SOFTWARE ENGINEERING LABORATORY SERIES

SEL-94-003

C STYLE GUIDE

AUGUST 1994

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on the process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The major contributors to this document are

Jerry Doland (CSC)
Jon Valett (GSFC)

Many people in both the Software Engineering Branch at NASA/GSFC and in the Software Engineering Operation at CSC reviewed this document and contributed their experiences toward making it a useful tool for Flight Dynamics Division personnel.

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

ABSTRACT

This document discusses recommended practices and style for programmers using the C language in the Flight Dynamics Division environment. Guidelines are based on generally recommended software engineering techniques, industry resources, and local convention. The *Guide* offers preferred solutions to common C programming issues and illustrates through examples of C code.

C
Style Guide

FIGURES

1 INTRODUCTION

"Good programming style begins with the effective organization of code. By using a clear and consistent organization of the components of your programs, you make them more efficient, readable, and maintainable."

– Steve Oualline, *C Elements of Style*

1.1 Purpose

This document describes the Software Engineering Laboratory (SEL) recommended style for writing C programs, where code with "good style" is defined as that which is

- Organized
- Easy to read
- Easy to understand
- Maintainable
- Efficient

1.2 Audience

This document was written specifically for programmers in the SEL environment, although the majority of these standards are generally applicable to all environments. In the document, we assume that you have a working knowledge of C, and therefore we don't try to teach you how to program in C. Instead, we focus on pointing out good practices that will enhance the effectiveness of your C code.

1.3 Approach

This document provides guidelines for organizing the content of C programs, files, and functions. It discusses the structure and placement of variables, statements, and comments. The guidelines are intended to help you write code that can be easily read, understood, and maintained.

- Software engineering principles are discussed and illustrated.
- Key concepts are highlighted.
- Code examples are provided to illustrate good practices.

2 READABILITY AND MAINTAINABILITY

This section summarizes general principles that maximize the readability and maintainability of C code:

- Organize programs using encapsulation and information hiding techniques.
- Enhance readability through the use of white space.
- Add comments to help others understand your program.
- Create names that are meaningful and readable.
- Follow ANSI C standards, when available.

2.1 Encapsulation and Information Hiding

Encapsulation and information hiding techniques can help you write better organized and maintainable code. **Encapsulation** means grouping related elements. You can encapsulate on many levels:

- Organize a program into files, e.g., using header files to build a cohesive encapsulation of one idea.
- Organize files into data sections and function sections.
- Organize functions into logically related groups within individual files.
- Organize data into logical groups (data structures).

Information hiding refers to controlling the visibility (or **scope**) of program elements. You can use C constructs to control the scope of functions and data. For example:

- Encapsulate related information in header files, and then include those header files only where needed. For example, #include <time.h> would be inserted only in files whose functions manipulate time.
- A variable defined outside the current file is called an **external variable**. An external variable is only visible to a function when declared by the extern declaration, which may be used only as needed in individual functions.

Figure 1 illustrates the information hiding concept. The code consists of two files, three functions, and six variables. A variable name appears to the right of each line that is within its scope.

File	**Code**	**Scope**
x.c	#include “local.h”	
	int a = 2;	
	static int b = 3;	a
	main()	ab
	{	ab
	int c = a + b;	ab
		abc
	xsub(c);	abc
	}	abc
	xsub(d)	ab
	int d;	ab
	{	ab d
	int e = 7 * d;	ab d
		ab de
	ysub(e);	ab de
	}	ab de
y.c	#include “local.h”	
	ysub(f)	
	int f;	
	{	f
	extern int a;	
		a f
	printf(“%d\n”, a + f);	a f
	}	a f

Figure 1 Information Hiding

2.2 White Space

Write code that is as easy as possible to read and maintain (taking into consideration performance tradeoffs for real-time systems when it is appropriate). Adding white space in the form of blank lines, spaces, and indentation will significantly improve the readability of your code.

2.2.1 Blank Lines

A careful use of blank lines between code "paragraphs" can greatly enhance readability by making the logical structure of a sequence of lines more obvious. Using blank lines to create paragraphs in your code or comments can make your programs more understandable. The following example illustrates how the use of blank lines helps break up lines of text into meaningful chunks.

Example: code paragraphing

```
#define LOWER  0
#define UPPER  300
#define STEP   20

main()  /* Fahrenheit-Celsius table */
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr - 32));
}
```

However, overuse of blank lines can defeat the purpose of grouping and can actually reduce readability. Therefore, use a single blank line to separate parts of your program from one another.

2.2.2 Spacing

Appropriate spacing enhances the readability of lexical elements such as variables and operators. The following examples illustrate how to use individual spaces to improve readability and to avoid errors. The second example is not only harder to read, but the spacing introduces an error, where the operator /* will be interpreted by the compiler as the beginning of a comment. Put one space after a comma to improve readability, as shown in the third example below.

Example: good spacing

```
*average = *total / *count;   /* compute the average */
```

Example: poor spacing

```
*average=*total/*count;       /* compute the average */
               ^ begin comment  end comment^
```

Example: comma spacing

```
concat(s1, s2)
```

2.2.3 Indentation

Use indentation to show the logical structure of your code. Research has shown that **four spaces** is the optimum indent for readability and maintainability. However, in highly nested code with long variable names, four-space indentation may cause the lines of code to overrun the end of the line. Use four spaces unless other circumstances make it unworkable.

Example: four-space indentation

```
main()
{
    int c;

    c = getchar();
    while (c!= EOF)
    {
        putchar(c);
        c = getchar();
    }
}
```

2.3 Comments

Judiciously placed comments in the code can provide information that a person could not discern simply by reading the code. Comments can be added at many different levels.

- At the program level, you can include a **README** file that provides a general description of the program and explains its organization.
- At the file level, it is good practice to include a **file prolog** that explains the purpose of the file and provides other information (discussed in more detail in Section 4).
- At the function level, a comment can serve as a **function prolog**.
- Throughout the file, where data are being declared or defined, it is helpful to add comments to explain the purpose of the variables.

Comments can be written in several styles depending on their purpose and length. Use comments to **add information** for the reader or to **highlight sections** of code. Do not paraphrase the code or repeat information contained in the Program Design Language (PDL).

This section describes the use of comments and provides examples.

- Boxed comments—Use for prologs or as section separators
- Block comments—Use at the beginning of each major section of the code as a narrative description of that portion of the code.
- Short comments—Write on the same line as the code or data definition they describe.
- Inline comments—Write at the same level of indentation as the code they describe.

Example: boxed comment prolog

```
/*****************************************************
 * FILE NAME                                         *
 *                                                   *
 * PURPOSE                                           *
 *                                                   *
 *****************************************************/
```

Example: section separator

```
/*****************************************************/
```

Example: block comment

```
/*
 *  Write the comment text here, in complete sentences.
 *  Use block comments when there is more than one
 *  sentence.
 */
```

Example: short comments

```
double ieee_r[];          /* array of IEEE real*8 values    */
unsigned char ibm_r[]; /* string of IBM real*8 values    */
int count;                /* number of     real*8 values    */
```

- Tab comment over far enough to separate it from code statements.
- If more than one short comment appears in a block of code or data definition, start all of them at the same tab position and end all at the same position.

Example: inline comment

```
switch (ref_type)
{

    /* Perform case for either s/c position or velocity
     * vector request using the RSL routine c_calpvs */

    case 1:

    case 2:

    ...

    case n:
}
```

In general, use short comments to document variable definitions and block comments to describe computation processes.

Example: block comment vs. short comment

preferred style:

```
/*
 * Main sequence:  get and process all user requests
 */

while (!finish())
{
    inquire();
    process();
}
```

not recommended:

```
while (!finish())  /*  Main sequence:        */
{                  /*                        */
    inquire();     /*  Get user request      */
    process();     /*  And carry it out      */
}                  /*  As long as possible   */
```

2.4 Meaningful Names

Choose names for files, functions, constants, or variables that are meaningful and readable. The following guidelines are recommended for creating element names.

- Choose names with meanings that are precise and use them consistently throughout the program.
- Follow a uniform scheme when abbreviating names. For example, if you have a number of functions associated with the "data refresher," you may want to prefix the functions with "dr_".
- Avoid abbreviations that form letter combinations that may suggest unintended meanings. For example, the name "inch" is a misleading abbreviation for "input character." The name "in_char" would be better.
- Use underscores within names to improve readability and clarity:

```
get_best_fit_model
load_best_estimate_model
```

- Assign names that are unique (with respect to the number of unique characters permitted on your system).
- Use longer names to improve readability and clarity. However, if names are too long, the program may be more difficult to understand and it may be difficult to express the structure of the program using proper indentation.
- Names more than four characters in length should differ by at least two characters. For example, "systst" and "sysstst" are easily confused. Add underscores to distinguish between similar names:

```
systst    sys_tst
sysstst   sys_s_tst
```

- Do not rely on letter case to make a name unique. Although C is case-sensitive (i.e., "LineLength" is different from "linelength" in C), all names should be unique irrespective of letter case. Do not define two variables with the same spelling, but different case.
- Do not assign a variable and a typedef (or struct) with the same name, even though C allows this. This type of redundancy can make the program difficult to follow.

2.4.1 Standard Names

Some standard short names for code elements are listed in the example below. While use of these names is acceptable if their meaning is clear, we recommend using longer, more explicit names, such as "buffer_index."

Example: standard short names

c	characters
i, j, k	indices
n	counters
p, q	pointers
s	strings

Example: standard suffixes for variables

_ptr	pointer
_file	variable of type file*
_fd	file descriptor

2.4.2 Variable Names

When naming internal variables used by a function, do not duplicate global variable names. Duplicate names can create **hidden variables**, which can cause your program not to function as you intended. In the following example, the internal variable “total” would override the external variable “total.” In the corrected example, the internal variable has been renamed “grand_total” to avoid the duplication.

Example: hidden variable

```
int total;
int func1(void)
{
    float total;       /* this is a hidden variable */
    ...
}
```

Example: no hidden variable

```
int total;
int func1(void)
{
    float grand_total; /* internal variable is unique */
    ...
}
```

In separate functions, variables that share the same name can be declared. However, the identical name should be used only when the variables also have the identical meaning. When the meanings of two variables are only similar or coincidental, use unique names to avoid confusion.

2.4.3 Capitalization

The following capitalization style is recommended because it gives the programmer as well as the reader of the code more information.

- **Variables:** Use lower-case words separated by underscores.
- **Function names:** Capitalize the first letter of each word; do not use underscores.
- **Constants:** Use upper-case words separated by underscores.
- **C bindings:** Use the letter “c” followed by an underscore and the binding name.

Example: capitalization style

open_database	variables
ProcessError	function names
MAX_COUNT	constants
c_ephemrd	C bindings

2.4.4 Type and Constant Names

- **Type names** (i.e., created with typedef): Follow the naming standards for global variables.
- **Enumeration types** (declared using enum) and **constants** declared using const: Follow the naming conventions for constants.

3 PROGRAM ORGANIZATION

This section discusses organizing program code into files. It points out good practices such as grouping logically related functions and data structures in the same file and controlling the visibility of the contents of those files. Figure 2 illustrates the organizational schema that the discussion will follow.

```
Program  README
         Standard libraries      <stdio.h>
                                 <math.h>
         Header files            "globals.h"
                                 "types.h"
         Program files           program_file.c
                                      File prolog
                                      Usage and operating instructions
                                      Header file includes
                                      External definitions and declarations
                                      Functions
                                           Function prolog
                                           Function parameters
                                           Internal definitions and declarations
                                           Statements
                                                Operators
                                                Expressions
                                      More external data
                                      More functions
         Module files            module_file.c
         Compilation utilities   Makefile
```

Figure 2 Program Organization

3.1 Program Files

A C program consists of one or more program files, one of which contains the main( ) function, which acts as the driver of the program. An example of a program file is given in Section 9. When your program is large enough to require several files, you should use encapsulation and data hiding techniques to group logically related functions and data structures into the same files. Organize your programs as follows:

- Create a README file to document what the program does.
- Group the main function with other logically related functions in a program file.
- Use module files to group logically related functions (not including the main function).
- Use header files to encapsulate related definitions and declarations of variables and functions.
- Write a Makefile to make recompiles more efficient.

