

CONTENTS

Section		Page
	HRF CODING STYLE GUIDE	
C1.0	<u>INTRODUCTION</u>	C-1
C1.1	PURPOSE	C-1
C1.2	SCOPE	C-1
C2.0	<u>PROGRAM STRUCTURE</u>	C-1
C2.1	COMPUTER SOFTWARE CONFIGURATION ITEM (CSCI)	C-1
C2.2	COMPUTER SOFTWARE UNIT (CSU)	C-1
C2.3	LIBRARIES	C-1
C2.4	SOURCE FILES	C-2
C2.5	HEADER FILES	C-2
C2.6	PROLOGUES	C-2
C3.0	<u>GENERAL GUIDELINES</u>	C-3
C3.1	DESIGN CONSIDERATIONS	C-3
C3.2	IMPLEMENTATION CONSIDERATIONS	C-3
C3.3	PORTABILITY RULES	C-4
C3.4	EXPRESSIONS	C-5
C3.5	IDENTIFIERS	C-6
C3.5.1	<u>Global Data</u>	C-6
C3.5.2	<u>Local Data</u>	C-6
C3.5.3	<u>Input/Output Data</u>	C-6
C3.5.4	<u>Pointers</u>	C-7
C3.6	DECLARATIONS	C-7
C3.7	FUNCTIONS	C-8
C3.7.1	<u>Interfaces</u>	C-9
C3.7.2	<u>Side Effects</u>	C-9
C3.8	HEADER FILES	C-9
C3.8.1	<u>Type Definitions</u>	C-10
C3.8.2	<u>Constants</u>	C-10
C3.8.3	<u>Macros</u>	C-11
C3.8.4	<u>Function Prototype Statements</u>	C-11
C4.0	<u>NAMING CONVENTIONS</u>	C-12
C4.1	FILE NAMES	C-12
C4.1.1	<u>CSCI File Names</u>	C-12
C4.1.2	<u>CSU File Names</u>	C-12
C4.1.3	<u>Library File Names</u>	C-12
C4.2	IDENTIFIER NAMES	C-13

CONTENTS (Cont'd)

SECTION		PAG
C5.0	<u>FORMATS</u>	C-14
C5.1	FILE FORMATS	C-14
C5.1.1	<u>CSCI Source File Formats</u>	C-14
C5.1.2	<u>CSU Source File Format</u>	C-14
C5.1.3	<u>Library Source File Format</u>	C-15
C5.1.4	<u>Header File Format</u>	C-15
C5.2	COMMENT FORMATS	C-15
C5.3	EXPRESSION FORMATS	C-16
C5.3.1	<u>The <i>if...else</i> Statement</u>	C-17
C5.3.2	<u>The <i>switch</i> Statement</u>	C-18
C5.3.3	<u>The <i>while</i> Statement</u>	C-18
C5.3.4	<u>The <i>do</i> Statement</u>	C-19
C5.3.5	<u>The <i>for</i> Statement</u>	C-19
C6.0	<u>C CODE SAMPLES</u>	C-19
C6.1	CSCI HEADER FILE	C-20
C6.2	CSCI SOURCE FILE	C-21
C6.3	CSU SOURCE FILE 1	C-22
C6.4	CSU SOURCE FILE 2	C-23
C6.5	LIBRARY HEADER FILE	C-26
C6.6	LIBRARY SOURCE FILE	C-27

APPENDIX C
HRF CODING STYLE GUIDE

Typographical Conventions

The font style conventions illustrated in Table C-1 are used throughout this document to identify special text.

TABLE C-1 FONT STYLES FOR SPECIAL TEXT
(OTHER THAN HEADINGS)

Text	Text Style	Example
C Reserved Words or Standard Files	<i>Italics</i>	<i>main, extern, errno.h</i>
Examples	<i>Bold Italics</i>	<i>My_File_Name</i>

C1.0 INTRODUCTION

C1.1 PURPOSE

This document is intended to be a coding style guideline for C code written for Human Research Facility (HRF) systems. It is not intended to be a rule book. Rather, it is intended to provide guidelines to improve code readability and portability. Many of these guidelines may be superseded by unique requirements imposed by a Commercial Off the Shelf (COTS) product or by the need to improve software performance.

C1.2 SCOPE

This document defines the program structure for all C coded Computer Software Configuration Items (CSCIs) and libraries within the HRF. It also specifies general guidelines that should be used to ensure sound design, implementation, and overall clarity and maintainability of the code. General naming conventions are specified, as are guidelines for commenting the code. A set of formats is provided as examples to help ensure a consistent look and feel among code written by a wide range of developers. The concepts presented shall be applied to other programming languages used for HRF software development as appropriate. Finally, Section C6.0, C Code Samples, contains some C code examples that incorporate the guidelines specified throughout the text of this document.

C2.0 PROGRAM STRUCTURE

C2.1 COMPUTER SOFTWARE CONFIGURATION ITEM (CSCI)

A CSCI is an aggregation of software or firmware that satisfies an end use function and is designated for configuration management. In general, each CSCI has a *main* routine that ties together all of the component routines into a cohesive functional unit. Occasionally, a CSCI may consist of multiple *main* routines. These cases are clearly identified and an explanation for the multiple *main* routines is provided in the corresponding software design document.

