**SPSRB Common Standards Working Group**

**General Programming**

**Principles and Guidelines**



**Version 2.0**

**September, 2010**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**VERSION NUMBER IDENTIFIER**

The document version number which also corresponds to the Document Control Number (DCN) identifies whether the document is a working copy, final, revision, or update, defined as follows:

**Working Copy or Draft:** a document not yet finalized or ready for distribution; sometimes called a draft. Use 0.1A, 0.1B, etc. for unpublished documents.

**Final Copy:** the first definitive edition of the document after it passes through the drafting stage. This first edition is always identified as Version 1.0.

**Revision:** an edition with minor changes from the previous edition, defined as changes affecting less than one-third of the pages in the document. The version numbers for revisions 1.1 through 1.9, 2.1 through 2.9, and so forth. After nine revisions, any other changes to the document are considered an update. A revision in draft, i.e. before being re-baselined should be numbered as 1.1A, 1.1B, etc.

**Update:** an edition with major changes from the previous edition, defined as changes affecting more than one-third of the pages in the document. The version number for an update is always a whole number (Version 2.0, 3.0, 4.0, and so forth).

**DOCUMENT HISTORY**

**DOCUMENT REVISION LOG**

The Document Revision Log identifies the series of revisions to this document since the baseline release. Please refer to the above page for version number information.

|  |  |  |  |
| --- | --- | --- | --- |
| **DOCUMENT TITLE: General Programming Principles and Guidelines** | | | |
| **DOCUMENT CHANGE HISTORY** | | | |
| **Revision No.** | **Date** | **Revision Originator Project Group** | **CCR Approval # and Date** |
| 1.0 | June 18 2009 | Initial Release by CSWG | SPSRB approved  June 17 2009 |
| 2.0 | June 2010 | Update by CSWG | Pending |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**LIST OF CHANGES**

Significant alterations made to this document are annotated in the List of Changes table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **DOCUMENT TITLE: General Programming Principles and Guidelines** | | | | | |
| **LIST OF CHANGE-AFFECTED PAGES/SECTIONS/APPENDICES** | | | | | |
| **Version Number** | **Date** | **Changed By** | **Page** | **Section** | **Description of Change(s)** |
| 2.0 | September 2010 | CSWG | All | All | Major layout and formatting update; sections in the document were rearranged and formatted to be consistent with all other CSWG documentation. |
| 2.0 | September 2010 | CSWG | 18 | 6.1 | New security section added. |
| 2.0 | September 2010 | CSWG | 12 | 2.6. 2.6.1 | Header format was restructured; examples were updated accordingly. |
| 2.0 | September 2010 | CSWG | 22 | Appendix A | The appendix was updated to reflect the new header format and layout standards. |
| 2.0 | September 2010 | CSWG | 18 | 5.1 | New standard was added to state that filenames should not be hardcoded. |
| 2.0 | September 2010 | CSWG | 10 | 2.4 | Inner nested loop standard was changed to a guideline. |
| 2.0 | September 2010 | CSWG | 16 | 4.1.1 | Example 6 was updated to be consistent with the text. |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**TABLE OF CONTENTS**

1. **Introduction ………………………………………………….**
   1. Programming Standards and Guideline Definitions …………….......
   2. Reference Documents ….....…………………………………………
2. **Formatting Basics …………………………………...............**
   1. Program Unit Organization ……………………..…………………...
   2. Program Unit Size ….………………………...…..…………….…...
   3. Naming Convention ………………………………………………...
   4. Indentations ………………………………………….………………
      1. Example 1: Use of Indentations in Nested Loops ………………..
   5. Nesting ……………………………………………….………………
      1. Example 2: Use of Parentheses in Nested Loops .………………..
   6. Headers ………………………………...…………….………………
      1. Example 3: Sample Header for the Module Noise ………………..
3. **Declarations and Return …………………………………….** 
   1. Variable Declarations ………………………………………..……….
   2. Input and Output (I/O) ……………………………………………….
   3. Check Return Values …………………..……………………………
      1. Example 4: I/O Statements………………………………………..
      2. Example 5: Use of Dynamic Memory Allocation ……………...…
4. **Readability ……………..…………………………………….** 
   1. Readability of General Programs ………………………..…..……….
      1. Example 6: Use of Consistent Variable Spaces and Size ………….
      2. Example 7: Use of Blocking and Comment Lines ..……………….
      3. Example 8: Use of Parentheses in Logical Expressions ..………….
5. **Features and Items to Avoid .……………………………….** 
   1. No Hardcoding ……..………………………………………..……….
6. **Security Concerns …………….…………………………….** 
   1. Security Issues ……..………………………………………..……….

