Climate Data Record (CDR) Program

General Software Coding Standards

CDR Program Document Number: CDRP-STD-0007
Originator Document Number: N/A
Revision 1.0 June 29, 2012
DSR Number: 004

REVISION HISTORY

Rev.	Author	DSR No.	Description	Date
1.0	Brian Newport, Global Science and Technology, Inc	DSR-004	Initial Delivery	06/29/2012

TABLE of CONTENTS

LIST of TABLES

LIST of EXAMPLES

1. Introduction

1.1 Purpose

The Climate Data Record (CDR) Program receives scientific algorithms in the form of source code that will be deployed in a full-time operational setting for the purpose of ongoing processing of new data, and for the purpose of reprocessing existing data. These source codes are written in various programming languages and styles and often lack coordinating documentation. The resulting software is often costly to maintain, since the code may be difficult to read and understand; supporting documentation may be inadequate; and the original developers may no longer be available to help maintain their code.

The purpose of this document is to define coding standards that will streamline the transition of CDR algorithms from research to Initial Operational Capability (IOC), and subsequently to Full Operational Capability (FOC). Implementation of these standards is evaluated as part of the Software Readiness dimension of the CDR Maturity Matrix (CDR-MTX-008) in conjunction with the more detailed CDR Maturity Evaluation Guidelines (CDR-GUID-0020), and will shift costs away from operations and maintenance as problems are resolved earlier in the software development life cycle. Promoting the accountability of scientists and software developers to create standardized software programs will benefit both the CDR Program and the research teams in the long run.

1.2 Audience

The principal audience for this document includes any research or development team providing software to the CDR Program. This includes Principal Investigators and other algorithm developers developing code for IOC, and all scientists and software developers involved in the transition of algorithms from IOC to FOC.

1.3 Scope

Coding standards are applicable to software development in any domain. The standards in this document were selected based on their particular relevance to the scientific data processing needs of the CDR Program. Examples are mostly in Fortran, but have analogs in other languages.

This document focuses on the code as written and delivered, and does not address other software engineering activities, such as the software development life cycle, project management, configuration management, requirements, architecture, design, integration, verification, validation, or quality assurance. However, Section 4 does discuss some aspects of the coding process.

1.4 Notation

1.4.1 Standards, Guidelines, and Recommendations

The CDR Program recognizes that many stylistic suggestions are subjective, and therefore should not have the same importance as techniques and practices that are known to improve code quality at run time and its maintainability in the future. For this reason, these standards are divided into three categories, which become progressively more important as the level of maturity increases:

Standard: Compliance with this category is required to achieve CDR Maturity Level 3 (IOC) and all subsequent levels of maturity.

Guideline: Compliance with this category is required to achieve CDR Maturity Level 5 (FOC) and all subsequent levels of maturity, and strongly encouraged at earlier levels of maturity. Non-compliances at FOC will need to be argued in writing by the developer, and recorded as a waiver if approved by the CDR Program.

Recommendation: Compliance with this category is desirable at CDR Maturity Level 5 (FOC) and all subsequent levels of maturity.

These three categories will be found in the above format throughout this document. If possible, all standards, guidelines and recommendations should be followed, keeping in mind their increasing importance at increasing levels of maturity.

1.4.2 Programming Language Neutrality

This document strives to use terminology that is programming language-neutral despite the variations that occur between different languages, and also takes into account the variations in terminology that occur across the field of software engineering. The most important definitions for this document are defined in the Glossary.

1.5 References

1.5.1 Applicable Documents

The following documents are applicable to the development and preparation of this document.

Document Title	Reference
Climate Data Record (CDR) Maturity Matrix	CDRP-MTX-0008 V4.0 (12/20/2011)
Climate Data Record (CDR) Maturity Evaluation Guidelines	CDRP-GUID-0020 V2.0 (8/4/2011)

1.5.2 Reference Documents

This document is based in part on the following sources, as well as lessons learned as a result of the CDR Program Office staff experience in moving scientific code to operations. Additional sources are given in Appendix B.

Document Title	Reference
Software Coding Guidelines	Clouds and the Earth's Radiant Energy System (CERES) Data Management System (DMS), 2008. http://science.larc.nasa.gov/ceres/SCG/SCG_V2.pdf [CERES 2008]
The Power of 10: Rules for Developing Safety-Critical Code	Holzmann, Gerard J., IEEE Computer, June 2006. http://spinroot.com/gerard/pdf/Power_of_Ten.pdf [Holzmann 2006]
Code Complete, 2nd Edition	McConnell, Steve, , Microsoft Press, 2004. [McConnell 2004]
Guidelines for the Use of the C Language in Critical Systems, 2nd Edition	Motor Industry Software Reliability Association, MISRA-C:2004, 2nd Edition, 2008. [MISRA 2008]
Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric	Watson, Arthur H., and McCabe, Thomas J., NIST Special Publication 500-235, August 1996. [NIST 1996]
Standards, Guidelines, and Recommendations for Writing Fortran 77 Code, Version 2.0	NOAA Satellite Products and Services Review Board (SPRSB), 2010, (Approval Pending). http://projects.osd.noaa.gov/spsrb/standards_docs/general_standards_v2.0.docx [SPRSB 2010]

2. Computing Platform

2.1 Hardware

It is expected that the CDR processing will be performed on 32-bit or 64-bit machines using the IEEE Standard for Floating-Point Arithmetic (IEEE 754).

2.2 Operating Systems

It is expected that the CDR Program will use a Unix-like operating system to produce Climate Data Records. However, the exact distribution and version is unknown and is likely to vary. This document includes standards that address portability between different environments.

2.3 Languages

Table 1 below defines the acceptable programming languages for algorithms supplied to the CDR Program. For each language the table also shows the corresponding standard, and the earliest version of a free compiler for that language that can be used to verify compliance with the algorithm submission standards below. The CDR Program has made no assessment of the extent to which any of these compilers complies with the relevant standard, and may use different compilers or different versions than those shown.

Table 1: Acceptable languages for algorithms supplied to the CDR Program.

Language	Standard or Documentation	Free Compiler and Options
C	ISO/IEC 9899:1990 (aka ANSI C; C90)	gcc 4.1.2 or later -ansi –pedantic –Wall
C++	ISO/IEC 14882:1998 as amended by ISO/IEC 14882:2003	gcc 4.1.2 or later -ansi –pedantic –Wall
FORTRAN 77	X3J3 http://www.fortran.com/F77_std/rjcnf0001.html	gcc 4.1.2 or later -Wall
Fortran 95	ISO/IEC 1539-1:2010, 1539-2:2000, 1539-3:1999	gcc 4.1.2 or later -std=f95 –pedantic –Wall
IDL	http://www.ittvis.com/language/en-us/productsservices/idl.aspx	N/A
Java	Gosling et al., The Java Language Specification, Third edition, Sun Microsystems (1996)	gcc 4.1.2 or later -Wall
Perl	http://www.perl.org/	Perl 5.12 or later -w

Language	Standard or Documentation	Free Compiler and Options
Python	http://www.python.org/	Python 2.7 or later Python 3.2 or later
Shell	http://www.gnu.org/software/bash/manual/	bash-4.1 or later

3. Algorithm Submission Standards

3.1 Programming Language

Standard: Each source file submitted to the CDR Program shall be coded in one of the languages specified in CDRP-STD-0007 "CDR Program General Software Coding Standards", Table 1: Acceptable languages for algorithms supplied to the CDR Program. [CDRP-STD-0007:001]

Rationale: Permits the use of free or proprietary compilers and restricts the choice of scripting languages.