C2.2 COMPUTER SOFTWARE UNIT (CSU)

A source file containing one or more functions used to build a CSCI. If a CSU consists of more than one function, the functions should work together to execute part of the CSCI design.

C2.3 LIBRARIES

A library is a special type of CSU that is a collection of functions that are used by one or more CSCIs and linked with the CSCI *main* function at compile time. A library by itself serves as a repository of function object code. This limits the amount of software that must be recompiled when a CSCI component is changed.

C2.4 SOURCE FILES

Source files contain all of the C code, data declarations and functions for a CSCI. The contents of a source file may represent a CSCI, a CSU or a library.

C2.5 HEADER FILES

Header files contain all of the constants, typedefs, structures, unions, enumeration types, macros and function prototypes required for the successful compilation of the CSCI or library.

C2.6 PROLOGUES

The following prologue shall be included at the beginning of each source or header file:

```
/*  
*  
* FILENAME:    filename.type  
*  
* DESCRIPTION: Description  
*  
* CSCI IDENTIFIER:  HRF-xyyz  
*  
*****  
/  
/
```

Where:

filename is the name of the file and indicates the purpose of the file contents.
type represents the type of file, e.g. .c for C source files and .h for header files.
Description is a brief summary of the file contents. If the content is a main function, the operation of the CSCI shall be included. For individual functions, function utilization information shall be provided. For function collections or libraries, rationale for grouping the functions shall be provided.

HRF-xyyz is the unique identifier associate with a CSCI, see the Human Research Facility Software Configuration management Plan and Procedure. If the file is associated with multiple CSCIs, all of the associated CSCI unique identifiers shall be listed.

See Naming Conventions for more information.

If a file contains more than one function, the following prologue shall precede each function in the file:

```
/******  
*  
*  
* FUNCTION:    FunctionName  
*  
* DESCRIPTION: Function description including usage.  
*  
*****  
/
```

C3.0 GENERAL GUIDELINES

C3.1 DESIGN CONSIDERATIONS

The set of design techniques collectively described as "top down" shall be applied to a CSCI as detailed below:

- Partitioning should be used so that interdependencies between functions are minimized (i.e., loosely coupled). Each function should have a well-defined purpose or algorithm.
- Hierarchical relationships should be enforced so that functions at one level can only invoke functions at a lower level. Typically, upper-level functions contain primarily decision and control logic, while the lower-level functions perform specific application tasks.

C3.2 IMPLEMENTATION CONSIDERATIONS

A C compiler should be set at the most stringent level of error checking possible for that compiler.

The makefile facility should be used to control compile sequences.

Executing code shall not dynamically modify itself or any other executable code. Only the data associated with a CSCI may be modified.

When working on systems that support signal processing, the use of signal handlers is strongly encouraged. This enables developers to produce meaningful error messages or avoid error messages altogether by using the appropriate user-defined signal handlers *signal* and *perror* and the include file *errno.h*.

C3.3 PORTABILITY RULES

A portable function is platform or machine independent. Any environment-specific and machine-specific code should be isolated and localized as much as possible. It should also be delimited with appropriate compiler directives.

Due to the C compiler aligning data types at certain byte boundaries, order members in a structure by data type length, longest first, whenever possible (i.e. 8-byte variables first, then 4-byte variables, 2-byte variables, etc.). This may not be possible when interfacing with external functions or hardware.

System-defined symbolic constants such as *NULL*, *EOF*, *"/f"*, etc., should be used. This usage promotes readability and portability between systems.

Avoid hard-coding character strings into the code. Instead, use *#define* constructs or read the strings from a database at execution time.

The following data types should be used because their length and format are standard across most modern architectures:

<u>Description</u>	<u>Data Type</u>	<u>Length (Bytes)</u>
ASCII Characters:	<i>char</i>	1
Signed Integer Numbers:	<i>char</i>	1
	<i>short</i>	2
	<i>int</i>	4
Unsigned Integer Numbers:	<i>unsigned char</i>	1
	<i>unsigned short</i>	2
	<i>unsigned long</i>	4
Floating Point Numbers:	<i>float</i>	4
	<i>double</i>	8

The absolute path name of an *#include* statement should not be specified in the code. Instead, the directory of the include file (or reference directory for *#include* statements with relative path names) should be included as an argument on the C compiler's command line in the CSCI makefile. For example:

```
cc -I- -I- -I/home/project_name file_a.c -lm      /* UNIX format */
```

C3.4 EXPRESSIONS

Complex or compound expressions in which the order of evaluation is important shall be avoided. For example, in the expression "*Array*[*i*++] = *Value*;", *Value* will be assigned to a different element in *Array* depending on whether *i* is incremented before or after the assignment takes place. This expression should be written as two separate expressions:

```
Array[i] = Value;  
i++;          /* if i is to be incremented after the variable assignment */  
or  
i++;  
Array[i] = Value;      /* if i is to be incremented before the variable assignment.  
*/
```

To avoid Side Effects, "++" and "--" shall not be used within another expression. Rather, when these operators are needed, they shall be used on their own line. A *for* loop increment operation is the only exception.