**APPENDIX A .....................................................................................**

**6**

7

7

**8**

8

9

9

10

11

11

12

12

13

**14**

14

15

15

15

15

**16**

16

16

17

18

**18**

18

**18**

19

**22**

# **1. INTRODUCTION**

The National Environmental Satellite Data Distribution Service (NESDIS) develops and implements algorithms that transform environmental satellite images of the Earth into meaningful environmental data which are then employed in a full-time operational setting. In the past, software developed within NESDIS was created by the differing entities throughout the service, each creating code to fulfill various research, operational and archival needs. This meant software was written in various programming languages and idiosyncratic styles, moreover suffering from a lack of coordinating documentation in most cases. The resulting software is consequently often costly to maintain as the source code may have been mislaid, the code may be difficult to read and understand, documentation may be inadequate, or the original developers may no longer be able to maintain their code.

The purpose of developing common software programming standards is to reduce the cost of the software lifecycle and streamline the algorithm implementation process. This follows a trajectory from initial research and software development to operational use and finally through to divestiture and retirement where costs accumulate throughout the lifecycle. Implementation of these Satellite Products and Services Review Board (SPSRB) approved coding standards will shift costs away from operations and maintenance as the problems are resolved upstream. Promoting the accountability of the developers and scientists to create standardized software programs will benefit NESDIS as a whole. Higher front-end expenditure will be repaid in the form of lower operational and maintenance costs over subsequent years. It is intended that the implementation expenses of the common software standards will be funded through the Office of Systems Development (OSD) Product System Development and Implementation (PSDI) process, and must be included in relevant budgets and projects plans when applying for PSDI funds.

Having common programming standards used by all SPSRB stakeholders will aid in cross-organization communication and implementation of codes. It will also produce a software catalog that:

* Is robust
* Is readily portable (platform independent)
* Is modular and reusable
* Is inexpensive to implement and maintain operationally
* Is written in a widely used and supported language
* Has a common look and structure
* Adheres to best programming practices
* Is well documented
* Is easily readable and understandable
* Behaves in a standard manner (exception handling, file input/output)
* Uses common shared libraries

**1.1 Programming Standards and Guideline Definitions**

It is recognized that certain stylistic suggestions which make code easier to read (e.g. lining up attributes, or using all lower case or mixed case) are subjective and therefore should not have the same weight as techniques and practices that are known to improve code quality. For this reason, the standards within documents produced by the SPSRB Common Standards Working Group are divided into three components; Standards, Guidelines and Recommendations (SGRs):

***Standard***:Aimed at ensuring portability, readability and robustness. Compliance with this category is mandatory.

***Guideline***:Good practices. Compliance with this category is strongly encouraged. The case for deviations will need to be argued by the programmer.

***Recommendation***:Compliance with this category is optional, but is encouraged for consistency.

These three standards will thus be found in the above format throughout this document, indicating the weight of a particular standard. If possible, all standards, guidelines and recommendations should be followed when programming. Else, programmers should include these components whenever possible, keeping in mind their respective weight. Please refer to these definitions as needed.

**1.2 Reference Documents**

Martin, B., M. Brown, A. Paller, D. Kirby. ***2010 CWE/SANS Top 25 Most Dangerous Programming Errors. Common WeaknessEnumeration***, The MITRE Corporation; 2010. Found at http://cwe.mitre.org/top25/archive/2010/2010\_cwe\_sans\_top25.pdf

Raymond, E. S. ***The Art of Unix Programming***. Addison-Wesley Professional; 1 edition (October 3, 2003). The principles may be found at http://en.wikipedia.org/wiki/Unix\_philosophy.

Seacord, R. ***Cert Top 10 Secure Coding Practices***. Last updated February 2010. Found at <https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices>.