3.2 README File

A README file should be used to explain what the program does and how it is organized and to document issues for the program as a whole. For example, a README file might include

- All conditional compilation flags and their meanings.
- Files that are machine dependent.
- Paths to reused components.

3.3 Standard Libraries

A standard library is a collection of commonly used functions combined into one file. Examples of function libraries include “stdio.h” which comprises a group of input/output functions and “math.h” which consists of mathematical functions. When using library files, include only those libraries that contain functions that your program needs. You may create your own libraries of routines and group them in header files.

3.4 Header Files

Header files are used to encapsulate logically related ideas; for example the header file “time.h” defines two constants, three types, and three structures, and declares seven functions needed to process time. Header files may be selectively included in your program files to limit visibility to only those functions that need them.

Header files are included in C source files before compilation. Some, such as “stdio.h” are defined system-wide, and must be included by any C program that uses the standard input/output library. Others are used within a single program or suite of programs. An example of a header file is given in Section 9.

- Use #include <system_name> for system include files.
- Use #include “user_file” for user include files.
- Contain in header files data definitions, declarations, typedefs, and enums that are needed by more than one program.
- Organize header files by function.
- Put declarations for separate subsystems in separate header files.
- If a set of declarations is likely to change when code is ported from one platform to another, put those declarations in a separate header file.
- Avoid private header filenames that are the same as library header filenames. For example, the statement #include <math.h> will include the standard library math header file if the intended one is not found in the current directory.
- Include header files that declare functions or external variables in the file that defines the function or variable. That way, the compiler can do type checking and the external declaration will always agree with the definition.
- Do not nest header files. Use explicit #include statements to include each header file needed in each program file.
- In the prolog for a header file, describe what other headers need to be included for the header to be functional.

3.5 Module Files

A module file contains the logically related functions, constants, types, data definitions and declarations, and functions. Modules are similar to a program file except that they don’t contain the main( ) function.

3.6 Makefiles

Makefiles are used on some systems to provide a mechanism for efficiently recompiling C code. With makefiles, the make utility recompiles files that have been changed since the last compilation. Makefiles also allow the recompilation commands to be stored, so that potentially long cc commands can be greatly abbreviated. An example of a Makefile is given in Section 9. The makefile

- Lists all files that are to be included as part of the program.

- Contains comments documenting what files are part of libraries.
- Demonstrates dependencies, e.g., source files and associated headers using implicit and explicit rules.

3.7 Standard Filename Suffixes

The suggested format for source code filenames is an optional prefix (e.g., to indicate the subsystem), a base name, and an optional period and suffix. The base name should be unique (length may vary depending on your compiler; some limit filenames to eight or fewer characters) and should include a standard suffix that indicates the file type. Some compilers and tools require certain suffix conventions for filenames. Figure 3 lists some standard suffixes; or use those dictated by your compiler.

File Type	Standard Suffix
C source file	.c
Assembler source	.s
Relocatable object	.o
Include header	.h
Yacc source	.y
Lex source	.l
Loader output file	.out
Makefile	.mak
Linker response files	.lnk *or* .rsp

Figure 3 Standard Filename Suffixes

4 FILE ORGANIZATION

The organization of information within a file is as important to the readability and maintainability of your programs as the organization of information among files. In this section, we will discuss how to organize file information consistently. Figure 4 provides an overview of how program file and module information should be organized.

```
File Prolog, including the algorithm expressed in PDL

Usage and Operating Instructions, if applicable for program files only

Header File Includes, in this sequence:
        #include <stdio.h> (or <stdlib.h>)
        #include <other system headers>
        #include "user header files"

Defines and Typedefs that apply to the file as a whole, including:
        enums
        typedefs
        constant macro defines
        function macro defines

External Data Declarations used by this file
        extern declarations of variables defined in other files
        non-static external definitions used in this file (and optionally
          in others if they are declared in those files using extern)
        static external definitions used only in this file

Functions
        function prolog
        function body

More External Data Declarations used from point of declaration to end of file

More Functions
```

Figure 4 File Organization Schema

4.1 File Prolog

A file prolog introduces the file to the reader. Every file must have a prolog. Figure 5 is an example of a prolog outline; field values are described below.

```
/*******************************************************************
 * FILE NAME:                                                      *
 *                                                                 *
 * PURPOSE:                                                        *
 *                                                                 *
 * FILE REFERENCES:                                                *
 *                                                                 *
 * Name                     I/O         Description                *
 * ----                     ---         -----------                *
 *                                                                 *
 * EXTERNAL VARIABLES:                                             *
 * Source:  <         >                                            *
 *                                                                 *
 * Name       Type          I/O         Description                *
 * ----       ----          ---         -----------                *
 *                                                                 *
 * EXTERNAL REFERENCES:                                            *
 *                                                                 *
 * Name                     Description                            *
 * ----                     -----------                            *
 *                                                                 *
 * ABNORMAL TERMINATION CONDITIONS, ERROR AND WARNING MESSAGES:    *
 *                                                                 *
 * ASSUMPTIONS, CONSTRAINTS, RESTRICTIONS:                         *
 *                                                                 *
 * NOTES:                                                          *
 *                                                                 *
 * REQUIREMENTS/FUNCTIONAL SPECIFICATIONS REFERENCES:              *
 *                                                                 *
 * DEVELOPMENT HISTORY:                                            *
 *                                                                 *
 * Date    Author    Change Id    Release    Description Of Change *
 * ----    ------    ---------    -------    --------------------- *
 *                                                                 *
 * ALGORITHM (PDL)                                                 *
 *                                                                 *
 *******************************************************************/
```

Figure 5 Program File Prolog Contents

- **File Name—**Specify the name of the file.
- **Purpose—** Briefly state the purpose of the unit.

- **File References—**Specify the name, I/O, and description of files used by functions within this file. If the file does not have file references, indicate so by entering “none.”
- **External Variables**—Specify the source, name, type, I/O, and description of variables being used by the unit that do not come in through the calling sequence. If the unit does not have external variables, indicate so by entering “none.”
- **External References**—Specify the exact name of each unit called or invoked by this unit, followed by a one-line description of the unit. If the unit does not have external references, indicate so by entering “none.”
- **Abnormal Termination Conditions, Error and Warning Messages—** Describe the circumstances under which the unit terminates abnormally. List error messages that this unit issues and briefly explain what triggers each.
- **Assumptions, Constraints, Restrictions**—Describe the assumptions that are important to the design and implementation of the unit (e.g., “It is assumed that all input data have been checked for validity.”) Include descriptions of constraints and restrictions imposed by the unit (e.g., “The unit must complete its execution within 75 microseconds.”) This section contains information that explains the characteristics and peculiarities of the unit.
- **Notes—**Specify any additional information needed to understand the file’s data or functions.
- **Requirements/Functional Specifications References—**Provide traceability between requirements and specifications and implementation.
- **Development History—**Outline the file’s development history:
  - **Date**, day, month, and year of the change
  - **Author**, author of the current implementation or change to the unit
  - **Change Id**, an identification number for the change; e.g., if the change is related to a numbered SPR, that number may be used to correlate the change to the SPR
  - **Release**, current software release and build in abbreviated form
  - **Description of Change**, brief narrative describing the change
- **Algorithm (PDL)—**Describe the algorithm used in the program in PDL format. See Section 4.2 for a detailed discussion of algorithm/PDL.

Header files (non-program files) such as those containing global definitions, prototypes, or typedefs, should have an abbreviated prolog as shown in Figure 6.

```
/*****************************************************************
 * NAME:                                                          *
 *                                                                *
 * PURPOSE:                                                       *
 *                                                                *
 * GLOBAL VARIABLES:                                              *
 *                                                                *
 * Variable          Type        Description                      *
 * --------          ----        -----------                      *
 *                                                                *
 * DEVELOPMENT HISTORY:                                           *
 *                                                                *
 * Date     Author   Change Id   Release   Description Of Change   *
 * ----     ------   ---------   -------   ---------------------   *
 *                                                                *
 *****************************************************************/
```

Figure 6 Header File Prolog

4.2 Program Algorithm and PDL

This section of the file prolog describes the overall algorithm of the program or any special or nonstandard algorithms used. This description in the prolog does not eliminate the need for inline comments next to the functions. In fact, adding comments to your functions is recommended to help others understand your code.

In the SEL environment, programmers follow a prescribed PDL style which is documented both in the *Programmer's Handbook for Flight Dynamics Software Development* as well as CSC’s *SSDM* (see Bibliography). The PDL constructs are summarized here, along with the corresponding C code. These guidelines are consistent with the *Programmer's Handbook*.

PDL describes the processing and control logic within software units through the use of imperative English phrases and simple control statements. Follow these general guidelines when creating PDL.

- Indent by four spaces the statements defining the processing to occur within a PDL control structure (unless the code is highly nested and it would run off the right side of the page).
- Within a control structure, align each PDL control structure keyword (e.g., align the IF, ELSE, etc.). Also align each embedded statement.

- If a single PDL statement spans multiple print lines, begin each statement continuation line one space to the right of the parent line.

PDL includes four types of statements, which are described in detail in the paragraphs to follow:

- Sequence
- Selection Control
- Iteration Control
- Severe Error and Exception Handling

4.2.1 Sequence Statements

A PDL sequence statement describes a processing step that does not alter logic flow. Specify this type of PDL statement as a declarative English-language sentence beginning with a single imperative verb followed by a single direct object.

```
verb object
```

Assignment statements may be used only in the event that mathematical formula must be specified.

```
C = A + B
```

To call a unit, use a verb (e.g., CALL) followed by the unit name. The unit name may be followed by a list of descriptive parameters from the calling sequence to that unit or by a phrase describing the function or purpose of the unit being called.

```
CALL <unit name>
```

To signal the end of processing within a unit, use the verb RETURN. A return statement implies an immediate return to the calling entity.

```
RETURN
```

4.2.2 Selection Control Statements

Selection control statements define the conditions under which each of several independent processing paths is executed. There are three PDL selection control structures: **IF THEN ELSE, IF THEN,** and **CASE**. Each of them is shown below in its PDL format and with an example of corresponding C code.

4.2.2.1 IF THEN ELSE

The basic format of an if then else statement is:

```
IF condition THEN
   true processing
ELSE
   false processing
ENDIF
```

Example: PDL

```
IF shuttle and payload mode THEN
   CALL addstr to display shuttle title
ELSE IF freeflyer only mode THEN
   CALL addstr to display ff title
ELSE
   CALL addstr to display both titles
ENDIF
```

Example: C code

```
if (objdisp == SHUT_PAYLOAD)
   addstr("SHUTTLE DATA");
else if (objdisp == FF)
   addstr("FREEFLYER DATA");
else
   addstr("SHUTTLE/FF DATA");
```

4.2.2.2 IF THEN

The general format of an if then statement is:

```
IF condition THEN
   true processing
ENDIF
```

Example: PDL

```
IF offset between request time and time of last calculated
 s/c position and velocity vectors exceeds wait time THEN
   COMPUTE elapsed seconds between epoch time and request
    time
ENDIF
```

Example: C code

```
if ((t_request - t_rv_ref) > t_wait)
   eptime = t_request - orbital_t_epoch;
```

4.2.2.3 CASE

The general format of a case statement is:

```
DO CASE of (name)
CASE 1 condition:
    case 1 processing
CASE 2 condition:
    case 2 processing
.
.
.
CASE n condition:
    case n processing
ELSE (optional)
    else-condition processing
ENDDO CASE
```

OTHERWISE can be substituted for the ELSE keyword.