Braces "{}" shall be used in conjunction with indentation to delineate complex logic. The braces should be vertically aligned. For example:

```
for (i=0; i<5; ++i)  
    {  
        }  
    }
```

Parentheses "(" should be used as necessary to enforce efficient expression evaluation.

Mixed-mode operations (i.e. multiplying integer and real identifiers) should be avoided. If operations need to be performed on mixed types, an explicit type cast should be used.

Complicated compound negative Boolean expressions should be avoided.

Whenever possible, Boolean expressions should be evaluated as True=1 and False=0.

The use of the conditional expression operator "?" is discouraged.

The *sizeof* function should be used to determine the amount of memory a data item uses, because the actual size of a data item is often not what is expected. This also maintains the portability of code.

The *goto* statement shall not be used.

Use of the *continue* statement is discouraged.

The use of *break* to exit a looping construct is discouraged.

The *return* statement, when used, should be the last statement within a function. The only exception is for implementing error handling. Avoid putting *return* statements inside *if* or loop constructs where the *return* statement could be skipped.

Each CSCI should have only one *exit* statement, which should be the last executable statement in the *main* routine. The only exception is for implementing error handling.

C3.5 IDENTIFIERS

C3.5.1 Global Data

In general, the use of global data should be avoided in C code. In the cases where global data is needed to meet performance or data visibility requirements, the following guidelines should be used:

Combine all of a CSCI's global data into a data structure in the CSCI's header file. Declare a variable, using the data structure defined in step 1, in the CSCI's *main*. Declare the variable as *extern*, using the data structure defined in step 1 and the identifier name from step 2, in the other CSU source files associated the CSCI.

An algorithm that prevents simultaneous access by two or more CSCIs should protect identifiers that are shared between multiple CSCIs.

C3.5.2 Local Data

Temporary (local) memory should be used whenever possible in order to improve processing speed.

C3.5.3 Input/Output Data

Precautions shall be made to ensure consistent input data (i.e. the input data shall not be allowed to change while operations are being made on that data).

Some input data words from external subsystems have unused bits that by convention are not set and are referred to as "don't care" bits. However, the code should treat these bits as if they were variable. Usually, this means masking out the "don't care" bits before they are used internally.

Multiple stores into a global output identifier should be avoided, whenever possible, to prevent intermediate values from being accessed by another function. Instead, store intermediate values in local identifiers where they are not available to other functions. Before exiting the function, copy the local value to the global location.

C3.5.4 Pointers

Dynamically allocated identifiers should only be used when necessary. In all cases where an identifier is dynamically allocated, its memory shall be explicitly freed before the CSCI terminates.

Pointer data types should not be mixed when assigning one pointer identifier to another.

When declaring a pointer, it should be initialized to a value of *NULL* or to the intended object. This makes it easy to determine whether or not a pointer has been assigned.

Pointers should be initialized or verified to ensure the pointer is not *NULL* prior to use.

Manipulation of "pointers to functions" should be limited. This is an extremely powerful practice, but when used improperly, it can cause system failures that can be extremely difficult to resolve. If "pointers to functions" are used, strict type adherence shall be maintained. The same consideration shall be made for "pointers to pointers" or "handles".

C3.6 DECLARATIONS

The scope of identifiers and functions shall be as local as possible to maintain the integrity of the system and to discourage outside access to a function's internal data.

All variable to be used in a function shall be declared at the beginning of the function. This avoids variable scope issues that can cause confusion when variable declarations are embedded in control constructs (e.g. if, while, etc.).

Statements that attempt to define a data type and declare a variable within the same statement should be avoided. Instead, define the data type in a header file, then declare the variable in a source file using the new data type.

Multiple identifier declarations (more than one user identifier in an identifier list) should be avoided. Exceptions can be made for insignificant identifiers, such as loop counters.

All identifiers and functions shall be explicitly declared (e.g. *short*, *int*, *double*). Avoid using "default" function declarations.

Identifiers shall be of declared types defined in a header file or the standard C types (e.g. *double*, *int*, *short*, *char*).

[S]tatic identifiers should be used only when necessary to maintain the value of an identifier from invocation to invocation of a code block.

The use of *register* identifiers should be avoided, except in cases where improved speed is absolutely necessary.

C3.7 FUNCTIONS

Each function shall have a prologue. If a source file contains only one function, however, the standard source file prologue will suffice. See Section C2.6, Prologues, for more information.

The behavior of each function shall adhere to the following considerations:

- Strong internal cohesion — A function has strong internal cohesion if it is designed to implement a single function or a set of closely related operations. Poor internal cohesion is demonstrated by a function that is designed to implement two or more unrelated operations (i.e. to calculate the time and find the square root of a number).
- Loose external coupling — A function has loose external coupling if it is designed so that its functioning is not affected (as much as possible) by changes made internally to other functions.
- Ease of understanding and readability.
- Ease of testing
- Portability
- Maintainability
- Abstraction - Functions should be designed so that a programmer can use a function by passing the appropriate arguments, but the programmer would not need to know the implementation details of the function.