**2. Formatting Basics**

Eric S. Raymond, in his book *The Art of Unix Programming*, summarizes the Unix philosophy as the widely-used engineering philosophy, ***"Keep it Simple, Stupid"*** (KISS Principle). He then describes how he believes this overall philosophy is applied as a cultural Unix norm:

1. *Rule of Simplicity*: Design for simplicity; add complexity only where you must.
2. *Rule of Modularity*: Write simple parts connected by clean interfaces.
3. *Rule of Clarity:* Clarity is better than cleverness.
4. *Rule of Composition*: Design programs to be connected to other programs.
5. *Rule of Transparency*: Design for visibility to make inspection and debugging easier
6. *Rule of Parsimony*: Write a big program only when it is clear by demonstration that nothing else will do.
7. *Rule of Repair*: When you must fail, fail noisily and as soon as possible.
8. *Rue of Economy*: Programmer time is expensive; conserve it in preference to machine time.
9. *Rule of Generation*: Avoid hand-hacking; write programs to write programs when you can.
10. *Rule of Optimization*: Prototype before polishing; get it working before you optimize it.
11. *Rule of Extensibility*: Design for the future, because it will be here sooner than you think.

The programming principles described here and in the language-specific coding documents adhere to the “KISS” principles above.

**2.1 Program Unit Organization**

***Standard***: *Elements of the program units shall include the following and shall be organized as shown:*

a. program unit identifier,

b. header,

c. INCLUDE files,

d. specification statements,

e. DATA or parameter statements for constants,

f. statement function statements,

g. executable statements,

h. statements to stop the execution of the program unit.

**2.2 Program Unit Size**

***Standard****: The maximum number of characters per line is 90.*

Otherwise the compilers will not read the text.

***Recommendation****: Each program unit is kept as small and simple as possible to perform a specific task.*

Use multiple, smaller routines with well-defined functions rather than a larger routine that does a lot of things. Unwieldy program units spanning hundreds of lines should be examined to see if they can be segmented.

**2.3Naming Convention**

There is no standard for naming conventions, as code will work with all names composed of recognized characters. That is why they are called naming “conventions”, not naming “standards.”

***Standard:*** *Names cannot be identical to reserved words or implementation supplied function names.*

This creates confusion, so it must be completely avoided.

***Recommendation****: Naming conventions within a programming community are under continual development, as programmers communicate with each other and agree to adopt particular conventions.*

***Recommendation****: When writing code, the names of files, subroutines, functions, modules and variables created by a programmer are often up to the programmer.*

A possible exception could be if a project to integrate several algorithms from disparate sources into a single program unit developed standardized abbreviations for common variables. Left to their own devices a programmer can choose to make the names long or short, descriptive or useless, clear or confusing. A lot depends on the mindset of the programmer. Will this code be reused? Is this "quick and dirty" code? Is this code so clear that it is self-explanatory? As this document deals with code that will be transitioned to operations and could potentially remain in operations for many years it is important that the code be readily maintainable and easily understood.

***Recommendation****: The names of files, subroutines, functions and variables can be extremely useful in making code more readable.*

Choosing names may seem not very important, but insisting on meaningful names helps a programmer to organize thoughts and produce code that is readable and reviewable.

***Recommendation****: Avoid names that look alike (e.g do not use characters that resemble each other).*

For example, names like 2 and z, 0 and O, 5 and S, or I and 1.

***Recommendation****: Name program units are to indicate their purpose.*

Familiarize yourself with the STAR Common Library when it becomes available. This serves two main purposes. First, find a library routine that you can use to implement your desired function and second, avoid using names that are similar to library routines.

***Recommendation****: Name symbolic variables to indicate what they are, not what values they may contain.*

***Recommendation****: Names should be as mnemonically descriptive as possible, subject to constraints imposed by language standards.*

***Recommendation****: Names shall not be identical to reserved words or implementation supplied function names.*

Names should not resemble reserved words or implementation-supplied function names.

**2.4 Indentations**

Indentation shall be used consistently to enhance readability throughout a program.

***Standard:*** *Each indentation should use at least two spaces.*

***Standard:*** *A comment line should be indented in the same way as the following executable line of code.*

***Standard:*** *Statements in nested loops should be indented so that all statements in the same nesting are indented by the same amount.*

***Guideline:*** *Statements in inner nested loops should be indented by a greater amount than statements in outer nested loops.*

### **2.4.1 Example 1: Use of Indentations in Nested Loops**

The following example shows how the use of indentations clarifies the intent of the code.

