

**Clouds and the Earth's Radiant Energy System
(CERES)**

Data Management System

**CERES Software Coding Guidelines
Release 1**

Scott Quier, Joseph Stassi, and Troy Anselmo

Science Applications International Corporation (SAIC)
One Enterprise Parkway
Hampton, Virginia 23666

November 1994

Preface

The Clouds and the Earth's Radiant Energy System (CERES) Data Management System supports the data processing needs of the CERES Science Team research to increase understanding of the Earth's climate and radiant environment. The CERES Data Management Team works with the CERES Science Team to develop the software necessary to support the science algorithms. This software, being developed to operate at the Langley Distributed Active Archive Center, produces an extensive set of science data products.

The Data Management System consists of 12 subsystems; each subsystem represents a stand-alone executable program. Each subsystem executes when all of its required input data sets are available and produces one or more archival science products.

The documentation for each subsystem describes the software design at various significant milestones and includes items such as Software Requirements Documents, Data Products Catalogs, Software Design Documents, Software Test Plans, and User's Guides.

TABLE OF CONTENTS

	<u>Page</u>
Preface.....	iii
1.0 Introduction	1
2.0 Design Guidelines	2
2.1 Languages and Operating Systems	2
2.2 Modules	2
2.2.1 Module Naming	2
2.2.2 Module Partitioning	3
2.2.3 Module Interfaces	3
3.0 Files	4
4.0 Comments	5
4.1 General Commenting Guidelines	5
4.2 Prologue Comments	6
4.3 Data Comments	7
4.4 Statement Comments	7
4.5 Marker Comments	7
5.0 Code Presentation	8
5.1 General Code Presentation	8
6.0 Data	9
6.1 Naming	9
6.2 Constants	9
6.3 Global Data	9
7.0 Miscellaneous	10
8.0 Programming Constructs	11
8.1 Naming	11
8.2 Statements	11
8.3 Looping and Branching	11
9.0 Functions and Subroutines	12
10.0 Error Handling	13
11.0 Portability	14
12.0 Numerical Methods	15
13.0 Fortran 77 Coding Guidelines	16
13.1 General Style Rules	16
13.2 Common Blocks	16
13.3 Labels	16
13.4 Constants	17

TABLE OF CONTENTS (cont'd)

	<u>Page</u>
13.5 Arrays	17
13.6 Loop Constructs	17
14.0 Fortran 90 Coding Guidelines	18
14.1 General Style Rules	18
14.2 Obsolete Fortran Features	19
14.3 Guidance for the use of Dynamic Memory	19
15.0 C Coding Guidelines	21
15.1 General Style Rules	21
15.1.1 Include Files.....	21
15.1.2 Data Declarations.....	21
15.1.3 Functions: Declarations, Prototypes, Calling.....	22
15.1.4 Flow Control, Loops, and Braces	22
15.1.5 Switch	23
15.1.6 IF-ELSE-IF Constructs	24
15.1.7 Operators.....	24
15.1.8 Portability.....	25
16.0 Ada Coding Guideline	29
References.....	30

Appendices

Appendix A	Terms, Acronyms, and Abbreviations	A-1
Appendix B	Module Prologue	B-1

1.0 Introduction

The purpose of this document is to provide a set of guidelines for source code development for the Clouds and the Earth's Radiant Energy System (CERES) ground data processing system. It originates from the need to standardize software development practices within the CERES Data Management Team. This document primarily provides guidelines for the format and presentation style of software source code. However, it also provides several design guidelines which are necessary to set the context for understanding the coding guidelines. These guidelines should be distributed to developers at the beginning of the project and used as a measurement instrument in design and code reviews.

The editors of this document recognize that there is a broad range of opinion on design and coding standards, and that in many cases the benefits of flexibility in adapting standards to a particular situation outweigh the advantages of adhering to a rigid set of standards. However, the value of establishing guidelines for software engineering is clearly demonstrated by the problems related to program accuracy, maintainability, and portability which are encountered by projects where standards have not been used to a sufficient degree.

These guidelines in this document are intended to be followed by all CERES subsystems. However, it is recognized that cases will arise where one must deviate from these guidelines in the interest of performance, reusability, or other mitigating circumstance. To avoid establishing a bureaucracy for granting waivers, the code inspections will be used as the vehicle for approving deviations from these guidelines. The rationale is that if the author must convince a reasonable group of peers that a deviation is warranted, an adequate level of control on compliance to these guidelines will be achieved.

Please note that the ideas presented here have been extracted from numerous references to avoid "reinventing the wheel" in all cases. The documents which provided much of the content found in this document are listed in "References," located at the end of the text portion of this document.

Finally, throughout this document, terms are used which may be ambiguous or not entirely clear. Appendix A contains a list of Terms, Acronyms, and Abbreviations and definitions that describe them.

2.0 Design Guidelines

2.1 Languages and Operating Systems

1. Software will be implemented in American National Standard Institute (ANSI) compliant Ada, C, Fortran 77, or Fortran 90. Language extensions may be used if approved by the project sponsor. If C++ is added to the list of Earth Observing System Data Information System (EOSDIS)-approved languages, it may subsequently receive approval for use in CERES. Guidelines specific to C++ have not been included in this document.
2. The source code interface to the operating system should be limited to the call interface specified by the Portable Operating System Interface for Computer Environments (POSIX) standard.
3. All code written for delivery to the EOSDIS Planning and Data Processing System (PDPS) shall use the PDPS Toolkit to the extent required by the Data Production Software and Science Computing Facility (SCF) Standards and Guidelines (EOSDIS document 423-16-01). This includes, but is not limited to, system and resource accesses, error message transaction, and metadata formatting. Use of PDPS Toolkit resources beyond these minimal requirements is strongly encouraged.
4. Inherited code which does not conform to the guidelines will be considered for porting on a case-by-case basis by the appropriate subsystem lead.

2.2 Modules

In this section, the term "module" is used to represent a named procedure, subroutine, or function.

