects also work against software reusability, a major goal of object oriented programming (OOP). Further, they are almost always easily avoided. Unfortunately, a mature culture of programming in languages such as C++ and Visual Basic has arisen in which, in many circles, it's considered alright to use side effects. Intended side effects can be short-circuited by run-time operation of certain statements ("if (A || B)… ; in this case, B may not be evaluated and any side effects associated with B may not be seen.)

Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence, called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects. The floating-point environment library **<fenv.h>** provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

An object that has volatile-qualified types may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine.

The order in which any side effects occur among the initialization list expressions is unspecified.

Macro side effects are inevitable if macro appears more than once. Ensure operator precedence in macros as well.

The order in which arguments to functions and operands to most operators are evaluated is unspecified. The evaluations may even be interleaved.

Beware global variables and non-void functions.

Null statements have no side effects, but they do not provide meaningful functionality and should be avoided (e.g. "if (a < 5);", "sqrt (5);". Such statements have no assignments and the results are not used. Most compilers would not generate executable code, but they may not indicate warnings or errors for these statements.

6. Status of the document

6.1 Releases

Version:	V1	
Revision:	R2	
Review:	V1, R2:	reviewed by: M. Medoff, J. Grebe, Feb. 22, 2011
Review:	V1, R1.2:	reviewed by: M. Medoff, J. Grebe
Version History:	V1, R2	J. Yozallinas; changes to agree with the new “61508 Compliant Process” book; added notes about side effects; Feb. 23, 2011
	V1, R1	Added missing requirements from IEC 61508; Jan. 22, 2010
	V0, R1.0:	Proposal Nov. 27, 1999
	V0, R2.0:	Update of rules for ease of interpretation July 23, 2002

Authors: John C. Grebe Jr., Michael Medoff

Release status: released