Good example of indentation usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

! Loop over values of x and y

! Compute the sides of a right triangle

! Then compute the square of the hypotenuse

DO i=1,5

x = x\_value(i)

DO j=1,4

y = y\_value(j)

a = x + 6

b = y / 4.5

c\_squared(i,j) = a \* a + b \* b

END DO

END DO

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

First bad example of indentation usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

! Loop over values of x and y

DO i=1,5

x = x\_value(i)

DO j=1,4

y = y\_value(j)

! Compute the sides of a right triangle

a = x + 6

b = y / 4.5

! Compute the square of the hypotenuse

c\_squared(i,j) = a \* a + b \* b

! Close the loops

END DO

END DO

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Second bad example of indentation usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

! Loop over values of x and y

DO i=1,5

x = x\_value(i)

DO j=1,4

y = y\_value(j)

! Compute the sides of a right triangle

a = x + 6

b = y / 4.5

! Compute the square of the hypotenuse

c\_squared(i,j) = a \* a + b \* b

! Close the loops

END DO

END DO

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**2.5 Nesting**

***Standard:*** *The nesting of parentheses in logical and arithmetic expressions shall be limited to four (4) levels.*

If an expression requires a greater level of nesting, it shall be separated into more than one expression.

### **2.5.1 Example 2: Use of Parentheses in Nested Loops**

The following example shows how the use of parentheses clarifies the intent of the code.

Good example of parentheses and spacing usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

mu\_s = COS ( pi \* sza / 180.)

mu\_v = COS ( pi \* sva / 180.)

tan\_s = TAN ( pi \* sza / 180.)

tan\_v = TAN ( pi \* sva / 180.)

d = SQRT (tan\_s \* tan\_s + tan\_v \* tan\_v –

& 2.0 \* tan\_s \* tan\_v \* COS ( pi \* relaz / 180.) )

fac = tan\_s \* tan\_v \* SIN (pi \* relaz / 180.)

cost = SQRT ( d \* d + fac \* fac )/ ((1. / mu\_s) + (1. / mu\_v ) )

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bad example of parentheses and spacing usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

mu\_s = COS(pi\*sza/180.)

mu\_v = COS(pi\*sva/180.)

tan\_s = TAN(pi\*sza/180.)

tan\_v = TAN(pi\*sva/180.)

cost = SQRT((tan\_s \* tan\_s + tan\_v \* tan\_v –

& 2.0 \* tan\_s \* tan\_v \* COS(pi\*relaz/180.) +

& (tan\_s \* tan\_v \* SIN(pi\*relaz/180.) \* tan\_s \*

& tan\_v \* SIN(pi\*relaz/180.)))) / ((1./mu\_s) + (1./mu\_v))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**2.6 Headers**

These headers must contain the items listed as a standard and should contain the other items for good measure. Please provide the fields in the order which they are presented to maintain consistency.

***Standard***: *Every new program unit shall contain a header.*

Designate information required in the header with the following keywords:

1. *NAME*: The name of the program unit.
2. *TYPE/LANGUAGE*: The language and type of program (e.g., F95 module, Per script).
3. *FUNCTION*: A brief description of the program unit function (e.g., 1-2 sentences).
4. *DESCRIPTION:* A description of the program unit processing (e.g., diagrams, PDL).
5. *FILES NEEDED:* Any files which are opened or closed by this program unit (e.g., algorithm I/O).
6. *MODULES/SUBROUTINES NEEDED:* Modules and subroutines needed from an external source.
7. *SUBROUTINES CONTAINED:* Any subroutines contained internally in the program text.
8. *CALLING SEQUENCE*: The source statements necessary to invoke a program unit.
9. *INPUTS*: A description of the program unit inputs (e.g., parameters, files).
10. *OUTPUTS*: A description of the program unit outputs (e.g., parameters, files).
11. *SYSTEM CALLS*: Any calls made to an operating system.

***Guideline***: *Additional header items listed below should be included as necessary.*

Designate information with the following keywords:

1. *REFERENCE*: The reference(s) to program unit design materials (e.g., requirements document, design document, standards, algorithm decisions).
2. *USAGE*: What the program is using (e.g., a calling sequence).
3. *RETURN VALUES*: Parameter values to be returned from the program unit.
4. *VERSION*: The version number, at least internally referenced.
5. *HISTORY*: The revision history of the program unit.
6. *DEPENDENCIES*: A description of the program unit dependencies (e.g., HW/SW dependencies, INCLUDE files, operating systems, initialization).
7. *ERROR CODES/EXCEPTIONS:* Description of, or link to, the error codes used in the program unit.
8. *RESTRICTIONS/LIMITATIONS*: Known restraints on the program (e.g., using a specific compiler version).

### **2.6.1 Example 3: Sample Header for the Module Noise**

Refer to Appendix A for more examples of proper header format and usage.