In instances in which the order of evaluation of functions is important, the required order of processing shall be stated explicitly in comment fields within the code.

A function prototype statement shall exist in a header file for each function to be used. Functions contained in the Standard C Libraries are exempt.

It is a good practice to keep functions as short as possible (i.e. more functions with less executable statements per function) in order to improve modularity. This concept is encouraged in most cases, especially for library functions. Be careful, however, because the incurred overhead associated with an increased number of function calls can degrade execution speed.

With the exception of error handling, the set of design techniques collectively described as "top down" shall be applied for each function as detailed below:

- Each function shall have one entry and one exit.
- Each function shall return to its calling function.
- The logic of each function shall be simplified as much as possible, without adversely affecting the ability of the function to meet its requirements.

In general, recursion should not be used, since it may decrease execution efficiency as a result of overhead.

C3.7.1 Interfaces

Interfaces should be kept as simple as possible. The number of inputs and outputs should be minimal.

Functions that do not return a value to the calling routine should be declared as *void* functions.

All functions shall be explicitly declared (e.g. *short*, *int*, *double*). Avoid using "default" function declarations.

Typically, interfaces will be as explicit (i.e. arguments passed through a parameter list) and as simple as possible. Global data should be avoided, when possible.

Structure and *union* parameters should be passed by reference to user-defined functions (i.e. pass the address rather than the contents of the structure/union).

C3.7.2 Side Effects

Side effects are defined as changes to identifiers outside the scope of the function. An example is modification of the function's actual parameters during the call of a function rather than inside the function (e.g. passing *i++* as a parameter). Side effects should be avoided, because their impacts are typically not obvious and are difficult to test, and they cause a function to be unnecessarily tightly coupled.

If side effects are necessary, then a comment describing all of the side effects of the function shall be included in the prologue and design document.

C3.8 HEADER FILES

Header files contain all constants, *typedefs*, macros, and function prototype statements for a CSCI or library. Source files only contain identifier declarations of these types.

When possible, put the interface constants, types, macros, and function prototypes in a separate file from the internal constants, types, macros, and function prototypes.

Each header file shall contain a prologue as specified in Section C2.6, Prologues.

Data shall not be declared in header files, since each inclusion of the file creates a copy of the data.

Header files shall not contain any executable code.

Including other header files from within a header file is discouraged in all cases. The only exception is if a subsequent statement within the header file absolutely requires something contained within another header file. These instances should be avoided if at all possible, however.

Header files should be protected from multiple inclusions by using the following construct:

```
#ifndef FILE_NAME_H

/* header file contents */

#define FILE_NAME_H
#endif
```

C3.8.1 Type Definitions

A *typedef* should be used to define new types for all structures, unions, and enumeration types.

A *typedef* should be used if several identifiers are going to be used for a common purpose and declared the same way (e.g. *typedef int SEMAPHORE_TYPE*).

[T]*typedef* statements that modify other user-defined *typedefs* (e.g. *typedef MEM_TYPE *MEM_TYPE_PTR*) should be avoided, because the true nature of the data type becomes difficult to track.

The use of the *union* declaration is strongly discouraged.

C3.8.2 Constants

The use of global constant declarations (*#define*), which symbolically name constants that occur frequently in one or more function blocks or programs or that have a recognizable significance, is strongly encouraged.

Constants should be used to ensure data consistency and integrity and to improve clarity. They should not be used simply to reduce the amount of typing within the

source code (e.g., *#define FNAME Global->file_name* is not recommended). Overuse of constants can make the source code difficult to read because of frequent references to the header file contents.

The use of the *const* type qualifier is similar to *#define* except that *const* variables can have a local scope when they are declared within a function.

C3.8.3 Macros

Macros should be used to enhance readability and maintenance, to save the overhead of a function call, or to reduce the complexity of an expression. They should not be overused (e.g., *#define GET_FILE_NAME(a) a.file_name*), because frequent references to the header file contents can make the source code difficult to read.

Each macro shall be preceded by a function prologue.

If a macro contains a control construct (e.g., *if...else*, *for*, *while*, *switch*, etc.), the entire construct shall be defined in the macro definition. A starting and ending macro combination is not acceptable.

Define a macro whenever the value is determined at run-time, otherwise create a constant if the value can be determined at compile time.

Macros should be defined to look like function calls. If a more robust function is required of the macro, it can be replaced by a function without changing any of the calling source code.

C3.8.4 Function Prototype Statements

As previously stated, function prototypes should be placed in the header file and should be formatted as follows:

```
function_type function_name(parameter_1_type      in_1,  
                             parameter_2_type      in_out_1,  
                             parameter_3_type      in_out_2,  
                             parameter_4_type      out_1,
```

```
                             parameter_n_type      direction_n);
```

Where *in_1* is an input to the function (not modified), *in_out_1* and *in_out_2* are inputs to the function that are modified by the function, and *out_1* is a parameter that is only passed to the function to be modified.