Example: PDL

```
DO CASE of axes color
    black:
        set color to black
    yellow:
        set color to yellow
    red:
        set color to red
    OTHERWISE:
        set color to green
ENDDO CASE
```

Example: C code

```
switch (axescolor)
{
    case 'B':
        color = BLACK;
        break;
    case 'Y':
        color = YELLOW;
        break;
    case 'R':
        color = RED;
        break;
    default:
        color = GREEN;
        break;
}
```

4.2.3 Iteration Control Statements

Iteration control statements specify processing to be executed repeatedly. There are three basic iteration control structures in PDL: **DO WHILE, DO FOR,** and **DO UNTIL**.

4.2.3.1 DO WHILE

The general format of a do while statement is:

```
DO WHILE "continue loop" condition true
    true processing
ENDDO WHILE
```

Example: PDL

```
DO WHILE ui buffer not empty
    CALL process_ui issue requests
ENDDO WHILE
```

Example: C code

```
while (ui_buf != EMPTY)
    process_ui(ui_buf, num);
```

4.2.3.2 DO FOR

The general format of a do for statement is:

```
DO FOR specified discrete items
    loop processing
ENDDO FOR
```

Example: PDL

```
DO FOR each axis view (X, Y, Z)
    CALL setview to create view
ENDDO FOR
```

Example: C code

```
for (i=0; i < 4; i++)
    setview(sys, i);
```

4.2.3.3 DO UNTIL

The general format of a do until statement is:

```
DO UNTIL "exit loop" condition true
    loop processing
ENDDO UNTIL
```

Example: PDL

```
DO UNTIL no ui requests remain
    CALL process_ui to issue requests
ENDDO UNTIL
```

Example: C code

```
do
    process_ui(ui_buf, num);
while (ui_count != 0);
```

4.2.4 Severe Error and Exception Handling Statements

When a serious error or abnormal situation occurs several levels deep in if or do statements, you may want simply to set an error flag and return to the caller. Using only the constructs described so far, the choices are limited to setting an abort flag and checking at each level of nesting. This can quickly complicate an otherwise clean design. Two PDL statements are available to aid in the handling of severe errors and exceptions: **ABORT** to (abort_label) and **UNDO**.

4.2.4.1 ABORT

ABORT to is used to jump to a named block of processing at the end of the routine. The block's purpose is to set a fatal error indication and exit the routine. Placing all abort processing at the end of the routine helps all abnormal condition logic to stand out from the normal processing.

Example: PDL

```
DO WHILE more records remain to be processed
    read next record from file
    IF an invalid record is encountered
        ABORT to INV_REC_FND
    ENDIF
```

(cont'd next page)

Example: ABORT PDL (cont'd)

```
    (process this record)
ENDDO WHILE
...
RETURN
INV_REC_FND:
    inform user of the invalid record just found
    set invalid record indicator
RETURN
```

In C, you use a **goto** statement to exit out of nested loops. Note that you should use goto statements only for unusual circumstances. In most cases, it is possible to use structured code instead of using a goto. The two examples below show the same scenario using structured code and using a goto statement.

Example: structured code

```
    while (... && no_error)
        for (...)
            if (disaster)
                error = true;
    if error
        error_processing;
```

Example: goto statement

```
    while (...)
        for (...)
            if (disaster)
                goto error;
error:
    error_processing;
```

4.2.4.2 UNDO

UNDO is used within a do (while, for, until) construct to terminate the current loop immediately. That is, processing jumps to the statement following the ENDDO of the current do construct. In C, you could use a break statement to exit out of an inner loop. If you can avoid the use of breaks, however, do so.

Example: PDL

```
DO WHILE more records remain to be processed
    read next record from file
    IF an invalid record is encountered
        UNDO
    ENDIF
    (process this record)
ENDDO WHILE
```

Example: C code with break statement

```
while <more records remain to be processed>
{
    read next record from file
    if <an invalid record is encountered>
        break;
    process this record
}
```

Example: C code with no break statement

```
while (more records remain to be processed && no_error)
{
    read next record from file
    if <an invalid record is encountered>
        error = true;
    else
        process this record
}
```

4.3 Include Directive

To make header file information available to your program files, you must specifically include those header files using the #include preprocessor directive. For optimum efficiency and clarity, include only those header files that are necessary.

- If the reason for the #include is not obvious, it should be commented.
- The suggested file order is:

```
#include <stdio.h> (or <stdlib.h>)
#include <other system headers>
#include "user header files"
```

4.4 Defines and Typedefs

After including all necessary header files, define constants, types, and macros that should be available to the rest of the file (from the point of declaration to the end of the file). Include the following, in the sequence shown:

- Enums
- Typedefs
- Constant macros (#define identifier token-string)
- Function macros (#define identifier(identifier, ..., identifier) token-string)

4.5 External Data Declarations and Definitions

After defining constants, types, and macros, you should next have a section in your file to declare external variables to make them visible to your current file. Define those variables that you want to be available ("global") to the rest of the file. The suggested sequence for declaring and defining external data is:

- Extern declarations of variables defined in other files
- Non-static external definitions used in this file (and, optionally, in others if they are declared in those files using the extern declaration)
- Static external definitions used only in this file

4.6 Sequence of Functions

This section provides general guidelines for arranging functions in the program file. The organization of information within functions is described in Section 5.

- If the file contains the main program, then the main( ) function should be the first function in the file.
- Place logically related functions in the same file.
- Put the functions in some meaningful order.
  - A breadth-first approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible before or after their calls).
  - If defining a large number of essentially independent utility functions, use alphabetical order.
- To improve readability, separate functions in the same file using a single row of asterisks.

- Place functions last in a program file, unless (due to data hiding) you need to declare external variables between functions.

Example: functions with separators

```
/*****************************************************/

main prolog
main body

/*****************************************************/

function_a prolog
function_a body

/*****************************************************/

function_b prolog
function_b body

/*****************************************************/
```

Example: functions with an external variable

```
/*****************************************************/

func1()
{
 ...
}

/*****************************************************/

/* The following external variable will be available
/* to func2 but not to func1 */

int count;

/*****************************************************/

func2()
{
...
}
```

5 FUNCTION ORGANIZATION

This section discusses guidelines for organizing information within functions. Figure 7 provides an overview of how information should be organized within functions.

```
Function prolog
        Name of the function
        Arguments of the function
        Return value of the function
Function argument declarations
        External variable declarations
        Internal variable declarations
            Automatic internal variable definitions
            Static internal variable definitions
Statement "paragraphs" (major sections of the code)
        Block comment introducing the algorithm to be performed by
          the group of statements
        Statements (one per line)
Return statement
```

Figure 7 Function Organization Schema

5.1 Function Prologs

Every function should have a function prolog to introduce the function to the reader. The function prolog should contain the following information:

- **Function name**
  - One or more words all in lower case and separated by underscores
  - Upper case OK if name includes a proper noun (e.g., Gaussian_distribution)
  - Followed by brief descriptive comment
- **Arguments** listed one per line with the type, I/O, and a brief description
- **Return value** describes what the function returns

Example: function prolog

```
/*******************************************************
*                                                      *
* FUNCTION NAME:                                       *
*                                                      *
* ARGUMENTS:                                           *
*                                                      *
* ARGUMENT     TYPE    I/O     DESCRIPTION        *
* --------     ----    ---     -----------        *
*                                                      *
* RETURNS:                                             *
*                                                      *
*******************************************************/
```

For a function with a non-boolean return value or no return value (a return of void), the name should be an imperative verb phrase describing the function's action, or a noun phrase. For a function that returns a boolean value, its name should be a predicate-clause phrase.

Example: imperative verb phrase

```
obtain_next_token
increment_line_counter
```

Example: noun phrase

```
top_of_stack
sensor_reading
```

Example: predicate-clause phrase

```
stack_is_empty
file_is_saved
```

5.2 Function Arguments

Declare function arguments when the function is defined (even if the type is integer). Define functions arguments beginning in column 1. Note that arguments are explained in the function prolog, and therefore do not require explanatory comments following the function declaration.

Example: function argument declarations

```
int getline (char *str, int length)
{
   ...
}
```

5.3 External Variable Declarations

Declare external variables immediately after the opening brace of the function block.

Example: external variable declaration

```
char *save_string(char *string)
{
   extern char      *malloc();
   ...
}
```

5.4 Internal Variable Declarations

Internal variables—i.e., those used only by the function (also known as local variables)—should be defined after the external variables. Follow these guidelines for internal-variable declarations:

- Align internal variable declarations so that the first letter of each variable name is in the same column.
- Declare each internal variable on a separate line followed by an explanatory comment.
  - The only exception is loop indices, which can all be listed on the same line with one comment.
- If a group of functions uses the same parameter or internal variable, call the repeated variable by the same name in all functions.
- Avoid internal-variable declarations that override declarations at higher levels; these are known as hidden variables. See Section 2.4.2 for a discussion of hidden variables.

5.5 Statement Paragraphing

Use blank lines to separate groups of related declarations and statements in a function (statement "**paragraphing**") to aid the reader of the code. In addition, inline comments can be added to explain the various parts of the function.

Example: statement paragraphing

```
char *save_string(char *string)
{
    register char *ptr;

    /*
     * if allocation of the input string is successful,
     * save the string and return the pointer; otherwise,
     * return null pointer.
     */

    if ((ptr = (char *) malloc(strlen(string) + 1)) !=
        (char *) NULL)

        strcpy(ptr, string);

    return(ptr);
}
```

5.6 Return Statement

The **return statement** is the mechanism for returning a value from the called function to its caller. Any expression can follow return:

```
return (expression)
```

- Using an expression in the return statement may improve the efficiency of the code. Overdoing its use, however, increases the difficulty of debugging.
- Do not put multiple return and exit statements in a function, unless following this rule would result in convoluted logic that defeats the overriding goal of maintainability.
- Always declare the return type of functions. Do not default to integer type (int). If the function does not return a value, then give it return type void.
- A single return statement at the end of a function creates a single, known point which is passed through at the termination of function execution.
- The single-return structure is easier to change. If there is more to do after a search, just add the statement(s) between the for loop and the return.

Example: single return

```
found = FALSE;
for (i=0 ; i<max && !found ; i++)
    if (vec[i] == key )
        found = TRUE;
return(found);
```

Example: multiple returns

```
for (i=0 ; i<max ; i++)
    if (vec[i] == key)
        return(TRUE);
return(FALSE);
```

6 DATA TYPES, OPERATORS, AND EXPRESSIONS

This section provides examples of the proper way to format constant and variable definitions and declarations and discusses data encapsulation techniques. There are several general guidelines to follow when working with types:

- Define one variable or constant per line.
- Use short comments to explain all variables or constants.
- Group related variables and constants together.

6.1 Variables

When declaring variables of the same type, declare each on a separate line unless the variables are self-explanatory and related, for example:

```
int year, month, day;
```

Add a brief comment to variable declarations:

```
int x;  /* comment */
int y;  /* comment */
```

Group related variables. Place unrelated variables, even of the same type, on separate lines.

```
int x, y, z;
int year, month, day;
```

6.2 Constants

When defining constants, capitalize constant names and include comments. In constant definitions, align the various components, as shown in the examples below. In ANSI C, there are several ways to specify constants: **const modifier, #define command**, and **enumeration data types.**

6.2.1 Const Modifier

Use the const modifier as follows:

```
const int SIZE 32;     /* size in inches */
const int SIZE 16 + 16; /* both evaluate to the number 32 */
```

6.2.2 #define Command

The #define preprocessor command instructs the preprocessor to replace subsequent instances of the identifier with the given string of tokens. It takes the form:

```
#define IDENTIFIER token-string
```

In general, avoid hard-coding numerical constants and array boundaries. Assign each a meaningful name and a permanent value using #define. This makes maintenance of large and evolving programs easier because constant values can be changed uniformly by changing the #define and recompiling.

```
#define NULL    0
#define EOS     '\0'
#define FALSE   0
#define TRUE    1
```

Using constant macros is a convenient technique for defining constants. They not only improve readability, but also provide a mechanism to avoid hard-coding numbers.