!----------------------------------------------------------------------------

! Name: Noise

!

! Type: F90 module

!

!Function:

! The F90 module is design to contain subroutines which are

! available for calls from external program units, unless declared

! with the private attribute

!

! Description:

! This module contains the various subroutines related to the

! handling of the instrument noise, such as generation of artificial

! noise or the computation of the noise

!

!Files Needed:

! N/A

!

! Modules Needed:

! - misc

! - IO\_Noise

!

! Subroutines Contained:

! - LoadNoise

! - ComputeNoise

! - NoiseOnTopOfRad

! - GenerateNoiseErr

! - BuildMatrxFromDiagVec

! - GRNF

!

!Calling Sequence:

! N/A

!

!Inputs:

! N/A

!

!Outputs:

! N/A

!

!System Calls:

! N/A

!

! History:

! 2007-03-15 S.A. Boukabara IMSG Inc. @ NOAA/NESDIS/ORA

! -Original Code

!

!----------------------------------------------------------------------------

**3. Declarations and Return**

**3.1 Variable Declarations**

***Standard***: *Align each declaration type name in the same column to improve readability.*

***Standard****: Avoid extremely long or continuation lines in a declaration statement by using multiple statements.*

***Standard****: List several variables of a single type on a line alphabetically.*

***Standard****: Explicitly dimension all arrays using parameters as much as possible to specify array dimensions/sizes. We recognize that it is not always possible to do so.*

***Standard****: Use of dynamic memory allocation is encouraged.*

**3.2 Input and Output (I/O)**

***Guideline****: Identify the input and output variables in the header.*

***Recommendation****:*  *Separate Input, Output, and Processing functions in a program so that all Input functions precede all Processing functions and are followed by all Output functions.*

Exceptions to this rule occur when memory constraints require dynamic allocation of memory within the processing function. When dynamic allocation is used, input and output functions within processing functions should be clearly identified by comments that identify the input/output variables with references to the Preamble and/or design documents.

**3.3 Check Return Values**

***Standard***: *Check for error return values, even from functions that "can't" fail. It is recommended that the following convention be used for error return values:*

* + A value of zero indicates the function completed successfully.
  + A negative value indicates the function failed.
  + A positive value indicates the function completed successfully but encountered something unexpected.
  + Include the system error text for every system error message.

***Standard***: *Take special care with I/O statements since these are usually affected by events beyond the control of the programmer.*

Include an item which causes control to be transferred to the statement attached to that label in the event of an error. This must, of course, be an executable statement and in the same program unit.

### **3.3.1 Example 4: I/O Statements**

The following example shows how the use an executable statement properly.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

READ(UNIT=IN, FMT=\*, ERR=999) VOLTS, AMPS

WATTS = VOLTS \* AMPS

rest of program in here . . . . . and finally

STOP

999 WRITE(UNIT=\*,FMT=\*)'Error reading VOLTS or AMPS'

END

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Standard***: *Handle the end-of-file condition when reading past the end of a sequential/ internal file.*

***Recommendation***: *Programmers should check the success of any dynamic memory allocation or deallocation.*

### **3.3.2 Example 5: Use of Dynamic Memory Allocation**

Check for the success of this allocation. Note, there is an error if STAT is not equal to zero.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ALLOCATE(x(N,N),STAT=alloc\_stat)

If(STAT.eq.0)then

. . .

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4. Readability**

**4.1 Readability of General Programs**

Consistency is the key to making programs easily readable.

***Standard***:Elements of the program units shall include the following and shall be organized as shown:

1. program unit identifier,
2. header,
3. INCLUDE files,
4. specification statements,
5. DATA or parameter statements for constants,
6. statement function statements,
7. executable statements,
8. statements to stop the execution of the program unit.

***Standard:*** *Begin each program unit at the top a new page.*

***Standard:***  *Use a maximum of 90 characters per line of code.*.

***Standard:*** *Alphabetic case shall be used consistently to enhance readability throughout a program.*

***Standard:*** *Place spaces before and after compound expressions (relational operators, reserved words, identifiers, and arithmetic operators) to enhance readability of compound expressions.*

Refer to the examples below.

### **4.1.1 Example 6: Use of Consistent Variable Spaces and Size**

The following example shows how the consistent use of variables clarifies the intent of the code.

Good examples of parentheses and spacing usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

tempC = ( tempF – 32.0 ) \* 5 / 9

C = A + B \* X

d = e + f + y

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bad example of parentheses and spacing usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C=a+B\*x

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Standard:***A comment line shall be separated from a preceding executable line of code by a single blank line.