Standard: Source code shall not use any language extensions beyond the standards listed in CDRP-STD-0007 "CDR Program General Software Coding Standards", Table 1: Acceptable languages for algorithms supplied to the CDR Program. [CDRP-STD-0007:002]

Rationale: Portability. Although some compilers may "add value" with various language extensions, licensing and other issues may unnecessarily constrain the choice of operational environments.

3.2 Completeness

Standard: Each source code package delivered to the CDR Program shall contain the complete source code needed to build the application executable(s). [CDRP-STD-0007:003]

Standard: Each submitted source file shall compile and link (or be interpreted) with no errors when using the compiler or interpreter and options (if any) for that language specified in CDRP-STD-0007 "CDR Program General Software Coding Standards", Table 1: Acceptable languages for algorithms supplied to the CDR Program. [CDRP-STD-0007:004]

Rationale: Consistent with the CDR Program expectation that the submitted code is the actual code that was used to produce the data product submitted for Initial Operational Capability (IOC).

Guideline: Each submitted source file should compile and link (or be interpreted) with no warnings when using the compiler or interpreter and options (if any) for that language specified in CDRP-STD-0007 "CDR Program General Software Coding Standards", Table 1: Acceptable languages for algorithms supplied to the CDR Program. [CDRP-STD-0007:005]

Rationale: A compiler warning should be fixed even if the developer believes it is erroneous. The developer may be confused and may later realize that warning was accurate. Fixing all warnings also relieves the burden on subsequent maintainers. See [Holzmann 2006], Rule 10.

Standard: Each source code package delivered to the CDR Program shall contain support for an automated build of the application executable(s). [CDRP-STD-0007:006]

Rationale: In most cases it is tedious and error prone to build an application using manual compilation and linking. This standard supports the automated build process that will be needed during the transition from IOC to Full Operating Capability (FOC), and could consist of a configure script plus a top level script or make file driving a combination of make files, or other automation tools.

Standard: Each source code package delivered to the CDR Program shall contain software and/or documentation with explicit instructions on how to obtain or create the complete set of supporting input files needed by the application, such as lookup tables and other ancillary files. [CDRP-STD-0007:007]

Rationale: Essential for the CDR to be reproduced successfully by a third party in the absence of the original development team.

Standard: Each source code package delivered to the CDR Program shall contain a README file that contains complete instructions in English for building, testing, and running the application, and any supporting software included in the package. [CDRP-STD-0007:008]

Rationale: Essential for the CDR to be reproduced successfully by a third party in the absence of the original development team.

Recommendation: It is recommended that the README file contain a citation that can be copied and pasted, in the same format as this example [CDRP-STD-0007:009]:

> Hayes, B., B. Tesar, and K. Zuraw, 2003: OTSoft: Optimality Theory Software (Version 2.1) [Software]. Available from http://www.linguistics.ucla.edu/people/hayes/otsoft/

Rationale: Offers a third-party user of the CDR code a convenient and repeatable method for referencing the code in any paper they may write.

3.3 Unused Code

Guideline: Unused variables, unused statements, unused routines, and unused files should be removed prior to submission to the CDR Program. [CDRP-STD-0007:010]

Rationale: Unused code increases the cost of maintenance by: increasing the amount of effort needed for comprehension; giving false hits on searches, and creating a risk that unused code will be out of synchronization if variable names or interfaces are changed elsewhere.

3.4 Symbolic Links

Standard: The source code file layout shall not contain any symbolic links ("soft links"). [CDRP-STD-0007:011]

Rationale: Although symbolic links have important uses at the system level they must be avoided in the source code area. Modern compilers offer features such as search paths that eliminate the need for symbolic links to header files. Other source code needed in two different places should be factored out as a shared file or library.

3.5 Deliverable Package

Standard: The delivery package shall be a gzipped tar archive, with the top-level application source directory as the root path element within the tar file (i.e. no path elements above the root level directory are found in the tar file). [CDRP-STD-0007:012]

Rationale: Ensures that the various submitted codes have a consistent layout at the top level. This can be achieved by executing the command tar -czf <tar file name> <srcdir>

Standard: The delivery package file name shall conform to the naming convention agreed to in the Submission Agreement, of the general form:
`<CDR title>_<application>_<version>.tar.gz` [CDRP-STD-0007:013]

Rationale: Ensures that the various submitted codes have a consistent and unique naming convention.

Standard: Source code packages delivered to the CDR Program shall not contain any compiled code such as object files and executables (binaries). [CDRP-STD-0007:014]

Rationale: All code will be recompiled by the CDR Program to evaluate its completeness and portability. In addition, compiled code cannot be evaluated for security compliance and will be discarded. The separation of source code from compiled code can be accomplished by using the "export" command in version control systems such as CVS and Subversion, or by implementing a "make clean" rule in a Makefile.

Standard: Source code packages delivered to the CDR Program shall not contain any directories or files created by the version control system. [CDRP-STD-0007:015]

Rationale: Version control systems typically create directories and files in the developer's source area. For example, CVS creates a CVS directory, and Subversion creates a hidden .svn directory. Such directories and files can cause problems when imported into the CDR Program version control system, unnecessarily increase the size of the package, and provide information not needed by the CDR Program. The code to be delivered should be extracted cleanly from the version control system using the "export" command or its equivalent.

4. Coding Process

Although this document is focused on standards for the code as written, there are aspects of the coding process that have a direct impact on the quality of the finished code. Standards and guidelines for the coding process are presented in this section.

4.1 Optimization and Portability

Standard: All performance optimizations that violate the other standards and guidelines in this document shall be documented at the point where they are being made, with comments that focus on (1) why the optimization is needed; and (2) how it works. [CDRP-STD-0007:016]

Rationale: Prevents removal of the optimization as a result of code review or maintenance activities.

Guideline: Any code that is platform-specific should be refactored into a separate routine for each target platform. [CDRP-STD-0007:017]

Rationale: Portability. Clearly isolates the platform-specific code and provides a well-defined extension point for porting the code to a new platform.

4.2 Duplicated Code

Guideline: Functionality that exists in two or more modules, software units, or source files should be evaluated for refactoring as a separate element. [CDRP-STD-0007:018]

Rationale: Avoids the duplication of unit test cases and the duplication of code changes, should they be needed.