2.2.1 Module Naming

1. Module names should correspond to the names used in program structure charts. Module names should describe what the routine does.
2. Names for utility modules should be prefixed by the name (or abbreviation) of the utility set to which they belong:
 - a) Prefixes must be unique.
 - b) Prefixes which might be confused with the naming conventions of common commercial libraries must be avoided (for example, routines beginning with "db" might be confused with Sybase DB-Library routines).
 - c) Separate the prefix from the rest of the module name with an underscore.

2.2.2 Module Partitioning

1. A block of code which appears in more than two places should be considered for packaging as a separate module.
2. Blocks of code which perform nearly the same function should be considered for packaging in a single, general-purpose module.
3. A module whose length exceeds 100 lines of code should be looked at closely for partitioning into smaller modules.

2.2.3 Module Interfaces

1. Isolate and minimize external program interfaces (such as file I/O) as much as possible.
2. Minimize the number of formal parameters. A long list of formal parameters is suspect as an indication of a module that is attempting to accomplish too much. However, do not group unrelated data elements into a single structure for the sole purpose of reducing the number of parameters.
3. Avoid passing data into a module that the module does not need to perform its task.
4. As a general rule, be careful when passing variables or using global variables to control the internal logic of a module. Passing a parameter into a module which controls the internal logic of that module can be an indication of poor coupling between modules.
5. Reduce the "distance" (i.e., the number of times data passes across a call interface) between where a variable is initialized and where it is used.
6. Include modules should be organized hierarchically according to scope and content. Examples of item categories include the following:
 - a) System-wide parameters
 - b) Parameters specific to a single program set
 - c) Parameters specific to a single program
 - d) Symbolic error and function return values
 - e) Instrument/device parameters
 - f) Physical constants
 - g) Structure, union and type definitions

3.0 Files

1. Every source module (both executable source and include modules) must begin with a prologue. Appendix B provides an annotated prologue template.
2. It is recommended that each file contain one and only one source code module except in cases where this places undue limits on the programmer. An example of when it is appropriate to place more than one module in a file would be where the modules in question form a cohesive and self-contained whole.
3. Files should be named according to the following conventions:

File Type	Name Template (replace italics with relevant information)
Fortran 77 module	<i>module_name.f</i>
Fortran 90 module	<i>module_name.f90</i>
Ada package specification	<i>module_name_.a</i>
Ada package body	<i>module_name.a</i>
C module	<i>module_name.c</i>
Include files	<i>module_name.h</i>
Bourne Shell scripts	<i>module_name.ch</i>
C Shell scripts	<i>module_name.csh</i>

4.0 Comments

Comments can be either beneficial or harmful to software maintainers. They can be beneficial by explaining aspects of the code that are otherwise not readily apparent. They can be harmful by containing inaccurate information or by being too numerous or not visually distinct enough, which can cause them to obscure the structure of the code. As a general rule, one should strive to write code which is clear and unambiguous without comments. However, we recognize that this goal is not always attainable, and comments can provide information that is useful for understanding and maintaining the code. This section provides guidelines for writing effective comments.

4.1 General Commenting Guidelines

1. Make the code as clear as possible to reduce the need for comments.
2. Misspelled, ungrammatical, ambiguous, or incomplete comments defeat their usefulness. If a comment is worth adding, it is worth adding correctly to increase its usefulness.
3. Never repeat information in a comment which is readily available in the code.
4. Use comments to emphasize the structure of the code and to draw attention to deliberate and necessary violations of standards.
5. Where a comment is required, make it concise and complete.
6. Avoid lengthy explanations within the body of the code. If long explanations are necessary, place them within the prologue of the module.
7. Use proper grammar and spelling in comments.
8. Indent comments to conform to the indentation of the code so as not to obscure the code's structure or readability.
9. Use asterisks "****" to make comments visually distinct from the code.
10. Use blocked comments to highlight divisions between different sections of the code. The following code gives an example of items (7) through (9) above.

```
c      ****
c      * Process 16-bit float data *
c      ****
c      if (data_type .eq. DFNT_INT16) then
c
c          *** Get the dimension scales into the array
c          iflag = get_dim_scales(dim, size, rscale)
c          if (iflag .ne. 0) then
c              write(*, 1050) dim
c              goto 9999
c          end if
c      end if
```

The C language example of the above concepts might look like:

```

/*****
* Process 16-bit float data
*****/
if (dataType == DFNT_INT16)
{
    /* Get the dimension scales into the array */
    iflag = GetDimScales(dim, size, rscale);
    if (iflag != 0)
    {
        printf(stderr, "ERROR.... %d\n", dim);
    } /* end of if */
} /* end of if */

```

4.2 Prologue Comments

1. Per the requirements set forth in the Data Production Software and Science Computing Facility (SCF) Standards and Guidelines, each module will contain a prologue (see Appendix B for language specific examples). Each module prologue will contain, as a minimum, the following information:
 - a) Module name
 - b) Language
 - c) Subsystem/module code number
 - d) Module type (e.g. include file, source code, package spec, package body)
 - e) Purpose
 - f) Input parameters
 - g) Output parameters
 - h) Revision history appearing in chronological order to contain the date, name and organization of the person making the change, summary of each change, version number, configuration management authorization information (SCCR #), and source for contributed or reused code.
2. The following items should be included in the module prologue as appropriate:
 - a) Explanation of how and why the module does what it does
 - b) Summary of complex algorithms
 - c) Reasons for significant or controversial implementation decisions
 - d) Names of other modules called by the module along with a one line description of what the called module does
 - e) Names of all global, local, and constant (PARAMETER and #define) data symbols referenced or set in the module. This will include a short description of the purpose of the data symbol. These should appear in the same sequence as that found in the code module (see Section 3.3)
 - f) Explanation of nonportable features within the module
 - g) Failure modes

3. The following items should not be included in the prologue comments:
 - a) Information about how the unit fits into the enclosing software system
 - b) Names of modules which call the module

4.3 Data Comments

1. Comment on all data types, objects, and exceptions unless their names are self-explanatory.
2. Include information on the semantic structure of complex pointer-based data structures that are maintained between data objects.
3. For C, the order should be `struct`, `double`, `float`, `long`, `short`, `char`, and `void`. For FORTRAN, the order should be `complex`, `double precision`, `real`, `integer`, `logical`, and `char`.