C4.0 NAMING CONVENTIONS

C4.1 FILE NAMES

C4.1.1 CSCI File Names

Following are naming conventions to be used for the CSCI *abcd*:

Executable: *abcd**
Source File(s): *abcd_*.c*
Header File: *abcd.h***

* No convention, but developers are encouraged to include *abcd* somewhere in the filename, preferably at the beginning (e.g., *abcd_valve.c*). If there is only one source file associated with a CSCI, the recommended name is *abcd.c*.

** Ideally, each CSCI should have a single header file. If multiple header files are required for the sake of clarity, each header file shall include *abcd* somewhere in the filename, preferably at the beginning (e.g., *abcd_macro.h*).

C4.1.2 CSU File Names

Following are naming conventions to be used for the CSU *lmno*:

Executable: *lmno**
Source File(s): *lmno_*.c*
Header File: *lmno.h***

* No convention, but developers are encouraged to include *lmno* somewhere in the filename, preferably at the beginning (e.g., *lmno_valve.c*). If there is only one source file associated with a CSU, the recommended name is *lmno.c*.

** Ideally, every CSU should be a component of one CSCI or become part of a library. In the ideal situations, the CSCI or library header file should be used. If a CSU header file is required for the sake of clarity, each header file shall be called *lmno.h*.

C4.1.3 Library File Names

Following are naming conventions to be used for the library *wxyz*:

Library: *lib_wxyz.a**
Source File(s): *lib_wxyz_*.c*
Header File: *lib_wxyz.h***

* No convention, but developers are encouraged to include *wxyz* somewhere in the filename, preferably at the beginning (e.g., *lib_wxyz_math.c*). If there is only one library source file, the recommended name is *lib_wxyz.c*.

** Ideally, each library should have a single header file. If multiple header files are required for the sake of clarity, each header file shall include *wxyz* somewhere in the filename, preferably at the beginning (e.g., *wxyz_macro.h*).

C4.2 IDENTIFIER NAMES

Descriptive names for user identifiers based on functionality and/or use shall always be used.

Identifier names, with the exception of loop counters and indices (e.g., "i", "j"), shall be longer than two characters. Identifier names should use underscores as word breaks. Two or three words make variable names long enough to be easily understood. Note that in ANSI/ISO C, the first 31 characters in a variable name are significant and therefore must be unique.

Identifier names shall not begin with an underscore (except for identifiers declared in macros).

Identifiers should not differ only by the presence/absence of underline characters, the interchange of capital letters with lower-case letters, or the interchange of symbols that look similar (e.g. "O" and "0", "l" and "1", "S" and "5", etc.).

Identifier names for pointer variables shall be prefixed by "p_", pointers to pointers shall be prefixed by "pp_", etc.

Function identifiers shall be in mixed case. Each separated word shall begin with an uppercase letter followed by all lowercase letters to improve readability. Words may be additionally separated using underscores. Each function identifier should contain the name of the CSU associated with the function as the last characters of the identifier. For example, a function in the Linked List CSU that inserts a cell into a linked list might be called *Insert_Cell_Linked_List*.

Global object identifiers shall begin with an uppercase letter followed by all lowercase letters, while local object identifiers shall be all lowercase. Underscores shall be used to separate words within the global object identifier to enhance readability. For example, a global object containing the Workstation task list might be called *Workstation_task_list*, while a local version of the same object would be *local_task_list*.

Constant, *typedef* and macro names shall be in all uppercase with underscores used to separate words.

C5.0 FORMATS

C5.1 FILE FORMATS

C5.1.1 CSCI Source File Format

A CSCI source file is structured as follows:

1. Prologue — Comment information used to track changes, etc.
2. Include Statements — Header files, including system header files (enclosed by $\langle \rangle$), library header files (enclosed by `"`), and the CSCI header file (also enclosed by `"`).
3. Global Data Declarations — Declarations for all data global to the CSCI, including shared memory data. Note that this data is either a standard C data type (*float*, *short*, *char*, etc.) or a type that is defined in one of the above header files.
4. The *main* function of the CSCI, which calls functions from other source files that comprise the CSCI.

C5.1.2 CSU Source File Format

All CSU source files, with the exception of the CSCI source file, are structured as follows:

1. Prologue — Comment information used to track changes, etc.
2. Include Statements — Header files, including system header files (enclosed by $\langle \rangle$), library header files (enclosed by `"`), and the CSCI header file (also enclosed by `"`).
3. Global Data Declarations — Declarations for all data global to the CSCI, including shared memory data. Note that this data is either a standard C data type (*float*, *short*, *char*, etc.) or a type that is defined in one of the above header files, and is preceded by the word *extern*, which specifies that the variable has been formally declared elsewhere (in the CSCI Source File).
4. One or more functions that perform operations required for the CSCI.

Note that this format differs from the format of a CSCI source file in two ways:

1. This file contains one or more functions, as opposed to a *main*, which is a CSCI master sequencing routine; and
2. All global data declared in this file are defined as *extern*, because they are formally declared in the CSCI source file.