4.3 Include File Organization

Guideline: Include files should be organized hierarchically or in groups according to scope and content. [CDRP-STD-0007:019]

Example 1. Suggested Include File Groupings

- Application-wide parameters.
- Parameters specific to a single set of programs.
- Parameters specific to a single program or library.
- Symbolic error and function return values.
- Instrument/device parameters.
- Physical constants.
- Structure, union, and type definitions.

4.4 Self-Modifying Code

Standard: The application shall not use any executable code created or modified at run-time. [CDRP-STD-0007:020]

Note: This does not preclude the use of an auto-coder that is executed as part of the build process, i.e., at compile time.

Rationale: Self-modifying code is extremely difficult to debug, and adds unnecessary complexity to error detection and exception handling.

5. Naming Conventions

The intent of this section is to promote names that are self-documenting, follow regular patterns, and are easy to read.

5.1 General Naming Conventions

Standard: The names of modules, source files, routines, variables, and other software elements shall not exceed 31 characters in length; including any file name extension (suffix). This standard does NOT apply to input and output files. [CDRP-STD-0007:021]

Rationale: Some compilers permit longer names but only consider the first 31 characters when comparing names. See also [MISRA 2008], Rule 5.1.

Standard: The names of software elements shall correspond to the names specified in any related documentation, including, but not limited to the Climate Algorithm Theoretical Basis Document (C-ATBD) and the Operational Algorithm Description (OAD). [CDRP-STD-0007:022]

Rationale: Documentation is essential for maintainability and extensibility, but loses value if not synchronized with the source code.

Guideline: The names of software elements should reflect the application domain. [CDRP-STD-0007:023]

Rationale: The code should be comprehensible to a scientist and reflect the documentation of the algorithm in scientific papers and elsewhere.

Guideline: A consistent system of units should be used wherever possible throughout the application, and defined using comments that clearly identify units and conversions in file headers, variable declarations, interface specifications, design documentation, and user documentation. [CDRP-STD-0007:024]

Rationale: Errors resulting from inconsistent units can be easily missed and could easily result in the incorrect determination of a measurement or trend.

Guideline: Multi-word names of software elements should use camel case, underscores, or hyphens to separate each word. [CDRP-STD-0007:025]

Rationale: Greatly includes readability of multi-word element names. Hyphenation can only be used in file names.

Guideline: The use of capitalization, underscores, and hyphenation in software element names should be consistent throughout the application. [CDRP-STD-0007:026]

Rationale: Uniformity supports rapid comprehension.

Guideline: The names of software elements should not be abbreviated to the extent that the meaning is lost or they no longer resemble an English word. [CDRP-STD-0007:027]

Note: Loop counters and array indices such as i, j, k may be appropriate, provided that their use matches the formulas expressed in the algorithm documentation or common usage in mathematics.

Rationale: Longer English names are more easily understood and have a lower probability of duplicating other names within the application or operating system. For file names there is no need to be limited to the 8.3 standard imposed by MS-DOS.

Guideline: Acronyms and abbreviations should already be well-accepted in the application domain. [CDRP-STD-0007:028]

Rationale: The use of new or unfamiliar acronyms and abbreviations obscures meaning, inhibits rapid comprehension, and requires additional comments to define the acronyms and abbreviations.

Recommendation: The names of software elements should be long enough for self-documentation and short enough that they do not obscure the visual structure of the code. [CDRP-STD-0007:029]

Rationale: Readability. For most names the optimum length is between 8 and 16 characters. See [McConnell 2004] page 262.

Recommendation: Avoid using variable and file names that differ only by characters that look alike. [CDRP-STD-0007:030]

Rationale: Avoid confusion. Pairs of characters that appear similar in commonly used fixed width fonts include 0 and O, 1 and l (lowercase L), 1 and I (uppercase i), 2 and Z, and 5 and S.

5.2 Date and Time Representation

Guideline: Date and time strings in file names, log files, and other human readable outputs shall be rendered in a format compliant with the ISO Standard "Data elements and interchange formats -- Information interchange -- Representation of dates and times" (ISO-8601), except as may be required by an existing Interface Control Document (ICD) or similar specification. [CDRP-STD-0007:031]

Rationale: Alphanumeric sorting of a list of strings containing ISO 8601 dates and times yields a list that is time-ordered. Use of ISO 8601 removes the ambiguity between the American month/day representation and the British day/month representation and eliminates the Y2K problem. In addition, a consistent ordering of year, month, day, etc., will reduce the proliferation of date-time conversion routines. Compliant formats include calendar dates as YYYY-MM-DD and UTC dates and times as YYYY-MM-DDTHH:MM:SS.SSZ.

5.3 File Naming Conventions for Source Code

Standard: File names for source code shall be constructed only from upper and lower case alphabetic characters, numeric characters, underscores, hyphens, and periods. [CDRP-STD-0007:032]

Rationale: Simplifies code that parses file names (such as code counters), and supports interoperability with spreadsheets and other tools.

Standard: The name of the source code file containing the application entry point shall be clearly identified by including the string "main" in the file name. [CDRP-STD-0007:033]

Rationale: Rapid comprehension aids maintainability.

Standard: File name extensions (suffixes) for source code shall follow the standards defined in compiler documentation. [CDRP-STD-0007:034]

Rationale: Nonstandard extensions often require workarounds, particularly in Make files.

Standard: Files in any source code directory shall have names that differ from other files within that directory by more than alphabetic case. [CDRP-STD-0007:035]

Rationale: For example, myFile.f and MyFile.f appear to be the same file on some operating systems. This type of name collision unnecessarily limits the choice of operating systems that can be used for code development and maintenance.

Guideline: File names for source code should reflect the functionality implemented within. [CDRP-STD-0007:036]

Rationale: Rapid comprehension aids maintainability.

5.4 Naming Conventions for Constants

Guideline: Symbolic constants should be clearly identifiable as such. [CDRP-STD-0007:037]

Rationale: Clearly separates constants from variables. Symbolic constants include those defined with language dependent keywords such as #define, const, and PARAMETER, and also include any variable used to hold a constant value.

Guideline: Symbolic constants should name the entity that the constant represents, not the number. [CDRP-STD-0007:038]

Rationale: There is no value in having a constant named THREE that has the value 3, but it might be appropriate to have a constant NDIM that refers to the number of spatial dimensions.

Guideline: Symbolic constants should be used in place of hardcoded literals for all of the following cases: [CDRP-STD-0007:039]

a. Geophysical, geometric, and mathematical constants.

b. Fixed array dimensions, when these dimensions are used for more than one array.

c. Dimensions and offsets associated with input or output data.

d. Constant loop limits, where these limits are used by more than one loop.

Rationale: Self-documenting code is easier to understand and maintain.

Guideline: Symbolic constants should be defined in a single location within the application source code. [CDRP-STD-0007:040]

Rationale: Prevents the possibility of having different values of a constant in different parts of the application. Also reduces the cost and risk of maintenance by eliminating the need to search the entire source code in the event that a constant needs to be modified.