4.4 Statement Comments

1. Minimize comments embedded among statements
2. Use comments only to explain parts of the code that are not obvious
3. Comment intentional omissions from the code
4. Do not use comments to paraphrase the code
5. Do not use comments to explain remote pieces of code, such as subprograms called by the current module
6. Where comments are necessary, make them visually distinct from the code

4.5 Marker Comments

For long or heavily nested if and case statements, mark the *else* and *end* of the statement with a comment summarizing the condition governing the statement.

5.0 Code Presentation

5.1 General Code Presentation

1. Make generous use of vertical and horizontal white space. Indentation and spacing reflect the block structure of the code. Appropriate use of white space and variable names adds significantly to the readability of code.
2. In all cases, use spaces as opposed to tabs for indenting. Tab characters are displayed differently by different terminals and printers.
3. Indent and align nested control structures, continuation lines, and embedded functions/subroutines consistently. Use an indentation scheme which visually distinguishes nested structures and continuation lines from labels.
4. Use spacing in equations to reveal operators and reinforce precedence. Use blanks to make code more readable.
5. Use parentheses to specify the order of subexpression evaluation for complex logical or arithmetic expressions.
6. Write no more than one simple statement per line.
7. Do not exceed 80 characters per line of code. This will enhance readability on standard-sized printouts and on older terminals.
8. Avoid clever code constructs which may be difficult for maintenance programmers to understand.
9. Eliminate unreachable code or code that is never called.
10. Always capitalize the E in scientific notation. Use upper case for the alphabetic characters representing digits in bases above 10.
11. Use of mixed case characters for reserved words will follow language conventions. Be consistent in the use of upper and lower case.
12. Write code for clarity rather than speed. Where optimization is required to meet performance requirements, a justification and explanation of how the optimization has been done should be included in the optimized code. Comments should include a discussion of all design factors which lead to the particular optimization strategy.

6.0 Data

6.1 Naming

1. Variable names should be composed of English words (and possibly numeric characters) or easily recognizable abbreviations thereof (except as indicated below).
2. Choose names that are as self-documenting as possible. Use names found in the problem domain but not obscure jargon.
3. Choose identifiers that describe the object's value during execution. Use singular, specific nouns as variable identifiers.
4. Use a consistent abbreviation strategy. Do not use an abbreviation of a long word as an identifier where a shorter synonym exists. Variable names may contain acronyms only where their meaning is readily understood by persons other than the code writer.
5. Use predicate clauses (e.g. IsOpen) or adjectives for boolean objects.

6.2 Constants

1. The names of symbolic constants (C #defines, Fortran PARAMETERS) follow the rules for variable names, except that symbolic constants should be in all capital letters.
2. Use symbolic values instead of literals where appropriate. Use of a symbolic constant for PI (3.124....) defined for the entire CERES DMS is appropriate where one for a loop limit may not be.
3. The following types of constants should be defined symbolically in a separately included file:
 - a) Physical and geometric constants
 - b) Threshold values
 - c) Dimension of program arrays
 - d) Dimensions and offsets associated with input or output data
 - e) Loop limits
 - f) Function return values
 - g) Error and other types of flag values (error codes must be integers)

6.3 Global Data

Global variables should be avoided since their presence increases coupling. If a global variable *must* be used, the design factors leading to its use should appear in the code as comments. In addition, it should be set only in the module in which it is defined.

7.0 Miscellaneous

1. All variables shall be initialized to a known value before being referenced. Do not assume that any variable will be initialized by the operating system, language, computer hardware, or by any other agency.
2. Avoid the use of unions (C language) and EQUIVALENCE statements (Fortran)
3. Ensure that all variables, symbols, and labels are referenced within the program.
4. Do not rely on the internal representation of data objects. Examples are:
 - a) Signed Integers - not all implementations are sign extended. Do not substitute shift operations for multiplication and division.
 - b) Floating Point - not all implementations are IEEE (bit 31:sign, bits 30-23:exponent, bits 22-0: fraction). Do not modify individual bits to modify a floating point value.
 - c) Alignment - not all implementations require integers to be long-word aligned (starting on a 32-bit boundary). At least one implementation allows integers to be aligned on 16-bit boundary. Avoid extracting information from memory outside of language provided operations.
 - d) Padding - between any two variables in a list there may or may not be areas of user/programmer inaccessible memory (padding). Avoid extracting information from memory outside of language-provided operations.
 - e) Enumerated types - the first item of an enumerated type may or may not start at zero and may or may not be defined in terms of a primitive data type (e.g. integer). Testing the value of an enumerated type against other than a member of the type will produce undefined behavior.

8.0 Programming Constructs

8.1 Naming

1. Use a consistent abbreviation strategy. Use abbreviations that are well accepted in the application domain.
2. Use action verbs for procedures. Use predicate-clauses (e.g. Is_Open) for boolean functions. Use nouns for nonboolean functions.

8.2 Statements

1. Do not make multiple assignments on a single line of source code.
2. Do not nest expressions or control structures beyond a nesting level of five.
3. The terminating symbol or keyword in nested statements longer than ~12 lines should contain an in-line comment explaining which nesting level is being terminated.
4. Choose names of flags so they represent states that can be used in positive form. Use:

```
if (Operator_Missing) then
```

rather than

```
if (not Operator_Found) then
```

5. Separate control logic and computation as much as possible.

8.3 Looping and Branching

1. All loops shall have a testable condition for exit.
2. Keep if...then...else constructs as short as possible with the most frequently executed branch coming first.
3. Avoid use of unconditional branching (GOTO).
4. All loop control variables will be of type integer.
5. Loops shall only be entered from the top. Do not branch into the middle of a loop.