C5.1.3 Library Source File Format

All library source files are structured as follows:

1. Prologue — Comment information used to track changes, etc.
2. Include Statements — Header files, including system header files (enclosed by `<>`) and the library's header file (enclosed by `"`).
3. One or more functions that perform a task provided by the library.

Note that this format differs from the format for CSU source files in two ways:

1. There are no CSCI header files included within a library source file; and
2. There are no global data declarations within a library — all library functions are self-contained.

C5.1.4 Header File Format

All header files are structured as follows:

1. Prologue — Comment information used to track changes, etc.
2. Constants — *#define* statements used to equate a constant value with a string.
3. Typedefs, Structures, Unions and Enumeration types — C constructs used to improve code readability and maintainability.
4. Macros — *#define* statements used to substitute a common series of statements throughout the CSCI/library with a simple string and accompanying arguments.
5. Function Prototypes — Prototypes, including argument list specifications, for all functions within the CSCI/library.

C5.2 COMMENT FORMATS

The purpose of comments is to serve as an aid for someone unfamiliar with the code. Comment WHY something is happening not just WHAT is happening.

Every functional block of code shall be explained by a comment.

Block comments should be indented to the same column as the following line of code.

Comment style shall be consistent throughout a file. Comment style should be consistent throughout all files associated with a CSCI.

Comments that encompass multiple lines should begin the first line with a `/*`. Each subsequent line should begin with a `*`. The last line should end with a `*/`. In C++, each comment line should begin with `///`.

Constants, structures, unions, enumeration types and macros should have comments to describe their overall use.

No comments should be embedded within simple executable statements.

Blank lines shall be used as necessary to delineate comments.

Personal pronouns in comments should be avoided.

A formal writing style (third person) should be used with normal capitalization practices (i.e. not entirely uppercase characters).

Comments shall not reference specific sections, paragraphs, figures, tables, etc. of documentation that could later change, thereby requiring a change to the code.

Comments that document actions outside of a function should be avoided. This ensures that the comments within a function only need to be changed when changes are made to the function itself. Comments pertaining to internal workings of invoked functions may be used if necessary, but are discouraged.

C5.3 EXPRESSION FORMATS

With the exception of complex mathematical expressions, which should be kept to one line whenever possible, each line of code should contain a maximum of 80 characters.

There should be a maximum of one statement per line.

Blank lines shall be used to enhance readability. For example, each block of code that performs a function should be separated by a blank line.

Function definitions, as well as the function's opening and closing braces, should begin in column one.

All nesting should be indented at least three spaces from the enclosing braces or the parent structure.

Braces should reside on separate lines by themselves and should be indented to the same column as the *if*, *while* or *for* statement with which they are associated.

Continuation of *if*, *while*, or *for* conditions should line up with the starting column of the previous line condition.

The connecting logical operator shall go on the same line as *if*, *while* or *for*.

No blanks should be encoded around unary operators such as "*abs*(-10)" or "*not*(*Boolean_Identifier*)".

A blank space should be encoded on both sides of all binary operators. For example, "*X = X + 1*", not "*X=X+1*".

C5.3.1 The *if...else* Statement

The format of the *if* statement should follow one of the following two forms. Format 2 is preferred. The following examples, demonstrating two coding formats for an *if* statement, are functionally identical. *Condition_1* and *Condition_2* are Boolean expressions (complex or simple).

Format 1:

```
if (Condition_1)
{
    /* Comment about statement(s) */
    sequence_of_statements . . .
}
else
{
    if (Condition_2)
    {
        /* Comment about statement(s) */
        sequence_of_statements . . .
    }
    else
    {
        /* Comment about statement(s) */
        sequence_of_statements . . .
    }
}
```

Format 2:

```
if (Condition_1)
{
    /* Comment about statement(s) */
    sequence_of_statements . . .
}
else if (Condition_2)
{
    /* Comment about statement(s) */
    sequence_of_statements . . .
}
else
```

```

{
  /* Comment about statement(s) */
  sequence_of_statements . . .
}

```

The two coding formats are used to increase understandability. Format 1 uses nested *if* statements and avoids a *switch*-like structure. Format 2 shows mutually exclusive conditions. It is structured much like a *switch* statement that always deals with mutually exclusive conditions.

A *NULL* choice shall be coded by deleting the alternative *else* completely. Using the second example above, if ***Condition_1*** and ***Condition_2*** cover all possibilities, the final *else* clause would be deleted.