Guideline: Enumerated types or symbolic constants should be used for return codes, quality flags, error flags, and any other type of flag. [CDRP-STD-0007:041]

Rationale: Self-documenting code.

6. Source Code File Organization

6.1 Structure

Standard: Every file containing source code shall begin with a header comment section. [CDRP-STD-0007:042]

Note: See header section below for contents.

6.2 Header Content

Standard: Source code file header information shall be designated with the following keywords or their synonyms: [CDRP-STD-0007:043]

a. NAME: The name of the source code file.

b. PURPOSE: One or two sentences describing the source code file function.

c. DESCRIPTION: A description of the processing performed within this source code file. For published algorithms, provide a reference to the publication (see references below) rather than duplicating that information here. Any changes and high level implementation details should be noted. For unpublished algorithms that are best represented by complex diagrams, these diagrams should appear in the design documentation submitted with the source code, and that documentation should be referenced below.

d. AUTHOR(S): A list of those who wrote the code in the file, and their organization name. This list can be easily kept up to date if each person that works on the code adds his or name.

e. COPYRIGHT: Insert the following statement exactly as written, except for the initial comment character, which should be appropriate to the language.

```
! COPYRIGHT
!   THIS SOFTWARE AND ITS DOCUMENTATION ARE CONSIDERED TO BE IN THE PUBLIC
!   DOMAIN AND THUS ARE AVAILABLE FOR UNRESTRICTED PUBLIC USE.  THEY ARE
!   FURNISHED "AS IS."  THE AUTHORS, THE UNITED STATES GOVERNMENT, ITS
!   INSTRUMENTALITIES, OFFICERS, EMPLOYEES, AND AGENTS MAKE NO WARRANTY,
!   EXPRESS OR IMPLIED, AS TO THE USEFULNESS OF THE SOFTWARE AND
!   DOCUMENTATION FOR ANY PURPOSE.  THEY ASSUME NO RESPONSIBILITY (1) FOR
!   THE USE OF THE SOFTWARE AND DOCUMENTATION; OR (2) TO PROVIDE TECHNICAL
!   SUPPORT TO USERS.
!
```

f. REVISION HISTORY: The revision history of the file in forward chronological order, beginning with the initial version. This section should be appended with a new entry each time that a revised version of the software is submitted to the CDR Program and more often if appropriate. At a minimum changes to algorithms, interfaces, and outputs should be documented. For each such revision the new entry should provide version identification (at a minimum the revision date), the developer's initials, a brief summary of the changes made, and the reason for the changes.

Note: It is not required to update the history every time that the file is checked into local version control, although it is a best practice to always add a check-in comment in the version control system being used. Such check-in comments can be used to update the header revision history during preparation for delivery.

Guideline: Additional source code file header items listed below should be included as appropriate, to the extent that this information is not obvious from reading the code and its associated comments: [CDRP-STD-0007:044]

a. FILES: Input and output.

b. EXTERNALS: Routines and variables defined external to this source code file.

c. SUBROUTINES, FUNCTIONS, and/or PUBLIC METHODS: Any externally visible subroutines, functions, and/or methods contained in this file.

d. REFERENCES: Reference(s) to any published documents and engineering documents that this code is responding to, such as C-ATBD, OAD, requirements document, design document, standards, and algorithm changes.

e. USAGE: What the program is using (e.g., a calling sequence).

   Note: For programs run from the command line it is preferable to put effort into run-time help rather than documenting the command line in the file-header. Run-time help is more valuable to the user and avoids the need to update this section of the header as the program evolves.

f. ERROR CODES/EXCEPTIONS: Description of the overall approach to error reporting and exception handling in this source code file. Error codes should be documented here if not adequately documented at the point they are defined.

g. COMPILER NOTES: Description of any special compiler flags needed, or limitations on which flags cannot be used, especially regarding the level of compiler optimization.

h. NOTES: Any other information needed to increase understanding of this source code file.

7. Routines

7.1 Size and Complexity

Recommendation: Routine lengths should not exceed 100 lines. [CDRP-STD-0007:045]

Rationale: Each routine should be easily understandable. It is much harder to understand a routine that spans multiple pages. Excessively long routines are often a sign of poorly structured code [Holzmann 2006, Rule 4]. See also Section 7.4 of [McConnell 2004].

Recommendation: The McCabe Cyclomatic Complexity of any routine should be no greater than 15. [CDRP-STD-0007:046]

Rationale: The cyclomatic complexity of a routine is the number of unit test cases needed to execute all control paths within that routine. See [NIST 1996].

7.2 Header

Guideline: All routines should be preceded by a consistently formatted comment block that includes the following elements: [CDRP-STD-0007:047]

a. The name of the routine.
b. A brief description of the routine's purpose.
c. A list of the inputs with a description of each, including the physical units where applicable.
d. A list of the outputs with a description of each, including the physical units where applicable.
e. Any additional notes that will aid an understanding of how this routine works (optional).

Rationale: Understandability and reusability.

7.3 Entry

Standard: Application components that can be executed from the command line shall provide online help that shows command line usage. [CDRP-STD-0007:048]

Rationale: Usability.

Guideline: A routine should have a single point of entry. [CDRP-STD-0007:049]

Rationale: As with GOTO, multiple entries reduce cohesion and can lead to spaghetti code.

Guideline: Parameter lists should be ordered in the following sequence: input parameters, parameters used for both input and output, output parameters. [CDRP-STD-0007:050]

Rationale: Consistency aids comprehension.

Guideline: Input parameters should not be modified within a routine, unless the parameter is passed by value and is also not a pointer. [CDRP-STD-0007:051]

Rationale: Maintainability and portability. Input parameters should never be modified in Fortran because some compilers pass all parameters by reference.

Recommendation: A routine should have no more than seven parameters. [CDRP-STD-0007:052]

Rationale: Limitations of human working memory. Sustained comprehension aids maintainability.

7.4 Exit

Guideline: Routines should have a single exit point. [CDRP-STD-0007:053]

Rationale: Multiple exits inhibit rapid comprehension and reduce maintainability. Most routines should return a value or an error code, and having multiple exits requires additional rework if these should change.

8. Variables

8.1 Variable Type and Size

Guideline: All variables should be explicitly declared and typed, to the extent that the language supports such declarations. [CDRP-STD-0007:054]

Rationale: Explicit declaration prevents a misspelled variable name being treated as a new variable by the compiler/interpreter, and also allows complete freedom in choosing self-documenting variable names. Can be achieved by using "IMPLICIT NONE" in Fortran, "use strict" in Perl, and by using the compiler options specified in Table 1 to catch bad declarations.

Guideline: Variable sizes should be declared for those languages where the size is not explicitly defined by the language standard. [CDRP-STD-0007:055]

Rationale: Portability. The size of numerical types is typically not well defined and may vary from platform to platform.