6.2.3 Enumeration Types

Enumeration types create an association between constant names and their values. Using this method (as an alternative to #define), constant values can be generated, or you can assign the values. Place one variable identifier per line and use aligned braces and indentation to improve readability. In the example below showing generated values, low would be assigned 0, middle 1, and high 2. When you assign values yourself, align the values in the same column, as shown in the second example.

Example: generated values

```
enum position
{
    LOW,
    MIDDLE,
    HIGH
};
```

Example: assigned values

```
enum stack_operation_result
{
   FULL       = -2,
   BAD_STACK  = -1,
   OKAY       = 0,
   NOT_EMPTY  = 0,
   EMPTY      = 1
};
```

6.2.4 Simple Constants

Use the const modifier instead of the #define preprocessor to define simple constants. This is preferable because #define cannot be used to pass the address of a number to a function and because #define tells the preprocessor to substitute a token string for an identifier, which can lead to mistakes (as illustrated in the example below).

Example: using #define

```
#define SIZE 10 + 10 /* 10 + 10 will be substituted for SIZE */
...
area = SIZE * SIZE;  /* this evaluates to 10 + 10 * 10 + 10  */
                     /* which is 10 + (10 * 10) + 10 = 120   */
```

Example: using the const modifier

```
const int SIZE = 10 + 10; /* SIZE evaluates to the number 20 */
...
area = SIZE * SIZE;      /* this evaluates to 20 * 20 = 400 */
```

6.3 Variable Definitions and Declarations

6.3.1 Numbers

Floating point numbers should have at least one number on each side of the decimal point:

```
0.5 5.0 1.0e+33
```

Start hexadecimal numbers with 0x (zero, lower-case x) and upper case A-F:

```
0x123  0xFFF
```

End long constants in upper-case L:

```
123L
```

6.3.2 Qualifiers

Always associate qualifiers (e.g., short, long, unsigned) with their basic data types:

```
short int x;
long int y;
unsigned int z;
```

6.3.3 Structures

The use of structures is one of the most important features of C. Structures enhance the logical organization of your code, offer consistent addressing, and will generally significantly increase the efficiency and performance of your programs.

Using common structures to define common elements allows the program to evolve (by adding another element to the structure, for example), and lets you modify storage allocation. For example, if your program processes symbols where each symbol has a name, type, flags, and an associated value, you do not need to define separate vectors.

Example: structures

```
typedef struct symbol
{
   char   *name;
   int type;
   int flags;
   int value;
} symbol_type;
symbol_type symbol_table[NSYMB];
```

6.3.4 Automatic Variables

An automatic variable can be initialized either where it is declared or just before it is used. If the variable is going to be used close to where it is declared (i.e., less than one page later), then initialize it where it is declared. However, if the variable will be used several pages from where it is declared, then it is better practice to initialize it just before it is used.

Example: variable initialized where declared

```
int max = 0;
/* use of max is within a page of where it is declared */
for (i=0; i<n; i++)
    if (vec[i] > max)
        max = vec[i];
```

Example: variable initialized where used

Use an assignment statement just before the for loop:

```
int max;
...
/* several pages between declaration and use */
...
max = 0;
for (i=0 ; i<n ; i++)
    if (vec[i] > max)
        max = vec[i];
```

Or use the comma operator within the for loop:

```
int max;
...
/* several pages between declaration and use */
...
for (max = 0, i=0; i<n; i++)
    if (vec[i] > max)
        max = vec[i];
```

6.4 Type Conversions and Casts

Type conversions occur by default when different types are mixed in an arithmetic expression or across an assignment operator. Use the cast operator to make type conversions explicit rather than implicit.

Example: explicit type conversion (recommended)

```
float f;
int i;
...
f = (int) i;
```

Example: implicit type conversion

```
float f;
int i;
...
f = i;
```

6.5 Pointer Types

Explicitly declare pointer entities (variables, function return values, and constants) with pointer type. Put the pointer qualifier (*) with the variable name rather than with the type.

Example: pointer declaration

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

6.6 Pointer Conversions

Programs should not contain pointer conversions, except for the following:

- **NULL** (i.e., integer 0) may be assigned to any pointer.
- **Allocation functions** (e.g., malloc) will guarantee safe alignment, so the (properly cast) returned value may be assigned to any pointer. Always use sizeof to specify the amount of storage to be allocated.
- **Size**. Pointers to an object of given size may be converted to a pointer to an object of smaller size and back again without change. For example, a pointer-to-long may be assigned to a pointer-to-char variable which is later assigned back to a pointer-to-long. Any use of the intermediate pointer, other than assigning it back to the original type, creates machine-dependent code. Use it with caution.

6.7 Operator Formatting

- Do not put space around the **primary operators:** -› , . , and [ ] :

  ```
  p->m    s.m    a[i]
  ```

- Do not put a space before **parentheses** following function names. Within parentheses, do not put spaces between the expression and the parentheses:

  ```
  exp(2, x)
  ```

- Do not put spaces between **unary operators** and their operands:

```
!p     ~b     ++i -n     *p     &x
```

- **Casts** are the only exception. *do put a space* between a cast and its operand:

```
(long) m
```

- Always put a space around **assignment operators**:

```
c1 = c2
```

- Always put a space around **conditional operators**:

```
z = (a > b) ? a : b;
```

- **Commas** should have one space (or newline) after them:

```
strncat(t, s, n)
```

- **Semicolons** should have one space (or newline) after them:

```
for (i = 0; i < n; ++i)
```

- For **other operators,** generally put one space on either side of the operator:

```
x + y      a < b && b < c
```

- Occasionally, these operators may appear with **no space** around them, but the operators with no space around them must **bind** their operands tighter than the adjacent operators:

```
printf(fmt, a+1)
```

- Use **side-effects** within expressions sparingly. No more than one operator with a side-effect (=, op=, ++, --) should appear within an expression. It is easy to misunderstand the rules for C compilation and get side-effects compiled in the wrong order. The following example illustrates this point:

```
if ((a < b) && (c==d)) ...
```

  If a is not < b, the compiler knows the entire expression is false so (c == d) is never evaluated. In this case, (c == d) is just a test/relational expression, so there is no problem. However, if the code is:

```
if ((a < b) && (c==d++))
```

  d will only be incremented when (a < b) because of the same compiler efficiency demonstrated in the first example.

CAUTION: Avoid using side-effect operators within relational expressions. Even if the operators do what the author intended, subsequent reusers may question what the desired side-effect was.

- Use **comma operators** exceedingly sparingly. One of the few appropriate places is in a for statement. For example:

```
for (i = 0, j = 1; i < 5; i++, j++);
```

- Use **parentheses** liberally to indicate the **precedence** of operators. This is especially true when mask operators (&, |, and ^) are combined with shifts.

- Split a string of conditional operators that will not fit on one line onto separate lines, breaking after the logical operators:

```
if (p->next == NULL &&
    (total_count < needed) &&
    (needed <= MAX_ALLOT) &&
    (server_active(current_input)))
{
    statement_1;
    statement_2;
    statement_n;
}
```

6.8 Assignment Operators and Expressions

C is an expression language. In C, an assignment statement such as “a = b” itself has a value that can be embedded in a larger context. *We recommend that you use this feature very sparingly*. The following example shows a standard C idiom with which most C programmers are familiar.

Example: embedded assignments

```
while ((c = getchar()) != EOF)
{
    statement_1;
    statement_2;
    statement_n;
}
```

However, do not overdo embedding of multiple assignments (or other side-effects) in a statement. Consider the tradeoff between increased speed and decreased maintainability that results when embedded statements are used in artificial places.

Example: nonembedded statements

```
total = get_total ();
if (total == 10)
   printf("goal achieved\n");
```

Example: embedded statements (not recommended)

```
if ((total = get_total() == 10)
     printf("goal achieved\n")
```

6.9 Conditional Expressions

In C, conditional expressions allow you to evaluate expressions and assign results in a shorthand way. For example, the following if then else statement

```
if (a > b)
    z = a;
else
    z = b;
```

could be expressed using a conditional expression as follows:

```
z = (a > b) ? a : b;  /* z = max(a, b) */
```

While some conditional expressions seem very natural, others do not, and we generally recommend against using them. The following expression, for example, is not as readable as the one above and would not be as easy to maintain:

```
c = (a == b) ? d + f(a) : f(b) - d;
```

Do not use conditional expressions if you can easily express the algorithm in a more clear, understandable manner. If you do use conditional expressions, use comments to aid the reader's understanding.

6.10 Precedence and Order of Evaluation

There are 21 precedence rules. Rather than trying to memorize the rules or look them up every time you need them, remember these simple guidelines from Steve Oualline's *C Elements of Style*:

- * % / come before + and -
- Put ( ) around everything else

7 STATEMENTS AND CONTROL FLOW

This section describes how to organize statements into logical thoughts and how to format various kinds of statements. The general principles for writing clear statements are as follows:

- Use blank lines to organize statements into paragraphs and to separate logically related statements.
- Limit the complexity of statements, breaking a complex statement into several simple statements if it makes the code clearer to read.
- Indent to show the logical structure of your code.

7.1 Sequence Statements

This section describes the rules for formatting statements in blocks.

7.1.1 Statement Placement

Put only one statement per line (except in for loop statements):

```
switch (axescolor)
{
   case 'B':
       color = BLACK;
       break;
   case 'Y':
       color = YELLOW;
       break;
   case 'R':
       color = RED;
       break;
   default:
       color = GREEN;
       break;
}
```

Avoid statements that rely on side-effect order. Instead, put the variables with operators ++ and -- on lines by themselves:

```
*destination = *source;
destination++;
source++;
a[i] = b[i++];
```

It is recommended that you use **explicit comparison** even if the comparison value will never change. For example, this statement:

```
if (!(bufsize % sizeof(int)))
```

should be written instead as

```
if ((bufsize % sizeof(int)) == 0)
```

to reflect the numeric (not boolean) nature of the test.

7.1.2 Braces

Compound statements, also known as blocks, are lists of statements enclosed in braces. The brace style we recommend is the Braces-Stand-Alone method. Place braces on separate lines and align them. This style, which is used throughout this document, allows for easier pairing of the braces and costs only one vertical space.

Example: Braces-Stand-Alone method

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
{
    c = s[i];
    s[i] = s[j];
    s[j] = c;
}
```

Although C does not require braces around single statements, there are times when braces help improve the readability of the code. **Nested conditionals** and **loops** can often benefit from the addition of braces, especially when a conditional expression is long and complex.

The following examples show the same code with and without braces. We encourage the use of braces to improve readability. Use your own judgment when deciding whether or not to use braces, remembering that what is clear to you may not be obvious to others who read your code.

Example: braces improve readability

```
for (dp = &values[0]; dp < top_value; dp++)
{
    if (dp->d_value == arg_value
     && (dp->d_flag & arg_flag) != 0)
    {
        return (dp);
    }
}
return (NULL);
```

Example: no braces

```
for (dp = &values[0]; dp < top_value; dp++)
    if  (dp->d_value == arg_value &&
        (dp->d_flag & arg_flag) != 0)
        return (dp);
return (NULL);
```

- If the span of a block is large (more than about 40 lines) or there are several nested blocks, comment closing braces to indicate what part of the process they delimit:

```
for (sy = sytable; sy != NULL; sy = sy->sy_link)
{
    if (sy->sy_flag == DEFINED)
    {
        ...
    }          /* if defined        */
    else
    {
        ...
    }          /* if undefined      */
}              /* for all symbols  */
```

- If a for or while statement has a dummy body, the semicolon should go on the next line. It is good practice to add a comment stating that the dummy body is deliberate.