C5.3.2 The *switch* Statement

The coding format for a *switch* statement should be as follows:

```

/* Comment about statement */
switch (condition)
{
  /* Case 1 comment */
  case first_condition:
    /* Comment about statement(s) */
    sequence_of_statement(s) . . .
    break;

  /* Case 2 comment */
  case second_condition:
    /* Comment about statement(s) */
    sequence_of_statement(s) . . .
    break;

  /* Case default comment */
  default:
    /* Comment about statement(s) */
    sequence_of_statement(s) . . .
    break;
}

```

C5.3.3 The *while* Statement

The format of the *while* statement should follow one of the following two forms:

Format 1: A wait loop (sleep until interrupted)

```
while (condition)
{
  /* Do nothing until interrupted */;
}
```

Format 2: A while condition loop (loop while a condition is met)

```
while (condition)
{
  /* Comment about statement(s) */
  sequence_of_statements . . .
}
```

C5.3.4 The *do* Statement

The coding format for a *do* statement should be a repeat until condition loop (loop until a condition is met) as follows:

```
do
{
  /* Comment about statement(s) */
  sequence_of_statements . . .
} while (condition);
```

C5.3.5 The *for* Statement

The coding format for a *for* loop should be as follows:

```
for (initial_expression; condition; loop_expression)
{
  /* Comment about statement(s) */
  sequence_of_statements . . .
}
```

C6.0 C CODE SAMPLES

The following sections contain examples of C source and header files. The example illustrates a simple CSCI, called ***Force***, which is comprised of the following files:

```
force.h      (CSCI header file)
force.c     (CSCI source file)
init_force.c (CSU source file 1)
proc_force.c (CSU source file 2)
```

Additionally, the *Force* CSCI accesses the function *Square_Root*, which is contained in the library *Lib_HRF_Math*. The *Lib_HRF_Math* library is comprised of the following files:

lib_hrf_math.h (library header file)
lib_hrf_math.c (library source file)

Note that the header file *lib_hrf_math.h* contains the macro *DMXM*, which isn't used by the *Force CSCI*, but is used by another CSCI that accesses this library.

C6.1 CSCI HEADER FILE

```

#ifndef FORCE_H
/*****
**
* FILENAME:    force.h
*
* DESCRIPTION:  This header file contains constants, typedefs,
*                  macros and function prototypes required by the
*                  Force CSCI.
*
* CSCI IDENTIFIER:  HRF-xyz
*
* ORIGINATOR        BASELINE DATE    AUTH. DOC.
* -----
* John. Q. Developer    5/95          SCR 1234
*
* REVISED BY        BASELINE DATE    AUTH. DOC.
* -----
* Judy B. Coder        10/95         SDR 1234
*
*****/
/

/* Constant Definitions */

#define NUM_VEHICLES 6

typedef enum { False, True } BOOLEAN;

typedef struct
{
    double mass_lbm[NUM_VEHICLES];
    double mass_slg[NUM_VEHICLES]
    double accel[NUM_VEHICLES];
    double accel_x[NUM_VEHICLES];
    double accel_x[NUM_VEHICLES];

```

```

        double accel_x[NUM_VEHICLES];
        double force[NUM_VEHICLES];
        BOOLEAN continue_flag;
        BOOLEAN veh_active[NUM_VEHICLES];
    } FORCESTRUCT;

```

```

/* Macro Definitions */

```

```

#define LBM_TO_SLUG(a,b)    b = (a) * 0.03108
#define SQUARE(a)          (a) * (a)

```

```

/* Function Prototype Statements */

```

```

void Init_Mass(void);
void Get_Accel(void);
void Force_Calc(void);

```

```

#define FORCE_H
#endif

```

C6.2 CSCI SOURCE FILE

```

/*****
*
*
* FILENAME:    force.c
*
* DESCRIPTION:    This is the top-level executive source file for the
*                    Force CSCI, which calculates force for a vehicle
*                    based on mass and acceleration.
*
* CSCI IDENTIFIER:    HRF-xyz
*
* ORIGINATOR          BASELINE DATE    AUTH. DOC.
* -----
* John. Q. Developer    5/95                SCR 1234
*
* REVISED BY          BASELINE DATE    AUTH. DOC.
* -----
* Judy B. Coder         10/95               SDR 1234
*
*****/
/

#include "force.h"

FORCESTRUCT Force;
main()

```

```

{
/* initialize mass parameters */
Init_Mass();

/* loop until requested to stop */
while (Force.continue_flag)
{
/* calculate force after acquiring current acceleration */
Get_Accel();
Force_Calc();
}
}

```

C6.3 CSU SOURCE FILE 1

```

/*****
*
*
* FILENAME:   init_force.c
*
* DESCRIPTION:   This function initializes the mass values for all
*                   of the vehicles in the current scenario by
*                   converting those values from pounds mass to slugs.
*
* CSCI IDENTIFIER:  HRF-xyyz
*
* ORIGINATOR       BASELINE DATE  AUTH. DOC.
* -----
* John. Q. Developer   5/95           SCR 1234
*
* REVISED BY       BASELINE DATE  AUTH. DOC.
* -----
* Judy B. Coder       10/95          SDR 1234
*
*****/
/

#include <stdio.h>
#include "force.h"

extern FORCESTRUCT Force;

void Init_Mass(void)
{
short i;

```

```

/* loop once for each vehicle */
for (i = 0; i < NUM_VEHICLES; i++)
{
    /* if a mass is specified, set the vehicle active flag */
    /* and convert mass from pounds mass to slugs */
    if (Force.mass_lbm[i] != 0)
    {
        Force.veh_active[i] = True;
        LBM_TO_SLUG(Force.mass_lbm[i], Force.mass_slg[i]);
    }
    else
        Force.veh_active[i] = False;
}
}