Example 2. Defining Integer Types with Specific Sizes in C

The file stdint.h appears on many systems, including Linux, and may be used to typedef integers of standard size in a portable way:

```
#include <stdint.h>

int8_t var1;
uint8_t var2;
uint16_t var3;
uint32_t var4;
/* And so on */
```

For more details see http://en.wikipedia.org/wiki/Stdint.h

Example 3. Defining Floating Point Types with Specific Sizes in C

The following example from [MISRA 2008] shows recommended type definitions for floating point types on a machine with 32-bit floats. A similar list could be made for 64-bit machines. These could be placed in a header file with conditional compilation according to the machine type.

```
typedef      float  float32_t;
typedef      double float64_t;
typedef long double float128_t;

These typedefs are then used in declarations, for example:
float64_t wavelength;
```

8.2 Use of Memory

Recommendation: Unions (in C and C++), EQUIVALENCE statements (Fortran), and their equivalents in other languages should only be used when there is no alternative. [CDRP-STD-0007:056]

Note: Examples of uses that may be permissible include system calls, device drivers, and when required by a library interface.

Rationale: Using different identifiers for the same memory locations introduces coupling that inhibits comprehension.

Guideline: The success of any dynamic memory allocation should be checked by the code. [CDRP-STD-0007:057]

Rationale: Problems resulting from undetected failure of dynamic memory allocation can be very difficult to troubleshoot.

Example 4. Fortran Example of Memory Allocation With Test

```
ALLOCATE(x(M,N), STAT = alloc_stat)
IF (STAT .eq. 0) THEN
    ! Success
ELSE
    ! Failure
ENDIF
```

8.3 Variable Scope

Guideline: Variables should have the lowest scope consistent with their use. [CDRP-STD-0007:058]

Note: This precludes most uses of global variables.

Rationale: “Scope” refers to the visibility of a variable within the application. Variables with global scope are visible everywhere and allow the coupling of otherwise unconnected parts of the application. Highly coupled applications are more difficult to maintain, extend, and reuse. Modern languages allow a variable’s scope to be restricted to the routine level.

Guideline: A variable in an inner scope should not have the same name as a variable in an outer scope, and therefore hide that variable. [CDRP-STD-0007:059]

Rationale: [MISRA 2008], Rule 5.2.

8.4 Variable Initialization

Standard: All variables shall be initialized to a known value before use. [CDRP-STD-0007:060]

Rationale: Developers often assume that variables will be initialized by the compiler, operating system, computer hardware, input file, or by operator action. Such assumptions can be incorrect, particularly when the code is moved to a new platform or when other changes are made. The resulting errors are often difficult to reproduce.

8.5 Arithmetic

Standard: Shift operations shall not be used to perform integer multiplication and division. [CDRP-STD-0007:061]

Rationale: Portability and maintainability. Sign extension may not be performed or may be performed differently on different platforms. Although languages such as IDL and Java provide a uniform shift behavior, most other languages do not. It is difficult for a maintainer to remember the behavior of different platforms when they have to work in several languages.

Standard: Individual bits and bytes in floating point numbers shall not be used or modified. [CDRP-STD-0007:062]

Rationale: Portability and maintainability. The storage layout of floating point values may vary from one compiler to another. In addition the floating point implementation may not be fully compliant with IEEE Standard for floating point arithmetic (IEEE 754).

9. Code Layout

Code layout can be a controversial topic. The intent of this section is to promote layouts that effectively communicate the structure and content of the code, while allowing flexibility on the part of the developer.

9.1 Maximum Line Length

Standard: Source code lines shall not exceed 132 characters in length, including any indentation, but not including the line termination character(s). [CDRP-STD-0007:063]

Rationale: Some compilers will not accept lines longer than 132 characters.

9.2 Declarations

Recommendation: Each variable or other declaration should be placed on its own line. [CDRP-STD-0007:064]

Rationale: It is easier to find a variable visually or by "grep" when each declaration has a line of its own. In addition this layout simplifies the removal of unused variables and the addition of new variables, thus reducing the cost of maintenance and future refactoring. These benefits outweigh the extra space.

9.3 Statements and Expressions

Recommendation: Source code should be written with no more than one statement per line. [CDRP-STD-0007:065]

Rationale: Putting multiple statements on one line inhibits readability. See [McConnell 2004] pp.758-9 for a detailed discussion of this topic.

Guideline: Binary and ternary operators should be surrounded by spaces. [CDRP-STD-0007:066]

Rationale: Enhances readability.

Recommendation: Parentheses should be used to specify the order of evaluation for any expression that has more than one type of operator. [CDRP-STD-0007:067]

Rationale: Parentheses clarify intent regardless of the precedence rules defined for any specific language.

9.4 Blocking and White Space

Guideline: At least one blank line should appear between the end of one routine and the start of the next. [CDRP-STD-0007:068]

Rationale: Provides visual separation.

Guideline: Related statements should be grouped in blocks. [CDRP-STD-0007:069]

Rationale: Similar to the use of paragraphs in English.

Guideline: Blocks should be separated by blank lines. [CDRP-STD-0007:070]

Rationale: Appropriate use of white space adds significantly to the readability of code.

Recommendation: The number of blank lines should be 8 to 16 percent of the total lines. [CDRP-STD-0007:071]

Rationale: Studies have shown that the range 8 to 16 percent is optimal for effective debugging. See [McConnell 2004] section 31.2

9.5 Indentation and Nesting

Guideline: The body of each control structure should be indented by one level relative to the line containing the control logic. [CDRP-STD-0007:072]

Note: Control structures include routines, if-else statements, loops, switch statements, and the case labels under a switch statement.

Rationale: A good visual layout communicates the logical structure of the code.

Guideline: With the exception of make files, all source code files should use spaces rather than tab characters for indentation. [CDRP-STD-0007:073]

Rationale: Tabs are not treated the same way by all editors. Thus tab-indented code created by one developer may not be consistently indented when viewed by another developer. Editors intended for programming can be configured to emit a specified number of spaces when the tab key is pressed. However, “make” programs typically require the use of tab character for indentation.

Guideline: Indents should be at least two spaces and no more than four spaces. [CDRP-STD-0007:074]

Rationale: Studies have shown that indents of two to four spaces provide optimum readability. See [McConnell 2004] Section 31.2.

Guideline: The indentation scheme should be consistent throughout the application [CDRP-STD-0007:075].

Guideline: Each new level of nesting should have an additional level of indentation. [CDRP-STD-0007:076]

Guideline: The indentation scheme should visually distinguish control structures from continuation lines from labels. [CDRP-STD-0007:077]

Guideline: Nesting should not exceed five levels. [CDRP-STD-0007:078]

Rationale: Comprehensibility is reduced when human short-term memory becomes oversubscribed. In addition, deep nesting conflicts with self-documenting names, which tend to be longer.