```
/* Locate end of string */
for (char_p = string; *char_p != EOS; char_p++)
    ;   /* do nothing */
```

- Always put a space between reserved words and their opening parentheses.
- Always put parentheses around the objects of sizeof and return.

7.2 Selection Control Statements

This section discusses the recommended formatting for selection control statements. Examples are given to show how to format single statements as well as blocks of statements.

7.2.1 If

- Indent single statements one level:

```
if (expression)
    one_statement;
```

- Indent a block of statements one level using braces:

```
if (expression)
{
    statement_1;
    ...
    statement_n;
}
```

7.2.2 If Else

- If else statements that have only simple statements in both the if and else sections do not require braces but should be indented one level:

```
if (expression)
    statement
else
    statement
```

- If else statements that have a compound statement in either the if or else section require braces and should be indented one level using braces:

```
if (expression)
    one_statement;
else
{
    statement_1;
    ...
    statement_n;
}
```

7.2.3 Else If

For readability, use the following format for else if statements:

```
if (expression)
    statement[s]
else if  (expression)
    statement[s]
else
    statement[s]
```

7.2.4 Nested If Statements

7.2.4.1 If If If

Use nested if statements if there are alternative actions (i.e., there is an action in the else clause), or if an action completed by a successful evaluation of the condition has to be undone. Do not use nested if statements when only the if clause contains actions.

Example: good nesting

```
status = delta_create((Callback)NULL, &delta);
if ( status == NDB_OK )
{
  if ((status = delta_record_condition(...)) == NDB_OK &&
       (status = delta_field_condition(...)) == NDB_OK &&
       (status=delta_field_condition(...)) == NDB_OK )

       status = delta_commit(delta, ...);

  (void)ndb_destroy_delta( delta);
}
```

Example: inappropriate nesting

```
status = delta_create((Callback)NULL, &delta);
if (status == NDB_OK)
{
    status = delta_record_condition( delta, ...);
    if (status == NDB_OK )
    {
        status = delta_field_condition(delta, ...);
        if (status == NDB_OK )
```

(cont'd next page)

Example: inappropriate nesting (cont'd)

```
            {
                status = delta_field_condition( ...);
                if (status == NDB_OK )
                    status = delta_commit(delta, ...);
            }
        }
        (VOID)ndb_destroy_delta(delta);
    }
    return(status);
```

7.2.4.2 If If Else

Because the else part of an if else statement is optional, omitting the "else" from a nested if sequence can result in ambiguity. Therefore, always use braces to avoid confusion and to make certain that the code compiles the way you intended. In the following example, the same code is shown both with and without braces. The first example will produce the results desired. The second example will not produce the results desired because the "else" will be paired with the second "if" instead of the first.

Example: braces produce desired result

```
if (n > 0)
{
    for (i = 0; i < n; i++)
    {
        if (s[i] > 0)
        {
            printf("...");
            return(i);
        }
    }
}
else    /* CORRECT -- braces force proper association  */
    printf("error - n is zero\n");
```

Example: absence of braces produces undesired result

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0)
        {
            printf("...");
            return(i);
        }
else    /* WRONG -- the compiler will match to closest */
        /* else-less if */
    printf("error - n is zero\n");
```

7.2.5 Switch

For readability, use the following format for switch statements:

```
switch (expression)
{
    case aaa:
        statement[s]
        break;
    case bbb: /* fall through */
    case ccc:
        statement[s]
        break;
    default:
        statement[s]
        break;
}
```

Note that the fall-through feature of the C switch statement should be commented for future maintenance.

All switch statements should have a default case, which may be merely a “fatal error” exit. The default case should be last and does not require a break, but it is a good idea to put one there anyway for consistency.

7.3 Iteration Control Statements

This section discusses the recommended formatting for iteration control statements. Examples are given to show how to format single statements as well as blocks of statements.

7.3.1 While

For one statement, use the following format:

```
while (expression)
    one_statement;
```

For a block of statements, use:

```
while (expression)
{
    statement_1;
    ...
    statement_n;
}
```

7.3.2 For

Use the following formats:

```
for (expression)
    one_statement;
for (expression)
{
    statement_1;
    ...
    statement_n;
}
```

If a for loop will not fit on one line, split it among three lines rather than two:

```
for (curr = *listp, trail = listp;
    curr != NULL;
    trail = &(curr->next), curr = curr->next)
{
    statement_1;
    ...
    statement_n;
}
```

7.3.3 Do While

For readability, use the following format:

```
do
{
    statement_1;
    statement_2;
    statement_3;
}
while (expression)
```

7.4 Severe Error and Exception Handling

This section discusses the recommended formatting for **goto** statements and **labels**. We also discuss the use of the **break** statement. Recommendations in this section correspond to the severe error and exception handling guidelines given in Section 4.2.4. Note that although gotos and labels are legal constructs of the C language, we do not recommend using them if you can write clear structured code without them.

7.4.1 Gotos and Labels

Goto statements should be used very sparingly, as in any well-structured code. They are useful primarily for breaking out of several levels of switch, for, and while nesting, as shown in the following example:

```
    for (...)
    {
        for (...)
        {
            ...
            if (disaster)
            {
                goto error;
            }
        }
    }
    ...
error:
    error processing
```

7.4.2 Break

A break statement can be used to exit an inner loop of a for, while, do, or switch statement at a logical breaking point rather than at the loop test. The following examples, which remove trailing blanks and tabs from the end of each input line illustrate the difference.

Example: logical break

```
while ((n = getline(line, MAXLINE)) > 0)
{
    while (--n >= 0)
    {
        if (line[n] != ' ' && line[n] != '\t' &&
         line[n] != '\n')
           break;
    }
}
```

Example: loop test

```
while ((n = getline(line, MAXLINE)) > 0)
{
    while (--n >= 0 &&
     (line[n]==' ' || line[n]=='\t' || line[n]=='\n'))
        ;   /* VOID */
    ...
}
```

8 PORTABILITY AND PERFORMANCE

Code is often developed on one type of computer and then ported to and executed on another. Therefore, it is judicious to make the code as portable as possible, requiring no changes or minimal ones—such as changes to system-specific header files. When writing software, consider the following guidelines that will enhance portability and performance.

8.1 Guidelines for Portability

- Use ANSI C whenever it is available.
- Write portable code first. Consider detailed optimizations only on computers where they prove necessary. Optimized code is often obscure. Optimizations for one computer may produce worse code on another. Document code that is obscure due to performance optimizations and isolate the optimizations as much as possible.
- Some code/functions are inherently nonportable. For example, a hardware device handler, in general, can not be transported between operating systems.
- If possible, organize source files so that the computer-independent code and the computer-dependent code are in separate files. That way, if the program is moved to a new computer, it will be clear which files need to be changed for the new platform.
- Different computers have different word sizes. If you are relying on a (predefined) type being a certain size (e.g., int being exactly 32 bits), then create a new type (e.g., typedef long int32) and use it (int32) throughout the program; further changes will require only changing the new type definition.
- Note that pointers and integers are not necessarily the same size; nor are all pointers the same size. Use the system function sizeof(...) to get the size of a variable type instead of hard-coding it.
- Beware of code that takes advantage of two's complement arithmetic. In particular, avoid optimizations that replace division or multiplication with shifts.
- Become familiar with the standard library and use it for string and character manipulation. Do not reimplement standard routines. Another person reading

your code might see the reimplementation of a standard function and would need to establish if your version does something special.

- Use #ifdefs to conceal nonportable quirks by means of centrally placed definitions.

Example: centrally placed definitions

```
#ifdef decus
#define UNSIGNED_LONG     long
#else
#define UNSIGNED_LONG     unsigned long
#endif
```

8.2 Guidelines for Performance

- Remember that code must be maintained.
- If performance is not an issue, then write code that is easy to understand instead of code that is faster. For example,

replace:	d = (a = b + c) + r;	with:	a = b + c; d = a + r;

- When performance is important, as in real-time systems, use techniques to enhance performance. If the code becomes “tricky” (i.e., possibly unclear), add comments to aid the reader.
- Minimize the number of opens and closes and I/O operations if possible.
- Free allocated memory as soon as possible.
- To improve efficiency, use the automatic increment ++ and decrement operators -- and the special operations += and *= (when side-effect is not an issue).
- ANSI C allows the assignment of structures. Use this feature instead of copying each field separately.
- When passing a structure to a function, use a pointer. Using pointers to structures in function calls not only saves memory by using less stack space, but it can also boost performance slightly. The compiler doesn’t have to generate as much code for manipulating data on the stack and it executes faster.

9 C CODE EXAMPLES

The following examples illustrate many of the principles of good style discussed in this document. They include:

- A **Makefile,** which provides an efficient mechanism for building several executables.
- A **.c file,** which illustrates program file organization and principles of readability.
- An **include** file, which illustrates clear and maintainable definition and organization of constants and external variables.