9.0 Functions and Subroutines

1. A routine should have only one entry point.
2. Minimize the number of returns from a subprogram.
3. Variable names that cross a module interface (i.e., that appear as parameters to another routine) should be the same as the parameter names used in program structure charts.
4. Highlight returns with comments or white space to keep them from being lost in other code.
5. Formal parameters should be listed in the Input, Both, Output order where it is reasonable to do so.
6. Identify all formal parameters as Input, Both, or Output through use of comments whenever this identification is not clear through language syntax.
7. In modules which do not have default parameters, define parameters in the sequence: Input, Both, Output (see C example below).

```
void geolocate_Pixel(  
    float          coefficient,      /* inputs */  
    pixelStructure thePixel,        /* in/out */  
    int            errorCode);      /* output */
```

8. Minimize the use of parameters which are both Input and Output.
9. Do not change the value of Input parameters.

10.0 Error Handling

Errors should be reported from the module that both detects and understands the nature of the error. Error messages should be as informative as possible. For example, printing a message like "Can't open file," is not nearly as helpful to the user as "Parameter file 'program.parms' not in current path." With the right strategy, error reporting can also make a program much easier to maintain. If the previous example was modified to read, "Error in module parameter_fetch: Parameter file 'program.parms' not in current path," a maintenance programmer would know exactly where the problem occurred.

EOSDIS will maintain a process log containing all messages generated by a program. In situations where data processing is done with little or no human interaction, such processing logs may be the only way to trace problems back to when they first appeared. Here, the more information stored in the log, the better.

1. Failure modes of each module should be documented in the prologue and then as appropriate within the body of the code.
2. Each module should return error status information to the module that called it.
3. Each module calling another should check the error status information returned to it before proceeding further.
4. Modules should be designed and implemented to detect and easily handle all foreseeable failures.
5. The error reporting mechanism should report: (1) the name of the routine where the error was detected; (2) an intuitively clear statement of the error that is both unambiguous and unencumbered by technical jargon; (3) suggested remedies.

11.0 Portability

The advantages of portable code are well known. As the expected life span of the CERES DMS is very long, issues of portability need to be of particular concern during design and implementation. This section gives some general guidelines for writing portable code. Additional guidelines are included in the three language-specific sections to follow.

Within this document, "portable" means that a source file can be compiled and executed on different machines with only minor (if any) changes being made. In general, a "new machine" may be different hardware, a different operating system, a different compiler, or any combination of these. The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

1. Write portable code first. Optimized code is often obscure and may produce code of lesser quality on another machine. Document performance optimizations, localize them as much as possible, and explain *how* it works and *why* it was needed.
2. Recognize that some things are inherently nonportable. Examples are code to deal with particular hardware registers, such as the program status word, and code that is designed to support only a particular piece of hardware.
3. Isolate machine-dependent code into separate modules as much as possible. This will ease the task of porting the code to a new machine. Comment the machine dependence in the module prologue.
4. Any behavior that is described as "implementation defined" (this would include authorized use of compiler extensions) should be treated as a machine (compiler) dependency. Assume that the compiler or hardware does it in a completely unexpected way.
5. Whenever possible, explicitly declare variable size. Do not depend on system default sizes, which often vary from machine-to-machine.
6. If language features allow it, and where it is appropriate, define precision and/or range for floating point numbers and range for integer values.
7. On some machines, the first half of a double precision may be a single precision with similar value. Do *not* depend on this.
8. Code that takes advantage of two's complement representation of numbers on most machines should not be used. Optimizations that replace arithmetic operations with equivalent shifting operations are particularly suspect.
9. The bytes of a word are of increasing significance with increasing address on machines such as the VAX (little-endian) and of decreasing significance with increasing address on other machines (big-endian). The order of bytes in a word and of words in larger objects (say a double word) might not be the same. Hence, any code that depends on the left-right orientation of bits in an object deserves special scrutiny. For this reason, it is decidedly non-portable to concatenate *any* two variables into a larger third variable.

12.0 Numerical Methods

1. Use `<=` and `>=` in relational expressions with real (floating point) operands instead of equality (e.g. `==` or `.eq.`).
2. Use explicit type conversion when using mixed mode arithmetic.
3. Do not rely on implicit rounding behavior when converting from one data type to another.

13.0 Fortran 77 Coding Guidelines

13.1 General Style Rules

1. Adhere to strict Fortran 77 as closely as possible, with the following exceptions:
 - a) Identifier names may be up to 31 characters.
 - b) INCLUDE statements may be used.
2. Avoid using EQUIVALENCE statement.
3. Never pass literal constants as arguments in calls to subprograms.
4. Explicitly declare all variables. Use IMPLICIT NONE to ensure declaration.
5. Do not abbreviate .TRUE. or .FALSE.
6. Do not use alternate returns.
7. Avoid the use of GOTO statements except as defined in Section 11.6 below.

13.2 Common Blocks

1. Avoid unnecessary use of COMMON blocks.
2. Declare COMMON blocks in INCLUDE files rather than multiple times in separate modules.
3. Use of data statement to initialize COMMON block variables should occur only in BLOCK DATA modules.
4. SAVE all COMMON blocks (if code gets upgraded to Fortran 90, standards do not require that COMMON block variables get "saved").
5. Do not group unrelated variables in COMMON blocks.
6. Do not mix CHARACTER and noncharacter types in a COMMON block
7. Use the following ordering of numeric types within COMMON blocks: double precision, real, integer.

13.3 Labels

1. Assign labels in ascending order.
2. Assign a separate sequence of labels to FORMAT statement labels and, group them at the end of the module.
3. Right-adjust labels on column five.

13.4 Constants

1. Use PARAMETERS to symbolically name all compile-time constants.
2. Use only constant expressions to define PARAMETERS.