Guideline: Comments should be indented to conform to the indentation of the code. [CDRP-STD-0007:079]

Recommendation: For levels that span 12 lines or more, the terminating symbol or keyword of each level in a nested control structure should contain an in-line comment explaining which level is terminated. [CDRP-STD-0007:080]

9.6 Control Flow

9.6.1 Unconditional Branch (GOTO)

Guideline: GOTO should not be used anywhere in the application. [CDRP-STD-0007:081]

Rationale: The use of GOTO encourages the rapid development of unstructured “spaghetti” code that is impossible to maintain or extend. All of the languages permitted by this standard offer control structures sufficiently rich that GOTO is never necessary. See also [Holzmann 2006], Rule 1.

9.6.2 Conditional branch (if – else)

Guideline: A control structure having only one statement in the body shall be coded with the body statement placed on a new line at the next level of indentation. [CDRP-STD-0007:082]

Rationale: Consistent with the indentation standards elsewhere in this document. Also facilitates use of a debugger, which may not be able to distinguish the control logic from the body when both are on the same line.

Guideline: A control structure having only one statement in the body shall be coded with block begin and end delimiters surrounding the body statement, for all languages that provide such delimiters. [CDRP-STD-0007:083]

Note: In C, C++, Java, and Perl the block begin and end delimiters are '{', and '}', in Fortran they are "THEN" and "ENDIF".

Rationale: Prevents the problem where additional correctly indented body statements are added but the programmer neglects to add the block begin and end delimiters.

Recommendation: An if … else if … construct should be terminated with an else clause. [CDRP-STD-0007:084]

Rationale: Defensive programming that ensures complete coverage of the conditions tested. Analogous to the switch statement final clause standard below. [MISRA 2008], Rule 14.10. See also the following recommendation.

Recommendation: The terminating else clause in an if … else if … else construct should generate an error message, warning message or assertion if this clause appears to be unreachable. [CDRP-STD-0007:085]

Rationale: Defensive programming to guard against errors in the control logic. Analogous to the switch statement final clause standard below. [MISRA 2008], Rule 14.10.

9.6.3 Switch and Case Statements

The syntax and behavior of switch and case statements varies from language to language. This section assumes an initial switch statement followed by a series of case labels.

Guideline: An unconditional break statement should terminate every non-empty case clause, i.e., no fall-through to the next label. [CDRP-STD-0007:086]

Rationale: [MISRA 2008], Rule 15.2. Inconsistent logic between successive case clauses makes the code difficult to understand. If the same non-trivial code appears in two clauses it should be factored out into a separate routine.

Recommendation: The final clause of a switch statement should be the default clause. [CDRP-STD-0007:087]

Rationale: Defensive programming. Ensures that the complete range of the switch condition is covered. [MISRA 2008], Rule 15.3.

9.6.4 Loops

Standard: Loops shall be entered only at the top. [CDRP-STD-0007:088]

Rationale: As with GOTO, this creates "spaghetti" code that is difficult to modify.

Standard: All loops shall have an exit condition that is certain to be reached, i.e., no infinite loops that are waiting for an external event such as operator input. [CDRP-STD-0007:089]

Rationale: The CDR code will be run in a highly automated environment. Care must be taken to ensure that there no inadvertent infinite loops. See also [Holzmann 2006], Rule 2.

Guideline: Floating point variables should not be used to count the number of iterations in a loop. [CDRP-STD-0007:090]

Rationale: Floating point representations of integers are not always exact and thus the number of iterations may differ from that expected.

10. Comments

This section contains commenting standards that augment the specific uses of comments that appear elsewhere in this document.

10.1 Comments in General

Standard: Comments shall not repeat information that is obvious in reading the code. [CDRP-STD-0007:091]

Rationale: Unnecessary duplication and requires double maintenance if the code changes.

Standard: Comments shall be used to justify any violations of standards. [CDRP-STD-0007:092]

Rationale: Protects a special case from being undone as a result of code review or maintenance.

Standard: Comments shall be used to document all data types, objects, and exceptions unless their names are self-explanatory. [CDRP-STD-0007:093]

Rationale: Essential for understandability and overcomes limitations of self-documentation.

Standard: Comments shall be concise, complete, and unambiguous. [CDRP-STD-0007:094]

Standard: Comments shall have correct spelling. [CDRP-STD-0007:095]

Rationale: The code will be made available to the public. Spelling errors reflect badly on the CDR Program and those who contribute to it.

Standard: Comments shall have correct grammar, either as full sentences in a paragraph format, or as sentence fragments in a bullet format. [CDRP-STD-0007:096]

Rationale: The code will be made available to the public. Grammatical errors reflect badly on the CDR Program and those who contribute to it.

10.2 Comments Reflecting Code Structure

Standard: Comments shall be used as needed to emphasize the structure of the code. [CDRP-STD-0007:097]

Guideline: Blocked comments should be used to highlight divisions between different sections of the code. [CDRP-STD-0007:098]

Example 5. Effective Commenting

The following examples in Fortran and C show a block comment as well as a concise comment that explains the next line.

```
! *****************************
! * Fortran
! * Process 16-bit float data
! *****************************
if (data_type .eq. DFNT_INT16) then
    ! *** Get the dimension scales into the array
    iflag = get_dim_scales(dim, size, rscale)
    ...


/***************************
* C
* Process 16-bit float data
****************************/
if (dataType == DFNT_INT16)
{
    /* Get the dimension scales into the array */
    iflag = GetDimScales(dim, size, rscale);
    ...
```

11. Operating System Interactions

11.1 Cohesion

Guideline: Operating system interfaces (such as file I/O) should be isolated and minimized. [CDRP-STD-0007:099]

Rationale: Supports maintainability and performance. For example, it may be difficult to comprehend an application where a file is opened at startup with a global identifier (such as logical unit number, file descriptor, or stream) and then read or written to at odd times by apparently unrelated routines. In addition, there is often a performance penalty associated with performing many small I/O operations instead of a few large operations.

11.2 Input and Output (I/O)

Standard: The source code shall not contain any hardcoded absolute paths to files or directories. [CDRP-STD-0007:100]

Rationale: Eliminates the need to modify and recompile the source code every time an I/O path is changed, and thus supports moving the compiled application from a development environment to a test or production environment. I/O paths should be passed to the application at run time via the command line or a configuration file. However, it is acceptable to hardcode the relative paths within a directory tree that has a configurable root.

Standard: The status of all file opens, reads, and writes shall be examined to determine whether an error occurred. [CDRP-STD-0007:101]

Rationale: The application output will almost always be incorrect if there is an I/O error during processing.

Guideline: Appropriate action should be taken in the event of an I/O error. [CDRP-STD-0007:102]

Note: Most such errors should be treated as non-recoverable, but in some situations it may be reasonable to repeat the operation or work around the failure.

Rationale: An incorrect result will almost always occur if there is an I/O error during processing. However, a retry strategy may be appropriate for operations on slow networked devices.

Guideline: Non-recoverable I/O errors should be treated in accordance with the exception and error handling standards defined elsewhere in this document. [CDRP-STD-0007:103]

Rationale: A systematic approach to error handling greatly aids the testing and debugging of large scale processing systems.