9.1 Makefile

```
# Makefile for UIX Testing ..
#
#
#
#   J. Programmer

#
#   This makefile can build 8 different executables.  The executables
#   share some of the same code and share libraries.

#
# Object code for the executables
#

INIT_OBJS = oi_seq_init.o oi_seq_drv_1.o

GEN_SCREEN_OBJS = oi_seq_gen_screen_PRIVATE.o\
        oi_seq_drv_1.o \
        oi_seq_resize_pane.o\
        oi_seq_get_pane_sizes_PRIVATE.o\
        oi_seq_init.o

FATAL_OBJS = oi_seq_drv_2.o\
        oi_seq_fatal_PRIVATE.o

PROC_FOCUS_EVENTS_OBJS = oi_seq_drv_3.o\
        oi_seq_proc_focus_events.o

LOAD_OBJS = oi_seq_load_drv.o\
        oi_seq_load.o\
        print_seq.o

SUB_BUILD_1 = \
        oi_seq_init.o\
        oi_seq_gen_screen_PRIVATE.o\
        oi_seq_resize_pane.o\
        oi_seq_get_pane_sizes_PRIVATE.o\
        oi_seq_proc_focus_events.o\
        oi_seq_load.o\
        oi_seq_change_exec_type.o\
        oi_seq_file_error_PRIVATE.o\
        oi_seq_enable_sequence_PRIVATE.o\
        oi_seq_new_app_PRIVATE.o\
        oi_seq_prep_load.o\
        oi_seq_change_current_PRIVATE.o\
        oi_seq_set_detail_pane_PRIVATE.o\
        oi_seq_retrieve_detail_pane_PRIVATE.o\
        oi_seq_subbld_1.o

SUB_BUILD_2 = \
```

```
        oi_seq_init.o\
        oi_seq_gen_screen_PRIVATE.o\
        oi_seq_proc_focus_events.o\
        oi_seq_quit.o\
        oi_seq_seqcr_spawn_PRIVATE.o\
        oi_seq_seqcr_continue.o\
        oi_seq_seqcr_handle_sigchld.o\
        oi_seq_seqcr_start.o\
        oi_seq_seqcr_term.o\
        oi_seq_load.o\
        oi_seq_change_exec_type.o\
        oi_seq_file_error_PRIVATE.o\
        oi_seq_enable_sequence_PRIVATE.o\
        oi_seq_new_app_PRIVATE.o\
        oi_seq_prep_load.o\
        oi_seq_change_current_PRIVATE.o\
        oi_seq_set_detail_pane_PRIVATE.o\
        oi_seq_retrieve_detail_pane_PRIVATE.o\
        oi_seq_new.o\
        oi_seq_remove_app.o\
        oi_seq_check_seq_ui.o\
        oi_seq_seqcr_check_seq_PRIVATE.o\
        oi_seq_insert_app.o\
        oi_seq_reconfigure_pane_PRIVATE.o\
        oi_seq_subbld_2.o

BUILD_2 = \
        oi_seq_change_current_PRIVATE.o\
        oi_seq_change_exec_type.o\
        oi_seq_enable_sequence_PRIVATE.o\
        oi_seq_fatal_PRIVATE.o\
        oi_seq_gen_screen_PRIVATE.o\
        oi_seq_init.o\
        oi_seq_load.o\
        oi_seq_new_app_PRIVATE.o\
        oi_seq_proc_focus_events.o\
        oi_seq_quit.o\
        oi_seq_retrieve_detail_pane_PRIVATE.o\
        oi_seq_save.o\
        oi_seq_set_detail_pane_PRIVATE.o\
        oi_seq_seqcr_check_seq_PRIVATE.o\
        oi_seq_seqcr_continue.o\
        oi_seq_seqcr_handle_sigchld.o\
        oi_seq_seqcr_spawn_PRIVATE.o\
        oi_seq_seqcr_start.o\
        oi_seq_seqcr_term.o\
        oi_seq_data.o\
        oi_seq_reconfigure_pane_PRIVATE.o

        oi_seq_b2_stubs.o\
        oi_session_mgr_main.o
```

```
# These are included in all executables
OBJS = test_main.o oi_seq_data.o stubs.o

INTERNAL_DEFINES = -DTEST_NO_NCSS
DEFINES =
DEBUG = -g
CUSTOM_FLAGS = -posix -W3 -DXTFUNCPROTO -DFUNCPROTO
CFLAGS = $(DEBUG) $(CUSTOM_FLAGS) $(INCDIR) $(DEFINES)\
        $(INTERNAL_DEFINES)

# INCLUDE PATHS
INCDIR = -I/u/cmps3/UIX/dev/include \
        -I/u/cmps3/UIX/codebase5/sco/source

# LIBRARIES
NCSS_LIBS = #-lncss_c -lrpcsvc -lrpc -lsocket
XLIBS = -lXtXm_s -lXmu -lX11_s -lPW

UIXLIBDIR = -L/u/cmps3/UIX/R1/lib/sco -L/u/cmps3/UIX/dev/lib/sco
UIX_LIBS = -luixdiag -luixutil
UIX_LIBS2 = -lmsgr

# Compilation for the executables ...

test_init:  $(INIT_OBJS) $(OBJS)
        $(CC) -o test_init $(INIT_OBJS) $(OBJS) $(UIXLIBDIR)
$(NCSS_LIBS)\
            $(UIX_LIBS) $(XLIBS)

test_gen_screen:    $(GEN_SCREEN_OBJS) $(OBJS)
        $(CC) -o test_gen_screen $(GEN_SCREEN_OBJS) $(OBJS) $(UIXLIBDIR)\
            $(NCSS_LIBS) $(UIX_LIBS) $(XLIBS)

test_fatal: $(FATAL_OBJS) $(OBJS)
        $(CC) -o test_fatal $(FATAL_OBJS) $(OBJS) $(NCSS_LIBS) $(UIXLIBDIR)\
            $(UIX_LIBS) $(XLIBS)

test_proc_focus_events: $(PROC_FOCUS_EVENTS_OBJS) $(OBJS)
        $(CC) -o test_proc_focus_events $(PROC_FOCUS_EVENTS_OBJS) $(OBJS)\
            $(UIXLIBDIR) $(UIX_LIBS)

test_load:  $(LOAD_OBJS) $(OBJS)
        $(CC) -o test_load $(LOAD_OBJS) $(OBJS)\
            $(UIXLIBDIR) $(UIX_LIBS) $(XLIBS)

sub_build_1:    $(SUB_BUILD_1) $(OBJS)
        $(CC) -o $@  $(SUB_BUILD_1) $(OBJS) $(UIXLIBDIR) $(NCSS_LIBS)\
            $(UIX_LIBS) $(XLIBS)

sub_build_2:    $(SUB_BUILD_2) $(OBJS)
        echo $(SUB_BUILD_2)
        $(CC) -o $@  $(SUB_BUILD_2) $(OBJS) $(UIXLIBDIR) $(NCSS_LIBS)\
            $(UIX_LIBS) $(XLIBS)

build_2:    $(BUILD_2)
```

```
        $(CC) -o $@  $(BUILD_2) $(UIXLIBDIR) $(NCSS_LIBS)\
            $(UIX_LIBS) $(XLIBS)

clean:
        /bin/rm $(INIT_OBJS) $(OBJS)  $(GEN_SCREEN_OBJS) $(FATAL_OBJS)\
             $(LOAD_OBJS) $(SUB_BUILD_1)
depend:
    makedepend -- $(CFLAGS) -- `/bin/ls *.c`

# DO NOT DELETE THIS LINE -- make depends on it.

# [a jillion lines that are dependencies generated by makedepend go here]
```