13.5 Arrays

1. Declare array dimensions in the type declaration rather than in a separate DIMENSION statement.
2. Use only INTEGER subscript expressions.

13.6 Loop Constructs

1. Terminate or begin every loop with a distinct CONTINUE (see item (3) below).
2. Do not modify the loop control variable.
3. Do not send the loop control variable as a parameter to a subroutine.
4. The following loop constructs are permitted:

a) Iterative Loop

```
do 10 I = 1, iterations
  ...
10  continue
```

b) While Loop

```
10  CONTINUE
    ...
    IF (condition) GOTO 20
    ...
    GOTO 10
20  CONTINUE
```

c) Do-while or repeat-until (all statements in loop execute at least once)

```
10  CONTINUE
    ...
    IF (condition) GOTO 10
```

14.0 Fortran 90 Coding Guidelines

Section 14.1 contains recommended guidelines on how to best use Fortran 90 features. This list of do's and don'ts will likely be expanded/modified as we gain experience with the new standard.

Section 14.1 lists remnant Fortran 77 features which should no longer be used.

Section 14.2 contains coding guidelines for the use of dynamic memory.

14.1 General Style Rules

1. Use free format syntax.
2. Use IMPLICIT NONE in all program modules. In other words, explicitly declare all variables.
3. Use labels only for FORMAT statements or to jump to error handling code (see item 9a in Section 14.2 below).
4. Collect all FORMAT statements together at the bottom of the module.
5. Always name 'program modules' and always use the END PROGRAM; END SUBROUTINE; END INTERFACE; END MODULE; constructs, again specifying the name of the 'program module'.
6. Use >, >=, ==, <, <=, /= instead of .gt., .ge., .eq., .lt., .le., .ne. in logical comparisons. The new syntax, being closer to standard mathematics, should be clearer.
7. Use the following conventions for variable declarations:
 - a) Do not use the DIMENSION statement or attribute: declare the shape and size of arrays inside parentheses after the variable name on the declaration statement.
 - b) Declare the length of a character variable using the (len =) syntax.
8. Use the USE statement only to specify which of the variables, type definitions, and subprograms defined in a module are to be made available to the USEing routine.
9. Use INTERFACE blocks for all external f90 routines. This allows the compiler to check that the type, shape and number of arguments are correct.
10. Use array notation whenever possible. To improve readability, show the array's shape in parentheses, e.g.:

```
1dArrayA(:) = 1dArrayB(:) + 1dArrayC(:)
2dArray(:, :) = scalar * Another2dArray(:, :)
```
11. Use the INTENT attribute to specify dummy arguments in subprograms as IN, OUT, or INOUT.
12. A module shall refer only to its own subprograms and to those intrinsic routines included in the Fortran 90 standard.

14.2 Obsolete Fortran Features

The following features are considered obsolescent in Fortran 90, or are made redundant by other Fortran 90 features. They should not be used.

1. COMMON blocks - Use MODULEs instead
2. EQUIVALENCE - Use POINTERS or derived data types instead
3. Assigned and computed GO TOs - Use the CASE construct instead
4. Arithmetic IF statements - Use the block IF construct instead
5. Labelled DO constructs - Use END DO instead
6. I/O routine's (END =) and (ERR =) - Use IOSTAT instead
7. Alternate return - Use the CASE construct instead
8. H Editing - Use quotations instead
9. GO TO
 - a) The only recommended use of GO TO is to jump to the error handling section at the end of a routine on detection of an error. The jump must be to a CONTINUE statement, and the label used must be 9999.
 - b) Any other use of GO TO should be avoided by making use of IF, CASE, DO WHILE, EXIT, or CYCLE statements.
10. PAUSE

Use of the following features are discouraged on the basis that they are bad programming practice and can degrade the maintainability of code:

1. ENTRY statements - A subprogram may only have one entry point
2. FUNCTIONS with side effects - This is common practice in C programming, but can be confusing
 - a) Functions should not alter variables in their argument list or in modules used by the function
 - b) Functions should not perform I/O operations
3. Implicitly changing the shape of an array when passing it into a subroutine

14.3 Guidance for the use of Dynamic Memory

There are three ways of obtaining dynamic memory in Fortran 90: automatic arrays, pointer arrays, and allocatable arrays.

Automatic arrays: These are arrays which are initially declared within a subprogram whose extents depend upon variables known at runtime (e.g. variables passed into the subprogram via arguments).

Pointer arrays: Array variables declared with the POINTER attribute may be allocated space at run time by using the ALLOCATE command.

Allocatable arrays: Array variables declared with the ALLOCATABLE attribute may be allocated space at run time by using the ALLOCATE command. However, unlike pointers, allocatables are not allowed inside derived data types.

1. Use automatic arrays in preference to the other forms of dynamic memory allocation.
2. Space allocated using Pointer arrays or Allocatable arrays must be explicitly freed using the DEALLOCATE statement.
3. Always test the success of a dynamic memory allocation. The ALLOCATE statement has an optional argument to let you do this.

15.0 C Coding Guidelines

15.1 General Style Rules

15.1.1 Include Files

Include files are files that are included into a compilation unit prior to compilation by the C preprocessor. Some, such as `stdio.h`, are predefined by the computer system or are delivered as part of the compiler package and must be included by any program using the standard I/O library. Include files (both private - those defined by the programmer and system) are also used to contain data declarations and defines that are needed by more than one module or program.

1. Do not name private include file names the same as library include files.
2. Do not use absolute path names when specifying them for inclusion in program modules. Use, instead, the `"#include <name>"` convention for getting them from system standard locations, or use the compiler "include path" option for private include paths.
3. Include files should not be nested (one include file "including" another). In extreme cases, where a large number of include files are to be included in several different module files, it is acceptable to put all common `#include` statements in one include file.

15.1.2 Data Declarations

1. The "pointer" qualifier, `*`, will be with the variable name rather than with the type.

```
char *s, *t, *u;
```

instead of

```
char* s, t, u;
```

which is wrong, since `'t'` and `'u'` are not then declared as pointers.