11.3 Remote File Transfer

Standard: The source code shall not perform any data transfers to or from remote sites, e.g., via ftp or scp. [CDRP-STD-0007:104]

Rationale: Security. The Readme file should contain sufficient information to obtain input files not included with the source code.

11.4 File Deletion

Standard: The command "rm –rf" shall not be used anywhere in the application. [CDRP-STD-0007:105]

Rationale: Security. It is all too easy to issue this command with an invalid path, or with a path that is simply '/', in which case this command will recursively delete every file owned by the user and every file to which the user has write permission.

11.5 Child Processes

Recommendation: Avoid use of the system() function and its analogs at the application level. [CDRP-STD-0007:106]

Rationale: Modern languages and their associated libraries provide features that can substitute for system() in many cases. Exception handling and error reporting from child processes is often difficult to accomplish in a systematic and well-controlled manner, making testing and debugging more difficult. If the number of spawned processes becomes large, it is possible to exceed the total number allowed on the system.

Standard: The exit status (return code) of every child process shall be examined to determine whether an error occurred. [CDRP-STD-0007:107]

Rationale: If the child process is necessary then it must work correctly. Even the simplest and most reliable child process will fail if its inputs are incorrect or other assumptions have been violated.

Example 6. Testing Exit Status on Unix-like Operating Systems

The following examples demonstrate exit status tests but do not show how to capture the child's stderr stream.

```
/*****************************************************************/
/* C Example */
status = system("cat /nothing");
if (status != 0) {
    /* Error handling code here */
}
else {
    /* Continue with normal processing */
}
```

```
!*******************************************************************
! Fortran 90/95 example
character(len=256) :: command
integer :: status
command = ‘cat /nothing’
call system (command, status)
if (status.ne.0) then
    ! Error handling code here
else
    ! Continue with normal processing
endif


####################################################################
# Perl example of error handling for system()
$status = system(“cat /nothing”);
if ($status != 0) {
    # Error handling code here
}
else {
    # Continue with normal processing here
}

####################################################################
# Perl example of error handling for captured child output
$text = `ls –l`
if ($@ != 0) {
    # Error handling code here
}
else {
    # Continue with normal processing here
}


####################################################################
# Python example of error handling for system()
import os
status = os.system(“ls –l”)
if status != 0:
    # Error handling code here
else:
    # Continue with normal processing here


####################################################################
# Python example of error handling for captured child output
# Note that popen() is deprecated in favor of the subprocess module
import os
f = os.popen(“ls –l”)
text = f.read()
status = f.close()
if status != 0:
    # Error handling code here
else:
    # Continue with normal processing here
```

Guideline: Appropriate action should be taken in the event of an error in a child process. [CDRP-STD-0007:108]

Note: Most such errors should be treated as non-recoverable, but in some situations it may be reasonable to repeat the operation or work around the failure.

Rationale: An incorrect result will almost always occur if an error occurs in a child process.

Guideline: Non-recoverable errors in child processes should be handled in accordance with the error handling standards defined elsewhere in this document. [CDRP-STD-0007:109]

Rationale: A systematic approach to error handling greatly aids the testing and debugging of large scale processing systems.

11.6 Use of stdout and stderr

Standard: The stdout and stderr streams shall not be merged. [CDRP-STD-0007:110]

Rationale: The stdout stream is buffered, while the stderr stream is not. If these streams are merged then messages sent to stderr may be embedded at random places in the standard output, and may not be correctly time-ordered relative to stdout.

Guideline: If the stdout stream is used as input to another program (e.g., via an intermediate file or pipe) then all logging, warning, and error messages shall be handled as specified by architecture and design documentation, or sent to stderr if no such documentation exists. [CDRP-STD-0007:111]

Rationale: The corruption resulting from embedded stderr messages is particularly problematic when the stdout stream is used as input by another process.

12. Exceptions and Error Handling

Guideline: Errors should be handled as close as possible to the point that they are detected. [CDRP-STD-0007:112]

Rationale: Resources are wasted by continuing execution when an error condition is known to exist.

Guideline: Each routine should implement exception and error handling consistent with the overall approach defined in the architecture or design documentation for the application. [CDRP-STD-0007:113]

Rationale: A consistent approach to exception and error handling throughout the application is necessary for effective troubleshooting and debugging.

Guideline: Exceptions should be used only to communicate abnormal or unexpected conditions. [CDRP-STD-0007:114]

Rationale: Exception handling mechanisms are much slower than typical calls because of the stack unwinding involved. Ideally, exceptions should occur vary rarely during normal operations. Examples of exceptions that may be encountered in numerical data processing include divide by 0 errors, file permission errors, and array out of bounds errors, all of which can be avoided by defensive programming.

Guideline: In the event of a "fatal" error condition being detected that would prevent further processing, or which would render the output unusable, the application should: [CDRP-STD-0007:115]

a. Provide an appropriate error logging message; and

b. Terminate with a non-zero exit status.

Rationale: Compliance with this guideline allows a higher level script or automation framework to determine that an abnormal termination has occurred and take appropriate action. Otherwise, processing may continue indefinitely, other unrelated processes may be affected, and a large amount of troubleshooting and manual cleanup may be needed, all of which constitute a waste of resources.

Recommendation: Each routine should be designed and implemented to detect and report all foreseeable failures, including those that "should never happen". [CDRP-STD-0007:116]

Rationale: Comprehensive error detection is essential for robust data processing, testability, and maintainability. In the long run it is cheaper to build this in at the start than to add it later.

Guideline: Each routine calling another should check the error status information returned to it before proceeding further, if such status is available from the routine being called. [CDRP-STD-0007:117]

Rationale: If the routine is necessary then it must work correctly. Even the most reliable routine will fail if its inputs are incorrect or other assumptions have been violated. See also [Holzmann 2006], Rule 7 and [MISRA 2008], Rule 16.10.

Guideline: The error reporting mechanism should report: (1) the name of the routine where the error was detected; and (2) an intuitively clear statement of the error that is both unambiguous and unencumbered by technical jargon. [CDRP-STD-0007:118]

Rationale: Clarity in this area greatly assists testing and troubleshooting.

Guideline: In the case of a system error (including I/O errors) the error reporting mechanism should also capture and report the system error message, if the content of that message would aid comprehension. [CDRP-STD-0007:119]

Rationale: Additional clarity that assists testing and troubleshooting.