```

C6.4 CSU SOURCE FILE 2

```

/*****
*
*
* FILENAME:   proc_force.c
*
* DESCRIPTION:   These functions perform the real-time processing
*                   associated with the Force CSCI, which calculates
*                   force based on vehicle accelerations.
*
* CSCI IDENTIFIER:   HRF-xyz
*
* ORIGINATOR         BASELINE DATE   AUTH. DOC.
* -----
* John. Q. Developer   5/95           SCR 1234
*
* REVISED BY         BASELINE DATE   AUTH. DOC.
* -----
* Judy B. Coder       10/95          SDR 1234
*
*****/
/
#include <stdio.h>
#include "lib_hrf_math.h"
#include "force.h"

extern FORCESTRUCT Force;

/*****

```

```

*
*
* FUNCTION:    Get_Accel
*
* DESCRIPTION:  This function acquires the current acceleration for
*                   each active vehicle in the current scenario.
*
*****
/
void Get_Accel(void)
{
    double temp;
    short i;

    /* loop once for each vehicle */
    for (i = 0; i < NUM_VEHICLES; i++)
    {
        /* if the vehicle is active, calculate total acceleration */
        /* by summing the acceleration vectors along each axis */
        if (Force.veh_active[i])
        {
            temp = SQUARE(Force.accel_x[i]) +
                SQUARE(Force.accel_y[i]) +
                SQUARE(Force.accel_z[i]);
            Force.accel[i] = Square_Root(temp);
        }
    }
}

/*****
*
*
* FUNCTION:    Force_Calc
*
* DESCRIPTION:  This function calculates the force on each active
*                   vehicle as a product of the vehicle's mass and
*                   acceleration.
*
*****
/
void Force_Calc(void)
{
    short i;

    /* loop once for each vehicle */
    for (i = 0; i < NUM_VEHICLES; i++)

```

```
{  
    /* if the vehicle is active, calculate force as a product */  
    /* of mass and acceleration */  
    if (Force.veh_active[i])  
        Force.force[i] = Force.mass[i] * Force.accel[i];  
    }  
}
```

```

#ifndef LIB_HRF_MATH_H
/*****
*
*
* FILENAME:   lib_hrf_math.h
*
* DESCRIPTION:   This is the header file for the lib_hrf_math library.
*                   It contains macros and function prototypes for all
*                   CSCIs that use this library.
*
* CSCI IDENTIFIER:   HRF-xyz
*
* ORIGINATOR         BASELINE DATE   AUTH. DOC.
* -----
* John. Q. Developer   5/95           SCR 1234
*
* REVISED BY         BASELINE DATE   AUTH. DOC.
* -----
* Judy B. Coder       10/95          SDR 1234
*
*****/
/

/* Macro Definitions */

#define DMXM(a,b,c) \
  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0] + a[0][2]*b[2][0]; \
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1] + a[0][2]*b[2][1]; \
  c[0][2] = a[0][0]*b[0][2] + a[0][1]*b[1][2] + a[0][2]*b[2][2]; \
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0] + a[1][2]*b[2][0]; \
  c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1] + a[1][2]*b[2][1]; \
  c[1][2] = a[1][0]*b[0][2] + a[1][1]*b[1][2] + a[1][2]*b[2][2]; \
  c[2][0] = a[2][0]*b[0][0] + a[2][1]*b[1][0] + a[2][2]*b[2][0]; \
  c[2][1] = a[2][0]*b[0][1] + a[2][1]*b[1][1] + a[2][2]*b[2][1]; \
  c[2][2] = a[2][0]*b[0][2] + a[2][1]*b[1][2] + a[2][2]*b[2][2];

/* Function Prototype Statements */

double Square_Root(double);

#define LIB_HRF_MATH_H
#endif

```

```

/*****
*
*
* FILENAME:   lib_hrf_math.c
*
* DESCRIPTION:   These functions perform standard mathematical
*                   calculation used in math models throughout the HRF.
*
* CSCI IDENTIFIER:   HRF-xyyz
*
* ORIGINATOR         BASELINE DATE   AUTH. DOC.
* -----
* John. Q. Developer   5/95           SCR 1234
*
* REVISED BY         BASELINE DATE   AUTH. DOC.
* -----
* Judy B. Coder       10/95          SDR 1234
*
*****/
/

#include <lib_math.h>
#include "lib_hrf_math.h"

/*****
*
*
* FUNCTION:   Square_Root
*
* DESCRIPTION:   This function returns the square root of the input
*                   variable to the calling routine. For negative
*                   input values, this function returns a zero.
*
*****/
/
double Square_Root(double input)
{
    double output;
    double temp;

    if (input <= 0)
        /* square root of 0 is 0, and the square root of a */
        /* negative number is undefined */
        output = 0;
}

```

```
else if (input == 1)  
    /* square root of 1 is 1 */  
    output = 1;  
else  
    {  
        temp = log(input);  
        output = exp(0.5 * temp);  
    }  
  
return(output);  
}
```