2. Unrelated declarations, even of the same type, should be on separate lines.
3. Constants used to initialize or set the value of variables of type long (whether in executable code or in data declaration statements) will be explicitly long. Use capital letters, (2L) two long (2l) looks like (21), the number twenty-one.
4. For structure template declarations, each element of the structure should be alone on a line with an in-line comment describing it.
5. Structures may be "typedefed" when they are declared. Give the structure and typedef the same name.

```

typedef struct NickNamesT
{
    int nick_count;           /* Ord position in array */
    char *RealName;         /* The real name */
    char *NickName;         /* Person's favorite name */
}NickNamesT;

```

15.1.3 Functions: Declarations, Prototypes, Calling

1. The return type of functions should always be declared. If the function does not return a value, it will be declared to return type void.
2. If function prototypes are supported by the compiler, use them.
3. The types of all parameters in the function parameter list will be declared. Do not default them to type int.
4. Function return type, function name, and formal parameter list should all appear on a single line if feasible.
5. Do not code declarations that override declarations found at higher levels. The following is an example of the discouraged, though ANSI C legal, construct:

```

{
    int count;
    ...
    {
        int count;
        ...
    }
}

```

6. Avoid passing expressions as function call parameters. Instead, evaluate the expressions prior to the function call.
7. Where NULL is passed as an actual function parameter, cast it as a pointer to the appropriate type:

```
retVal = funct_call((int *)NULL);
```

15.1.4 Flow Control, Loops, and Braces

Flow control statements and loop structures include, but are not limited to, "if-else", "for", "while", "switch" and "do-while". The "best" brace style (the placement of braces within program code) has become a very personal point of contention among many C language programmers. When writing CERES DMS code, the choice of brace style is left to the individual Subsystem teams, with the following caveats:

1. Where a Subsystem has made a decision, all new code will be written to conform. The style will be constant.

2. Where heritage code (that which has been in the scientific community for a number of years) is introduced to the Subsystem, all modifications to that code will adhere to the de-facto brace style of that code.
3. When writing new code, or modifications to "heritage" code, the bodies of all flow control, loop and switch structures will be enclosed by braces, irrespective of the number of lines of code contained in such bodies. For examples:

```
for(count = 0; count < MAXSCAN; ++count)
{
    ...
}

while(count < MAXSCAN)
{
    ...
}

do
{
    ...
}while(Done == FALSE);

if(iscaps(c) == TRUE)
{
    ...
}
else
{
    ...
}
```

15.1.5 Switch

1. Avoid using "fall-through" program code in switch statements. Where it *absolutely* must be implemented, comment it as in the following example:

```
switch(expr)
{
case ABC:
case DEF:
    statement;
    break;

case UVW:
    statement;
    /* FALLTHROUGH */

case XYZ:
    statement;
    break;
```

```
    default:
        break;
}
```

2. The default case may not be strictly required and may not make sense. However, if there is even the slightest possibility that "expr" (from the above example) will not be an anticipated value, the default case will be coded.
3. The last case of a switch statement will terminate with a break statement. This may eliminate maintenance problems in the future.

15.1.6 IF-ELSE-IF Constructs

An IF-ELSE-IF construct should be written with the ELSE conditions left justified:

```
#define STREQ(s1, s2) (strcmp((s), (s2)) == 0)

if(STREQ(reply, "yes"))
{
    statements for yes
}
else if(STREQ(reply, "no"))
{
    statements for no
}
else if(STREQ(reply, "maybe"))
{
    statements for maybe
}
```

15.1.7 Operators

1. In keeping with the intent of statements in Section 4.1, most binary operators should be separated from their operands by blanks. In the C language, a couple of notable exceptions are the '->' and '.' operators, which should not be delimited by white space.
2. Expressions involving mixed operators (e.g. use of +, -, *, and / in a single expression) should be parenthesized as C language has some unexpected precedence rules. However, because humans do not match parentheses well, too many can make code much more difficult to read.
3. The binary comma (',') operator should generally be avoided except in, for example, the for statement, where it is very useful in implementing multiple initialization or incrementation operations.
4. Complex operations, such as those using nested ternary ('?:') operators, can be confusing and should be avoided in favor of multiple or nested if statements (which in many cases will compile to the same instructions).

5. The ‘++’ and ‘--’ operators can be used with operands of either type integer or any pointer type. They are particularly suited to the task of setting a pointer to the next (previous) element in an array.
6. Do not use the ‘++’ and ‘--’ operators in actual function or macro parameters, e.g.:

```
retVal = call_fn(x++, y++);
```

instead, code it as:

```
retVal = call_fn(x, y);
++x;
++y;
```

7. Avoid the use of the ‘++’ and ‘--’ operators in combination with other operators. Such may easily lead to mistakes as to the order in which the expression(s) are evaluated and the side effects that go with it (such as undefined behavior). Do not code the following:

```
y = i+ ++i;
```

instead (one possible interpretation is):

```
y = i;
++i;
Y += i;
```

15.1.8 Portability

This section describes portability issues that, in addition to those discussed in Section 9.0, need to be considered during design and implementation of the CERES DMS.