Appendix A. Acronyms and Abbreviations

Acronym or Abbreviation	Definition
ANSI	American National Standards Institute
C-ATBD	Climate Algorithm Theoretical Basis Document
CDR	Climate Data Record
FOC	Full Operating Capability
IEEE	Institute of Electrical and Electronic Engineers
ICD	Interface Control Document
IOC	Initial Operating Capability
MISRA	Motor Industry Software Reliability Association
NCDC	National Climatic Data Center
NOAA	National Oceanic and Atmospheric Administration
OAD	Operational Algorithm Description

Appendix B. Glossary

Term	Definition
Big-endian	A byte ordering of a multi-byte word in which the most significant byte is stored in the lowest address of the memory space occupied by the word.
Binary Operator	An operator having two operands. Examples include the arithmetic operations "+", "-", "*", and "/".
Enumerated Type	A data type consisting of a set of named values.
Exception	A special condition that changes the normal flow of program execution. For example, a division by zero.
Executable	A file containing machine code that can be immediately loaded into memory and run by the operating system.
Hardcoded	A value defined in source code.
Library	A module that implements functionality useful in a range of applications, is specifically designed to be reused without modification, and is packaged and delivered separately from any specific application.
Little-endian	A byte ordering of a multi-byte word in which the most significant byte is stored in the highest address of the memory space occupied by the word.
Maintainability	The ease with which the software may be understood, modified, and tested, in order to add or change functionality, improve performance, or correct defects.
Module	An implementation unit of software that provides a coherent unit of functionality [SEI 2011]. For the purposes of this document a module consists of one or more source code files, i.e., individual routines are not considered to be modules. Software applications are typically decomposed into several modules during high level design. Modules may be defined at different levels of decomposition, i.e., a high-level module may be constructed from lower level modules. The lowest level modules are often called "software units". Coherence implies that a module can be tested independently of the application, although testing may require a test harness to substitute for the interfaces and data normally provided by the remainder of the application.
Physical Line	A non-blank, non-comment line of code. Each continuation line counts as an additional physical line.
Platform	A combination of specific hardware and a specific operating system.
Portability	The ease with which the software may be modified to operate in an environment different to that for which it was specifically designed. Complete portability implies that no modification is needed.
Readability	The ease with which the source code can be read and understood at the detailed statement level.
Robustness	The degree to which the software continues to operate in the presence of invalid inputs or stressful environmental conditions.

Routine	A sequence of executable statements with intervening comments and white space that is invoked ("called") from an executable statement, and which returns control to the calling statement upon completion. Depending on the programming language and type of routine, a routine may return data to the caller or modify the input data provided by the caller. This generic definition includes all "functions", "subroutines", "methods", "program units", and "main programs" as they may be defined in various programming languages.
Scope	The locations in an application's source code where a variable, routine, or other named software element is accessible to the code at that location.
Source Code File	Any file containing code that will be compiled or interpreted to machine readable instructions. This definition includes scripts and so-called "include" or "header" files that are inserted into other files during compilation or interpretation.
Software Unit	The smallest element of an application that is testable as an independent entity. May consist of one or more source files. Often synonymous with "module".
Ternary Operator	An operator having three operands. The most common is the "?" operator in C, Java, and other languages.

Appendix C. Further Reading

In addition to the references in Section 1, the following sources were examined during development of this document:

Defense System Software Development DOD-STD-2167A, Appendix B, Department of Defense, 1988. Found at http://www.everyspec.com/DoD/DoD-STD/download.php?spec=DOD-STD-2167A.008470.pdf

GNU Coding Standards, Free Software Foundation, 2011, Stallman, Richard, et al., 2011. Found at http://www.gnu.org/prep/standards/

Kroah-Hartman, Greg, Documentation/Coding Style and beyond (presentation), 2002. Found at http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/index.html

Motor Industry Software Reliability Association, MISRA C++:2008, Guidelines for the use of the C++ language in critical systems, 2008.

SDST-096 MODIS Science Software Delivery Guide, Rev.C, 2004. http://modis-sdst.gsfc.nasa.gov/documents/SDST_096_RevC_Final_092804.doc

Software Architecture Glossary, Software Engineering Institute, Carnegie Mellon, 2011. http://www.sei.cmu.edu/architecture/start/glossary/

General Programming Principles and Guidelines, Version 1.0, NOAA Satellite Products and Services Review Board (SPRSB), 2009. http://projects.osd.noaa.gov/spsrb/standards_docs/General_Prog_Standards_June2009.pdf

Fortran77 Programming Standards, Version 1.0, NOAA Satellite Products and Services Review Board (SPRSB), 2009, Kenneth A. Jensen. http://projects.osd.noaa.gov/spsrb/standards_docs/Fortran77_Prog_Standards_and_Guidelines.pdf

Standards, Guidelines and Recommendations for Writing Fortran 95 Code, Version 1.0. NOAA Satellite Products and Services Review Board (SPRSB), 2009, S.-A. Boukabara and P. Van Delst. http://projects.osd.noaa.gov/spsrb/standards_docs/Fortran95_standard_rev22Jun2009.pdf

General Programming Principles and Guidelines, Version 2.0, NOAA Satellite Products and Services Review Board (SPRSB), 2010, (Approval Pending). http://projects.osd.noaa.gov/spsrb/standards_docs/general_standards_v2.0.docx

Standards, Guidelines and Recommendations for Writing Fortran 90/95 Code, Version 2.0, NOAA Satellite Products and Services Review Board (SPRSB), 2010, (Approval Pending), S.-A. Boukabara and P. Van Delst. http://projects.osd.noaa.gov/spsrb/standards_docs/fortran95_v2.0.docx

Standards, Guidelines and Recommendations for Writing C Code, Version 1.0, NOAA Satellite Products and Services Review Board (SPRSB), 2010, (Approval Pending), S.-A. Boukabara and P. Van Delst. http://projects.osd.noaa.gov/spsrb/standards_docs/Ccode_v1.0.docx

TD 11.2: C Programming Standards and Guidelines, Version 3.0, NOAA NESDIS Center for Satellite Applications and Research (STAR), 2007, Alward Siyyid et al. http://www.star.nesdis.noaa.gov/star/documents/PAL/Version3/TrainingDocuments/STAR_TD-11.2.0_v3r0.pdf

TD 11.1: Fortran Programming Standards and Guidelines, Version 3.0, NOAA NESDIS Center for Satellite Applications and Research (STAR), 2009, Ken Jensen and Alward Siyyid, 2009. http://www.star.nesdis.noaa.gov/star/documents/PAL/Version3/TrainingDocuments/STAR_TD-11.1.0_v3r0.pdf

TD 11.1A: Transition From Fortran 77 to Fortran 90, Version 3.0, NOAA NESDIS Center for Satellite Applications and Research (STAR), 2009, Ken Jensen. http://www.star.nesdis.noaa.gov/star/documents/PAL/Version3/TrainingDocuments/STAR_TD-11.1.0_v3r0.pdf

Torvalds, Linus, Linux Kernel Coding Standards, v2.6.37, 2007. http://lxr.linux.no/#linux+v2.6.37/Documentation/CodingStyle

Software Development Guidelines, University of California at Riverside, 2000. http://www.literateprogramming.com/sdg.pdf

LAND PEATE VIIRS Science Data Processing Software Systems Description, Version 1.2, Revision B, 2007. http://modis-sdst.gsfc.nasa.gov/documents/VIIRS_Science_Data_Processing_Software.pdfCDRHeading 2