9.2 C Program File: RF_GetReference.c

```
/***********************************************************************
 *
 *
 *  FILE NAME: RF_GetReference.c
 *
 *
 *  PURPOSE:  This function determines if a requested reference
 *            vector is in need of update.  It uses analytic routines
 *            to update vectors and these updates are reflected in the
 *            reference.h include file.
 *
 *
 *  FILE REFERENCES:
 *
 *  Name                      IO     Description
 *  ------------              --     ---------------------------------
 *  none
 *
 *
 *  EXTERNAL VARIABLES:
 *
 *  Source : debug.h
 *
 *  Name                  Type      IO   Description
 *  -------------         -------   --   -----------------------------
 *  debug_file_handle     FILE*     I    File handle for debug file
 *                                         name
 *  debug_level           int[9]    I    Debug level array
 *
 *  Source : HD_reference.h
 *
 *  Name                  Type      IO   Description
 *  -------------         -------   --   -----------------------------
 *  ephem_file_lu         long      I    FORTRAN logical unit number
 *                                         for the ephemeris file
 *  ephem_method          char      I    Method for computing
 *                                         ephemeris information:
 *                                           F = Use ephemeris file
 *                                           A = Compute analytically
 *                                               using Keplerian
 *                                               elements
 *  keplerian             double[6] I    Keplerian orbital elements at
 *                                         the epoch time
 *                                         (orbital_t_epoch):
 *                                           [1] Semimajor axis [km]
 *                                           [2] Eccentricity
 *                                           [3] Inclination [rad]
 *                                           [4] Right ascension of
 *                                               the ascending node
 *                                               [rad]
```

```
*                                               [5] Argument of perigee
*                                                   [rad]
*                                               [6] Mean anomaly [rad]
*  m_order                long      I    Order of magnetic field
*  maxit                  long      I    Maximum number of iterations
*                                          to converge the true
*                                          anomaly
*  MU_E                   double    I    Earth gravitational constant
*                                          [km^3/sec^2]
*  NUMPTS                 int       I    Number of points used by the
*                                          EPHEMRD interpolator
*  orbital_t_epoch        double    I    Base epoch time of the
*                                          orbital elements [sec]
*  THREEB                 double    I    Gravitational constant of
*                                          perturbations [Km^2]
*  ttol                   double    I    Tolerance in the calculations
*                                          of the true anomaly [rad]
*  t_b_ref                double    IO   Time of last calculated Earth
*                                          magnetic field vector [sec]
*  t_e_ref                double    IO   Time of last calculated s/c
*                                          to Earth unit vector [sec]
*  t_m_ref                double    IO   Time of last calculated s/c
*                                          to Moon unit vector [sec]
*  t_o_ref                double    IO   Time of last calculated orbit
*                                          normal unit vector [sec]
*  t_rv_ref               double    IO   Time of last calculated s/c
*                                          position and velocity
*                                          vectors[sec]
*  t_s_ref                double    IO   Time of last calculated s/c
*                                          to Sun unit vector [sec]
*  e_pos                  double[3]  O   S/C to Earth unit vector
*  m_pos                  double[3]  O   S/C to Moon unit vector
*  mag_field              double[3]  O   Earth magnetic field vector
*                                          [mG]
*  mag_field_unit         double[3]  O   Earth magnetic field unit
*                                          vector
*  orbit_normal           double[3]  O   Orbit normal unit vector
*  s_c_pos                double[3]  O   S/C position vector [km]
*  s_c_vel                double[3]  O   S/C velocity vector [km/sec]
*  s_pos                  double[3]  O   S/C to Sun unit vector
*
*
*
*  EXTERNAL REFERENCES:
*
*  Name                Description
*  ------------------  ---------------------------------------------
*  c_ephemrd           Retrieves vectors from an ephemeris file and
*                        interpolates them for a requested time
*  c_calpvs            Generates s/c position and velocity vectors
*                        using J2 effects
*  c_sunlunp           Generates Earth to Sun or Earth to Moon
*                        vectors
*  c_emagfld           Generates Earth magnetic field vectors
*  c_nmlist            Opens the magnetic field file for reading
```

```
*  GetSun               Compute s/c to Sun unit vector
*  GetOrbitNormal       Compute orbit normal vector
*  GetEarth             Compute s/c to Earth vector
*  GetMoon              Compute s/c to Moon unit vector
*  SecsToCalendar       Converts time from secornds to standard
*                         calendar format
*  c_packst             Converts time from standard calendar format to
*                         an unpacked array format
*  c_calmjd             Computes the modified Julian date of an
*                         unpacked array format time
*  c_jgrenha            Computes the Greenwich Hour Angle using
*                         analytical data
*  c_unvec3             Unitizes a vector and computes its magnitude
*
*
*  ABNORMAL TERMINATION CONDITIONS, ERROR AND WARNING MESSAGES:
*    none
*
*
*  ASSUMPTIONS, CONSTRAINTS, RESTRICTIONS: none
*
*
*  NOTES:
*    CALLED BY:  InitReference, CalcNadirAngle, ConvertAttitude,
*                ComputeAttitude, CompSunNad, CalcLambdaPhi
*
*
*  REQUIREMENTS/FUNCTIONAL SPECIFICATIONS REFERENCES:
*    FASTRAD Functional Specifications, Sections 4.3.1 - 4.3.6
*
*
*  DEVELOPMENT HISTORY:
*  Date        Name          Change  Release  Description
*                            ID
*  --------  -------------  ------  -------  --------------------------
*  09-16-93  J. Programmer            1          Prolog and PDL
*  10-25-93  J. Programmer            1          Coded
*  11-16-93  J. Programmer            1          Controlled
*  12-02-93  J. Programmer            1          Integrated new RSL
*                                                  routines
*  12-20-93  J. Programmer      12    1          Created intermediate
*                                                  variables for #define
*                                                  arguments of calpvs
*                                                  in order to pass
*                                                  by address
*  02-15-94  J. Programmer      15    2          Corrected time errors
*                                                  using RSL routines
*  05-03-94  J. Programmer            3          Enhancements to RSL
*                                                  prototypes
*  05-10-94  J. Programmer            3          Added Earth magnetic
*                                                  field read capability
*  05-10-94  J. Programmer            3          Added ephemeris read
*                                                  capability
*
*
```

```
ALGORITHM
*
*
*  DO CASE of reference type
*
*  CASE 1 or 2, request is for s/c position or velocity vectors
*
*    IF offset between request time and time of last calculated s/c
*     position and velocity vectors exceeds wait time THEN
*
*      COMPUTE elapsed seconds between epoch time and request time
*
*      IF ephemeris method is for reading file THEN
*        CALL c_ephemrd to read ephemeris file getting s/c position and
*         velocity vectors
*      ELSE (analytic computation)
*        CALL c_calpvs to generate new s/c position and velocity
          vectors
*      ENDIF
*
*      SET new time of last calculated s/c position and velocity
*       vectors to request time
*
*    ENDIF
*
*    IF reference type is for s/c position vector THEN
*      SET return vector to s/c position vector
*    ELSE
*      SET return vector to s/c velocity vector
*    ENDIF
*
*  CASE 3, request is for s/c to Sun unit vector
*
*    IF offset between request time and time of last calculated s/c to
*     Sun unit vector exceeds wait time THEN
*
*      CALL SecsToCalendar c_packst and c_calmjd to get modified
        Julian date
*      CALL c_sunlunp to generate new Earth to Sun vector
*      CALL GetSun to compute new s/c to Sun unit vector
*
*      SET new time of last calculated s/c to Sun unit vector to
        request time
*
*    ENDIF
```

```
*    SET return vector to s/c to Sun unit vector
*
*  CASE 4 or 5, request is for Earth magnetic field vector or Earth
*   magnetic field unit vector
*
*    IF offset between request time and time of last calculated Earth
*     magnetic field vector exceeds wait time THEN
*
*      CALL SecsToCalendar c_packst and c_calmjd to get modified
        Julian date
*      CALL c_jgrenha to get the Greenwich Hour Angle
*      CALL c_emagfld to generate new Earth magnetic field vector
*      CALL c_unvec3 to SET Earth magnetic field unit vector
*
*      SET new time of last calculated Earth magnetic field vector to
*       request time
*
*    ENDIF
*
*    IF reference type is for Earth magnetic field vector THEN
*      SET return vector to Earth magnetic field vector
*    ELSE
*      SET return vector to Earth magnetic field unit vector
*    ENDIF
*
*  CASE 6, request is for orbit normal unit vector
*
*    IF offset between request time and time of last calculated orbit
*     normal unit vector exceeds wait time THEN
*
*      CALL GetOrbitNormal to generate new orbit normal unit vector
*
*      SET new time of last calculated orbit normal unit vector to
*       request time
*
*    ENDIF
*
*    SET return vector to orbit normal unit vector
*
*  CASE 7, request is for s/c to Moon unit vector
*
*    IF offset between request time and time of last calculated s/c to
*     Moon unit vector exceeds wait time THEN
*
*      CALL SecsToCalendar c_packst and c_calmjd to get modified Julian
        date
*      CALL c_sunlunp to generate new Earth to Moon vector
*      CALL GetMoon to compute new s/c to Moon unit vector
*
*      SET new time of last calculated s/c to Moon unit vector to
        request time
*
*    ENDIF
*
*    SET return vector to s/c to Moon unit vector
```

```
 *
 *  CASE 8, request is for s/c to Earth unit vector
 *
 *    IF offset between request time and time of last calculated s/c to
 *     Earth unit vector exceeds wait time THEN
 *
 *      CALL GetEarth to compute new s/c to Earth unit vector
 *
 *      SET new time of last calculated s/c to Earth unit vector to
 *       request time
 *
 *    ENDIF
 *
 *    SET return vector to s/c to Earth unit vector
 *
 *  END CASE
 *
 *  RETURN
 *
 ******************************************************************** */

/* Include global parameters */

#include "HD_debug.h"
#include "HD_reference.h"

/* Declare Prototypes */

void c_ephemrd (long    , long    , long    , double , double *,
                double *, double *, double *, long *);
void c_calpvs  (double  , double  , double *, double , double  ,
                long    , double *, double *, long *);
void c_sunlunp (double  , double  , double *, double *);
void c_emagfl2 (long    , double  , double  , double , double *,
                long    , double *, long *);
void c_nmlist  (long    , long *  , char *  , long *);
void c_packst  (double  , double *);
void c_calmjd  (double *, double *);
void c_jgrenha (double  , double  , long    , long   , double *,
                long *);
void c_unvec3  (double *, double *, double *);

void   GetSun        (double[3], double[3]);
void   GetOrbitNormal(double[3]);
void   GetEarth      (double[3]);
void   GetMoon       (double[3], double[3]);
double SecsToCalendar(double);

/*******************************************************************
 *
 * FUNCTION NAME:    GetReference
 *
 *    ARGUMENT LIST:
 *
 *    Argument       Type       IO     Description
```

```
 *    -------------  --------   --     --------------------------------
 *    ref_type       int        I      Type of reference data requested
 *                                       = 1, S/C position vector
 *                                       = 2, S/C velocity vector
 *                                       = 3, S/C to Sun unit vector
 *                                       = 4, Earth magnetic field
 *                                            vector
 *                                       = 5, Earth magnetic field unit
 *                                            vector
 *                                       = 6, Orbit normal unit vector
 *                                       = 7, S/C to Moon unit vector
 *                                       = 8, S/C to Earth unit vector
 *    t_request      double     I      Time of requested reference
 *                                       vector
 *    t_wait         double     I      Wait time between reference
 *                                       vector calculations
 *    ref_vector     double[3]   O     Requested reference vector
 *
 *    RETURN VALUE: void
 *
 *******************************************************************/

void GetReference(int ref_type, double t_request, double t_wait,
                  double ref_vector[3])
{
    /*    LOCAL VARIABLES:
     *
     *    Variable       Type      Description
     *    -------------  -------   -----------------------------------
     *    sun            double[3] Earth to Sun vector [km] (from
     *                               c_sunlunp)
     *    moon           double[3] Earth to Moon vector [km] (from
     *                               c_sunlunp)
     *    caldate        double    Epoch time in calendar format
     *    starray        double[6] Epoch time in unpacked array format
     *    mjd            double    Modified Julian Date [days]
     *    gha            double    Greenwich Hour Angle [rad]
     *    a1diff         double    A.1 - UT1 time difference [sec]
     *    numselc        long      Number of secular terms of nutation
     *                               to compute (1- 39, nominally  1)
     *    numterm        long      Number of nonsecular terms of
     *                               nutation to compute (1-106,
     *                               nominally 50)
     *    fdumm          double    Unused return value (from c_unvec3)
     *    ierr           long      Return code from RSL routines
     *    m              double    Variable for #defined MU_E
     *    t              double    Variable for #defined THREEB
     *    eptime         double    Elapsed seconds between epoch time
     *                               and requested time [sec]
     *    dpos           double    Array of dummy position vectors used
     *                               by ephemris read routine
     *    dvel           double    Array of dummy velocity vectors used
     *                               by ephemris read routine
     *    loop_counter   int       Loop counter
     *    i              int       Loop counter
```

```
 *    j              int       Loop counter
 */

double int       sun[3],  moon[3], caldate, starray[6], mjd, gha,
                   a1diff, fdumm;
double int       m, t;
double int       eptime;
long int   numselc, numterm;
long int   ierr = -100;
long int   two = 2;
long int   four = 4;
long int   zero = 0;
int  int   i,j;
char             *mag_path = "/public/libraries/rsl/hpux/emag1990.dat";

static int       loop_counter = 0;
static double int      dpos[3][100], dvel[3][100];

/* Initialize local parameters for RSL routines */

a1diff  = 0.0;
numselc =   1;
numterm =  50;

if (debug_level[RF] > TRACE)
    fprintf(debug_file_handle,"ENTER GetReference\n");

if (debug_level[RF] > INPUT)
{
    fprintf(debug_file_handle,"\tINPUT\n");
    switch (ref_type)
    {
    case 1:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 1) S/C position vector\n");
        break;
    case 2:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 2) S/C velocity vector\n");
        break;
    case 3:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 3) S/C to Sun unit vector\n");
        break;
    case 4:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 4) Earth mag field vector\n");
        break;
    case 5:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 5) Earth mag field unit vector\n");
        break;
```

```
    case 6:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 6) Orbit normal unit vector\n");
        break;
    case 7:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 7) S/C to Moon unit vector\n");
        break;
    case 8:
        fprintf(debug_file_handle,
         "\t\treference type (ref_type = 8) S/C to Earth unit vector\n");
        break;
    }
    fprintf(debug_file_handle,
     "\t\trequest time [sec]      (t_request) = %lf\n",t_request);
    fprintf(debug_file_handle,
     "\t\twait time [sec]            (t_wait) = %lf\n",t_wait);
}

/* Begin Case of reference type */

switch (ref_type)
{

/* Perform case for either s/c position or velocity vector request
 * using the RSL routine c_calpvs */

case 1:
case 2:

    if (debug_level[RF] > INPUT)
    {
        fprintf(debug_file_handle,
         "\t\tlast pos and vel vector time [sec] (t_rv_ref) = %lf\n",
         t_rv_ref);
        fprintf(debug_file_handle,
         "\t\tephemeris read method flag (ephem_method)     = %c\n",
         ephem_method);
    }

    if ((t_request - t_rv_ref) > t_wait)
    {
        eptime = t_request - orbital_t_epoch;

        if (debug_level[RF] > INTERMEDIATE)
        {
            fprintf(debug_file_handle,"\tINTERMEDIATE\n");
            fprintf(debug_file_handle,
             "\t\tRequest time [secs from reference]
             (eptime) = %lf\n",eptime);
        }

        if (ephem_method == 'F')
        {
            if (loop_counter == 0)
```

```
        {
            for (i=0; i<100; i++)

                for (j=0; j<3; j++)
                {
                    dpos[j][i] = 0.0;
                    dvel[j][i] = 0.0;
                }
            loop_counter++;
        }

        c_ephemrd(ephem_file_lu,four,zero,eptime,
           dpos,dvel, s_c_pos,s_c_vel,&ierr);

        if (ierr)
            if (debug_level[RF] > TRACE)
                fprintf(debug_file_handle,
                 "**** Error code from c_ephemrd = %ld\n",ierr);
    }
    else
    {

        m = MU_E;
        t = THREEB;

        c_calpvs(eptime,m,keplerian,t,ttol,maxit, s_c_pos,s_c_vel,&ierr);

        if (ierr)
            if (debug_level[RF] > TRACE)
                fprintf(debug_file_handle,
                 "**** Error code from c_calpvs = %ld\n",ierr);

        if (debug_level[RF] > INTERMEDIATE)
        {
            fprintf(debug_file_handle,
             "\t\tEarth gravitational constant [km^3/sec^2]
             (MU_E) = %lf\n",MU_E);
            fprintf(debug_file_handle,
             "\t\tGrav. constant [Km^2]
             (THREEB) = %lf\n",THREEB);
            fprintf(debug_file_handle,
             "\t\ttolerance of true anomaly [rad]
             (ttol) = %lf\n",ttol);
            fprintf(debug_file_handle,
             "\t\tmax iters of true anomaly (maxit) = %d\n",maxit);
            fprintf(debug_file_handle,
             "\t\ttime of request [sec from epoch]
             (eptime) = %lf\n",eptime);
            fprintf(debug_file_handle,
             "\t\tsemi major axis [km]
             (keplerian[1]) = %lf\n",keplerian[0]);
            fprintf(debug_file_handle,
             "\t\teccentricity     (keplerian[2]) = %lf\n",keplerian[1]);
            fprintf(debug_file_handle,
```

```
                 "\t\tinclination [rad] (keplerian[3]) =
                 %lf\n",keplerian[2]);
                fprintf(debug_file_handle,
                 "\t\tra of asc node [rad] (keplerian[4]) =
                 %lf\n",keplerian[3]);
                fprintf(debug_file_handle,
                 "\t\targ of perigee [rad] (keplerian[5]) =
                 %lf\n",keplerian[4]);
                fprintf(debug_file_handle,
                 "\t\tmean anomaly [rad] (keplerian[6]) =
                 %lf\n",keplerian[5]);
            }
        }

        t_rv_ref = t_request;

        if (debug_level[RF] > INTERMEDIATE)
        {
            fprintf(debug_file_handle,
             "\t\ts/c position vector [km] (s_c_pos) = %lf,%lf,%lf\n",
             s_c_pos[0],s_c_pos[1],s_c_pos[2]);
            fprintf(debug_file_handle,
             "\t\ts/c velocity vector [km] (s_c_vel) = %lf,%lf,%lf\n",
             s_c_vel[0],s_c_vel[1],s_c_vel[2]);
        }
    }

    if (ref_type == 1)
        for (i=0 ; i<3 ; i++)
            ref_vector[i] = s_c_pos[i];
    else
        for (i=0 ; i<3 ; i++)
            ref_vector[i] = s_c_vel[i];
    break;

/* Perform case for s/c to Sun unit vector request using the RSL
 * routine c_sunlunp */

case 3:

    if (debug_level[RF] > INPUT)
        fprintf(debug_file_handle,
         "\t\tlast sun vector time [sec] (t_s_ref) = %lf\n",t_s_ref);

    if ((t_request - t_s_ref) > t_wait)
    {
        caldate = SecsToCalendar(t_request);

        c_packst (caldate,starray);
        c_calmjd (starray,&mjd);

        c_sunlunp(mjd,t_request,sun,moon);
        GetSun   (sun,s_pos);

        t_s_ref = t_request;
```

```
        if (debug_level[RF] > INTERMEDIATE)
        {
            fprintf(debug_file_handle,"\tINTERMEDIATE\n");
            fprintf(debug_file_handle,
             "\t\tModified Julian Date [days] (mjd) = %lf\n", mjd);
            fprintf(debug_file_handle,
             "\t\ttime of request [sec] (use t_request see above) \n");
        }
    }

    for (i=0 ; i<3 ; i++)
        ref_vector[i] = s_pos[i];
    break;

/* Perform case for Earth magnetic field vector or Earth magnetic
 * field unit vector using RSL routines c_emagfld and c_unvec3 */

case 4:
case 5:

    if (debug_level[RF] > INPUT)
        fprintf(debug_file_handle,
         "\t\tlast Earth mag field vector time [sec] (t_b_ref) = %lf\n",
         t_b_ref);
    if ((t_request - t_b_ref) > t_wait)
    {
        caldate = SecsToCalendar(t_request);

        c_packst (caldate,starray);
        c_calmjd (starray,&mjd);

        c_jgrenha(mjd,a1diff,numselc,numterm,&gha,&ierr);

        if (ierr)
            if (debug_level[RF] > TRACE)
                fprintf(debug_file_handle,
                 "**** Error code from c_jgrenha = %ld\n",ierr);

        c_nmlist(1,&two,mag_path,&ierr);

        if (ierr)
            if (debug_level[RF] > TRACE)
                fprintf(debug_file_handle,
                 "**** Error code from c_nmlist = %ld\n",ierr);

        c_emagfl2(two,mjd,t_request,gha,s_c_pos,m_order,mag_field,&ierr);

        if (ierr)
            if (debug_level[RF] > TRACE)
                fprintf(debug_file_handle,
                 "**** Error code from c_emagfl2 = %ld\n",ierr);
        c_unvec3 (mag_field,mag_field_unit,&fdumm);

        t_b_ref = t_request;
```

```
        if (debug_level[RF] > INTERMEDIATE)
        {
            fprintf(debug_file_handle,"\tINTERMEDIATE\n");
            fprintf(debug_file_handle,
             "\t\tModified Julian Date [days] (mjd) = %lf\n", mjd);
            fprintf(debug_file_handle,
             "\t\ttime difference [sec] (a1diff) = %lf\n", a1diff);
            fprintf(debug_file_handle,
             "\t\tnutation number      (numselc) = %d\n", numselc);
            fprintf(debug_file_handle,
             "\t\tnutation number      (numterm) = %d\n", numterm);
            fprintf(debug_file_handle,
             "\t\tGreenwich Hour Angle [rad] (gha) = %lf\n", gha);
            fprintf(debug_file_handle,
             "\t\torder of magnetic field (m_order) = %d\n", m_order);
            fprintf(debug_file_handle,
             "\t\ts/c position vector [km] (s_c_pos) = %lf,%lf,%lf\n",
                s_c_pos[0],s_c_pos[1],s_c_pos[2]);
            fprintf(debug_file_handle,
             "\t\ttime of request [sec] (use t_request see above) \n");
        }
    }

    if (ref_type == 4)
        for (i=0 ; i<3 ; i++)
            ref_vector[i] = mag_field[i];
    else
        for (i=0 ; i<3 ; i++)
            ref_vector[i] = mag_field_unit[i];
    break;

/* Perform case for orbit normal unit vector request */

case 6:

    /* Debug : Intermediate */

    if (debug_level[RF] > INPUT)
        fprintf(debug_file_handle,
            "\t\tlast normal unit vector time [sec] (t_o_ref) = %lf\n",
            t_o_ref);

    if ((t_request - t_o_ref) > t_wait)
    {
        GetOrbitNormal(orbit_normal);
        t_o_ref = t_request;
    }

    for (i=0 ; i<3 ; i++)
        ref_vector[i] = orbit_normal[i];
    break;
```

```
/* Perform case for s/c to Moon unit vector request using the RSL
 * routine c_sunlunp */

case 7:

    if (debug_level[RF] > INPUT)
        fprintf(debug_file_handle,
            "\t\tlast moon vector time [sec] (t_m_ref) = %lf\n",t_m_ref);

    if ((t_request - t_m_ref) > t_wait)
    {
        caldate = SecsToCalendar(t_request);

        c_packst (caldate,starray);
        c_calmjd (starray,&mjd);

        c_sunlunp(mjd,t_request,sun,moon);
        GetMoon  (moon,m_pos);

        t_m_ref = t_request;

        if (debug_level[RF] > INTERMEDIATE)
        {
            fprintf(debug_file_handle,"\tINTERMEDIATE\n");
            fprintf(debug_file_handle,
             "\t\tModified Julian Date [days] (mjd) = %lf\n", mjd);
            fprintf(debug_file_handle,
             "\t\ttime of request [sec] (use t_request see above) \n");
        }
    }

    for (i=0 ; i<3 ; i++)
        ref_vector[i] = m_pos[i];
    break;

/* Perform case for s/c to Earth unit vector request */

case 8:

    if (debug_level[RF] > INPUT)
        fprintf(debug_file_handle,
         "\t\tlast Earth vector time [sec] (t_e_ref) = %lf\n",t_e_ref);

    if ((t_request - t_e_ref) > t_wait)
    {
        GetEarth(e_pos);

        t_e_ref = t_request;
    }

    for (i=0 ; i<3 ; i++)
        ref_vector[i] = e_pos[i];
    break;
```

```
    } /* end switch */

    if (debug_level[RF] > OUTPUT)
    {
        fprintf(debug_file_handle,"\tOUTPUT\n");
        fprintf(debug_file_handle,
          "\t\trequested reference vector (ref_vector) = %lf,%lf,%lf\n",
          ref_vector[0],ref_vector[1],ref_vector[2]);
    }

    if (debug_level[RF] > TRACE)
        fprintf(debug_file_handle,"EXIT GetReference\n\n");

    return;

} /* end */
```