1. Pay attention to word sizes. Objects may be nonintuitive sizes. Pointers are not always the same size as *ints*, the same size as each other (e.g., *void ** may not be same size as *char ** or *int **), or freely interconvertible. Some machines have more than one possible size of a given type. The size you get can depend both on the compiler and on various compile-time flags.
2. The *void ** type is guaranteed to have enough bits of precision to hold a pointer of any data object. The *void(*)()* type is guaranteed to be able to hold a pointer to any function. Use these types when you need a generic pointer. (Use *char ** and *char(*)()*, respectively, with older compilers). Be sure to cast pointers back to the correct type before using them.
3. Even when, an *int ** and a *char ** are the same *size*, they may have different *formats*. For example, the following will fail on some machines that have `sizeof(int *) equal to sizeof(char *)`. *The code fails because free expects a char * and receives an int **.

```
int *p = (int *)malloc(sizeof(int) * 10);
if(p != NULL)
{
    free(p);
}
```

4. Note that the *size* of an object does not guarantee the precision of that object. The Cray-2 may use 64 bits to store an *int*, but a *long* cast into an *int* and back to a *long* may be truncated to 32 bits.
5. The integer *constant zero* may be cast to any pointer type. The resulting pointer is called a *null pointer* for that type, and is different from any other pointer of that type. A null pointer always compares equal to the constant zero. *A null pointer might not compare equal with a variable that has the value zero. Null pointers are not always stored with all bits reset to zero.* Null pointers for two different types are sometimes different. A null pointer of one type cast to a pointer of another type will be cast in to the null pointer for that second type.
6. On ANSI compilers, when two pointers of the same type access the same storage, they will compare as equal. When non-zero integer constants are cast to pointer types, they *may* become identical to other pointers. *On non-ANSI compilers, pointers that access the same storage may compare as different.* Two such pointers (e.g. $(int *)2$) and $(int *)3$), for instance, may or may not compare equal, and they may or may not access the same storage.
7. Avoid signed characters. For example, on some VAX machines characters are sign extended when used in expressions. This is not the case on many other machines. Code that assumes signed/unsigned is nonportable.
8. In general, if the word size or value range is important, typedef "sized" types. Large programs should have a central header file which supplies typedefs for commonly used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code. Unsigned types, other than *unsigned int*, are highly compiler-dependent. If a simple loop counter is being used where either 16 or 32 bits will do, then use *int*, since it will get the most efficient (natural) unit of the current machine.
9. Data *alignment* is also important. For instance, on various machines, a 4-byte integer may start at any address, start only at an even address, or start only at a multiple-of-four address. Thus, a particular structure may have its elements at different offsets on different machines, even when given elements are the same size on all machines. Indeed, a structure containing a 32-bit pointer and an 8-bit character may be three sizes on three different machines. As a corollary, pointers to objects may not be interchanged freely; saving an integer through a pointer to 4 bytes starting at an odd address will sometimes work (depending on the machine), sometimes cause a core dump, and sometimes fail silently (destroying other data in the process). Pointer-to-character is a particular trouble spot on machines which do not address to the byte. Alignment considerations and loader peculiarities make it very rash to assume that two consecutively declared variables are together in memory, or that a variable of one type is aligned appropriately to be used as another type.
10. Because there may be unused holes in structures, suspect unions used for "type cheating." Specifically, a value should not be stored as one type and retrieved as another.
11. Different compilers use different conventions for returning structures. This causes a problem when libraries return structure values to code compiled with a different compiler. Pointers to structure are not a problem.
12. Do not make assumptions about the parameter passing mechanism, especially pointer sizes and parameter evaluation order, size, etc. The following code, for instance is *very non-*

portable. This example has lots of problems. The stack may grow up or down (indeed, there need not even be a stack!). Parameters may be widened when they are passed, so a *char* might be passed as an *int*, for instance. Arguments may be pushed left-to-right, right-to-left, in arbitrary order, or passed in registers (not pushed at all). The order of evaluation may differ from the order in which they are pushed. One compiler may use several (incompatible) calling conventions for different situations.

```

{
    char c;
    ...
    c = foo(getchar(), getchar());
    ...
}

char foo(char c1, char c2)
{
    char bar = *(&c1 + 1);
    return(bar);
}

```

13. On some machines, the null character pointer ((char *)0) is treated the same way as a pointer to a null string. Do *not* depend on this.
14. Do not modify string constants. Two particularly notorious (bad) examples are:

```

char *s = "/dev/tty??";
strcpy(&s[8], ttychars);

```

or

```

char *s = "/dev/tty??";
strcat(s, ttychars);

```

use instead, something analogous to the following:

```

char *s = "/dev/tty??";
char thePath[MAXPATHLEN];
strcpy(thePath, s);
strcat(s, ttychars);

```

15. The address space may have holes. Simply computing the address of an unallocated element in an array (before or after the actual storage of the array) may cause the program to crash. If the address is used in a comparison, sometimes the program will run but destroy data, give wrong answers, or loop forever. In ANSI C, a pointer into an array of objects may legally point to the first element after the end of the array; this is usually safe in older implementations as well. This "outside" pointer may not be dereferenced.
16. Only the == and != comparisons are defined for all pointers of a given type. It is only portable to use <, <=, >=, or > to compare pointers when they both point in to (or to the first element of) the same array. It is, likewise, only portable to use arithmetic operators on pointers that both point into the same array or the first element afterwards.

17. Become familiar with existing library functions and defines. (But not *too* familiar. The internal details of library facilities, as opposed to their external interfaces, are subject to change without warning. They are also often quite nonportable.) You should not be writing your own string compare routine, terminal control routines, or making your own defines for system structures. "Rolling your own" is a fruitful source of bugs. If possible, be aware of the *differences* between the common libraries (such as ANSI, POSIX, and so on).
18. Use *lint* when it is available. If your compiler has switches to turn on warnings, use them.
19. Function parameters should be cast to the appropriate type. Always cast NULL when it appears in nonprototyped function calls.

16.0 Ada Coding Guideline

The document *Ada Quality and Style: Guidelines for Professional Programmers*, Version 2.01.01, December 1992 should be followed for all Ada source code. This document is published by the Software Productivity Consortium, Inc., and it can be obtained via anonymous ftp at [ajpo.sei.cmu.edu](ftp://ajpo.sei.cmu.edu/public/adastyle) in the directory `/public/adastyle`.