9.3 Include File: HD_reference.h

```
/*************************************************************************
 *
 *    FILE NAME: HD_reference.h
 *
 *
 *    PURPOSE: Defines all reference data variables.
 *
 *
 *    GLOBAL VARIABLES:
 *
 *    Variables             Type            Description
 *    -------------         --------        ----------------------------
 *    e_pos                 double[3]       S/C to Earth unit vector
 *
 *    ephem_file_lu         long            FORTRAN logical unit number
 *                                            for the ephemeris file
 *
 *    ephem_file_name       char[30]        Name of the ephemeris file
 *
 *    ephem_method          char            Method for computing
 *                                            ephemeris information:
 *                                            F = Use ephemeris file
 *                                            A = Compute analytically
 *                                                using Keplerian
 *                                                elements
 *
 *    keplerian             double[6]       Keplerian orbital elements
 *                                            at the epoch time
 *                                            (orbital_t_epoch):
 *                                            [1] Semimajor axis [km]
 *                                            [2] Eccentricity
 *                                            [3] Inclination [rad]
 *                                            [4] Right ascension of
 *                                                the ascending node
 *                                                [rad]
 *                                            [5] Argument of perigee
 *                                                [rad]
 *                                            [6] Mean anomaly [rad]
 *
 *    m_order               long            Order of magnetic field
 *
 *    m_pos                 double[3]       S/C to Moon unit vector
 *
 *    mag_field             double[3]       Earth magnetic field vector
 *                                            [mG]
 *
 *    mag_field_unit        double[3]       Earth magnetic field unit
 *                                            vector
```

```
 *
 *    maxit                  long            Maximum number of
 *                                             iterations to converge
 *                                             the true anomaly
 *
 *    MU_E                   double          Earth gravitational
 *                                               constant [km^3/sec^2]
 *
 *    NUMPTS                 int             Number of points used by
 *                                             the EPHEMRD interpolator
 *
 *    orbit_normal           double[3]       Orbit normal unit vector
 *
 *    orbital_t_epoch        double          Base epoch time of the
 *                                             orbital elements [sec]
 *
 *    s_c_pos                double[3]       S/C position vector [km]
 *
 *    s_c_vel                double[3]       S/C velocity vector
 *                                             [km/sec]
 *
 *    s_pos                  double[3]       S/C to Sun unit vector
 *
 *    t_b_ref                double          Time of last calculated
 *                                             Earth magnetic field
 *                                             vector [sec]
 *
 *    t_e_ref                double          Time of last calculated s/c
 *                                             to Earth unit vector
 *                                             [sec]
 *
 *    t_m_ref                double          Time of last calculated s/c
 *                                             to Moon unit vector [sec]
 *
 *    t_o_ref                double          Time of last calculated
 *                                             orbit normal unit vector
 *                                             [sec]
 *
 *    t_rv_ref               double          Time of last calculated s/c
 *                                             position and velocity
 *                                             vectors[sec]
 *
 *    t_s_ref                double          Time of last calculated s/c
 *                                             to Sun unit vector [sec]
 *
 *    THREEB                 double          Gravitational constant of
 *                                             perturbations [Km^2]
 *
 *    ttol                   double          Tolerance in the
 *                                             calculations of the true
 *                                             anomaly [rad]
 *
```

```
 *
 *    DEVELOPMENT HISTORY:
 *
 *    Date      Author        Change  Release   Description of Change
 *                            ID
 *    --------  ------------  ------  -------   ------------------------
 *    09-23-93  J. Programmer              1     Prolog and PDL
 *    10-07-93  J. Programmer              1     Controlled
 *    12-02-93  J. Programmer              1     Integrated new RSL
                                                   routines
 *    12-17-93  J. Programmer              2     Added maxit and ttol;
                                                   added MU_E and THREEB
 *                                                 as #defines
 *    04-06-94  J. Programmer     27       3     Corrected the THREEB
                                                   value
 *    05-10-94  J. Programmer              3     Added ephemeris read
                                                   capability
 *
 *

*********************************************************************/

    #define  MU_E       398600.8
    #define  THREEB     66042.0
    #define  NUMPTS     4

    extern long         ephem_file_lu;
    extern double       e_pos[3];
    extern char         ephem_file_name[30];
    extern char         ephem_method;
    extern double       keplerian[6];
    extern long         m_order;
    extern double       m_pos[3];
    extern double       mag_field[3];
    extern double       mag_field_unit[3];
    extern long         maxit;
    extern double       orbit_normal[3];
    extern double       orbital_t_epoch;
    extern double       s_c_pos[3];
    extern double       s_c_vel[3];
    extern double       s_pos[3];
    extern double       t_b_ref;
    extern double       t_e_ref;
    extern double       t_m_ref;
    extern double       t_o_ref;
    extern double       t_rv_ref;
    extern double       t_s_ref;
    extern double       ttol;
```

BIBLIOGRAPHY

Atterbury, M., *ESA Style Guide for 'C' Coding*, Expert Solutions Australia Pty. Ltd., Melbourne, Australia (1991)

Computer Sciences Corporation, *SEAS System Development Methodology (Release 2)* (1989)

Indian Hill C Style and Coding Standards, Bell Telephone Laboratories, Technical Memorandum 78-5221 (1978)

Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978)

Minow, M., *A C Style Sheet*, Digital Equipment Corporation, Maynard, Massachusetts

Oualline, S., *C Elements of Style*, M&T Publishing, Inc., San Mateo, California (1992)

Wood R., and Edwards, E., *Programmer's Handbook for Flight Dynamics Software Development*, SEL-86-001 (1986)

INDEX

V

W