References

1. *JPL Software Implementation Guidelines* (JPL D-10622), Version 1.0, April 26, 1993
2. *JPL Software Management Standards Package* (JPL D-4000), Version 3.0, 1988
3. Software Productivity Consortium, Inc. *Ada Quality and Style: Guidelines for Professional Programmers*, Version 02.01.01, December 1992
4. *European Standards For Writing and Documenting Exchangeable Fortran 90 Code*, Draft Version 0.3, July 1994
5. *Fortran 77 Coding Guidelines*, David L. Levine, University of California, Irvine, CA, 11 December 1989
6. *ERBE Software Development Standards*, NASA Langley Research Center, Hampton, VA, September 1981
7. *Data Production Software and Science Computing Facility (SCF) Standards and Guidelines* (EOSDIS document 423-16-01), January 14, 1994

Appendix A
Terms, Acronyms, and Abbreviations

ANSI	American National Standards Institute
POSIX	Portable Operating System Interface for Computer Environments
SCF	Science Computing Facility

Terms, Acronyms, and Abbreviations

Throughout this document, several terms are used which may be ambiguous or not entirely clear. Such terms are listed here, in alphabetical order, that such confusion may be avoided.

Term	Definition
CERES	Clouds and the Earth's Radiant Energy System
DMS	Data Management System
EOS	Earth Observing System
EOSDIS	EOS Data and Information System
function	A code module that receives all input through the calling interface, performs defined operations based upon that input or data known only within that module, and returns one, and only one value - not through the calling interface.
module	A source code file containing the definition of a function or subroutine designed to accomplish one, and only one, task.
PDPS	Planning and Data Processing System (formerly called PGS)
PGS	Product Generation System (now called PDPS)
SCCR	Software Configuration Change Request
subroutine	A code module that receives none, some, or all of its input through the calling interface; performs defined operations based upon that input on data that may or may not be known only to the module, and returns zero or more values through the calling interface formal parameters.
subsystem	A coherent set of code (usually compiled into one executable program) which performs a major part of the functions of the DMS.
symbol	The "label" or string of characters used to represent either a module, variable, or constant.
variable	A symbol referencing a location in the program data address space.

Appendix B
Module Prologue

Module Prologue

```
01 /*****
02 !C
03
04 Name: char *foo(int Input1, int Input2)
05
06 !Description:
07 Module ID : 4.1.3.6
08 Module Type: C Module
09
10 Purpose:
11
12 This function will take the input values, allocate a buffer to
13 hold floating point value, perform black magic, load the
14 results of the black magic (one value at a time) into the buffer
15 and return the address of this buffer to the calling routine.
16
17 !Input Parameters:
18 int Input1 - This does something.
19 int Input2 - This does something else.
20
21 !Output Parameters:
22 Return value is the address of the black magic buffer.
23
24 !Revision History:
25
26 SCCR # N/A (Initial code write).
27 Revision 1.0 1994/07/27 01:01:24
28 Scott Quier (s.r.quier@larc.nasa.gov)
29 Initial delivery of software.
30
31 SCCR # 0001
32 Revision 1.1 1994/07/27 13:01:34
33 Scott Quier (s.r.quier@larc.nasa.gov)
34 Had to modify the code to comply with standards. No problem.
35
36 !Team-unique Header:
37
38 Discussion of Complex Algorithms:
39
40 Significant/Controversial Implementation Decisions:
41
42 Called Modules:
43
44 Global Variables:
45
46 Local Variables:
47
48 Constants:
49
50 Non-Portable Features/Language Extensions:
51
52 Error Handling:
53
54 !end
55 *****/
                                <code follows here>
```

- Line 02: Initial marker to take the following values:
- | | |
|----------|---------------------------------------|
| !FXY | FortranXY source code (XY = 77 or 90) |
| !FXY-INC | FortranXY include file |
| !C | C language source code |
| !C-INC | C include file |
| !ADA | Ada source code |
| !ADA-INC | Ada include file |
- Line 04: Name of the procedure or include file. If this is not the main procedure or an include file, it should contain the function statement. This serves two purposes. First, it shows the module's name and type. Secondly, it shows the input/output parameter names and types.
- Line 06: A concise but complete description of the module. This includes module ID, module type, and a summary of the purpose of the module. Any references for methods and/or algorithms should be included in the summary. Use as many lines as desired.
- Line 17: Header for formal input parameters
- Line 18 - 19: Formal input parameters in the order they appear in the module declaration statement. Parameters should appear with a short 1 or 2 line description and its units (if appropriate).
- Line 21: Header for formal output parameters and function return values.
- Line 22: Output parameters and function return values (not global variables) in the order they are contained in the module declaration.
- Line 24: Revision History header. If you are using an automated tool for revision control, you should insert any statements required immediately after this line.
- Line 26 - 35: Revision History. Each revision should contain a minimum of the revision number, date and time of revision, name and e-mail of person making revision.
- Line 36: Team-unique Header. Each team may design its own header section.
- Line 38: In addition to mentioning them in the description, complex algorithms should be discussed in some detail here. The area is separate from that above to highlight code that has been identified as difficult.
- Line 40: Where an algorithm has been implemented in a controversial (as opposed to non-portable) manner, discussion to this effect should appear here.
- Line 42: Other modules called from this module, listed in alphabetical order with a short (1 - 2 line) description of the called module's purpose.
- Line 44: List all global variables (those found in common blocks, etc.) either referenced or set. Include a short (1 - 2 line) description of the variable's use. For C, the order should be `struct`, `double`, `float`, `long`, `short`, `char`, and `void`. For FORTRAN, the order should be `complex`, `double precision`, `real`, `integer`, `logical`, and `char`.
- Line 46: List all local variables declared in the module. Listing order is the same as for globals. Include a short (1 - 2 line) description of the variable's use.

- Line 48: List all symbolic constants referenced in the module, in alphabetical order. Include a short (1 - 2 line) description of the variable's use.
- Line 50: Discuss the implementation of non-portable code or the use of language extensions.
- Line 52: Error Handling: Describe here any errors which may be returned or exceptions which may be raised by this module. Describe any error handling provided by the module and uses made of the PDPS Toolkit error handling services.
- Line 043: End of the source code prologue.