***Guideline:*** Blocking with blank lines shall be used consistently to enhance readability throughout a program.

***Guideline:*** All comment lines that are followed by an executable line of code should be separated from the executable line of code either by a single blank line or by no blank line.

This is an optional matter of style that should be used consistently throughout a program.

### **4.1.2 Example 7: Use of Blocking and Comment Lines**

The following example shows how the use of proper comment line formatting clarifies the intent of the code.

Good example of blocking and comment line usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

! Compute the sides of a right triangle

a = x + 6

b = y / 4.5

! Compute the square of the hypotenuse

c\_squared = a \* a + b \* b

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Good example of blocking and comment line usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

! Compute the sides of a right triangle

a = x + 6

b = y / 4.5

! Compute the square of the hypotenuse

c\_squared = a \* a + b \* b

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bad example of blocking and comment line usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

! Compute the sides of a right triangle

a = x + 6

b = y / 4.5

! Compute the square of the hypotenuse

c\_squared = a \* a + b \* b

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Guideline:*** *The evaluation of logical and arithmetic expressions shall be clarified through the use of parentheses and spaces.*

### **4.1.3 Example 8: Use of Parentheses in Logical Expressions**

The following example shows how the use of parentheses clarifies the intent of the code.

Good example of parentheses usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

pk = pk - 1.0 + ( 0.5 \* REAL(ning) )

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bad example of parentheses usage:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

pk = pk - 1.0 + 0.5 \* REAL(ning)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**5. Features and Items to Avoid**

**5.1 No Hardcoding**

***Standard:*** *Filenames should not be hardcoded into software.*

Use parameters instead of filenames when hardcoding.

***Guideline:*** *Variables should not be hardcoded into software.*

It is good to keep the code as flexible as possible. Global constants such as mathematical or geophysical constants, such as Pi or the Earth’s radius, should be contained in a single constants file. Examples of software elements that should not be hardcoded include file paths, temporal data (Year, Month or day), spatial extents (latitudes or longitudes), etc.

**6. Security Concerns**

This document lists general issues that apply to all software. Language specific considerations are addressed in those specific documents (for example, C is specifically prone to buffer overflows). These practices will result in code that is more reliable, easier to maintain, and less likely to impact other applications on the same system.

**6.1 Security Issues**

The issue of software security has become a high priority issue within NESDIS. How this will affect the code we develop for our scientific applications is still evolving. Most of the information available on the internet refers to applications that sit on web servers. Very few of our applications are exposed to the public in this fashion. Most of our applications reside on servers shared with other applications. When one considers availability and reliability under the umbrella of software security, it becomes apparent that there are issues programmers need to consider. This section outlines some potential security issues to avoid when developing code for operational use. We call these “security issues”, not because they specifically constitute a vulnerability to outside attack, but because they are software defects that may pose a risk to all operational programs and hardware running on a given machine or within a network. These design flaws can take down software, hardware, and networks, and because they occur internally they can have devastating impacts.

***Standard:*** *Never hardcode passwords*.

This is a security violation. Set up keys when there is a need to transfer data between systems.

***Standard:*** *Always verify inputs.*

When you code a read statement from any data source, know what you expect and check for that, whether it is a number within a certain range, a date string, a filename, an array, etc. Check string lengths, integer size, and array bounds, for example. Be especially careful if you are reading input to allocate memory, such as from a file header. If that content is corrupt or larger than anticipated, one program could consume all of the machine’s memory resource. If you do not read what you expect, generate an error and stop processing.

***Standard:*** *Clean up after the application.*

Include a clean-up script in the application package. Things that can potentially fill a disk, if not removed include: input data, output data, log files, internal mail messages, intermediate files. This issue is especially problematic when new files are added after the initial delivery as a maintenance change. (Software security sources recommend the institution of disk quotas, so that one application that goes out of control does not impact other applications on the shared system.)

***Standard:*** *Watch for hung programs and standard output.*

It is convenient to direct the standard output and standard error from a program to a log file for diagnostic purposes. However, a program hung in an infinite loop can spew output to a log file and potentially fill the disk. Therefore, a software package should always be able to limit file sizes and check for hung processes. The run time for a process can be obtained using the ps system command and the process can be killed using the kill system command. Any resources that program had, that were not released upon its death, should be freed.

***Standard:***  *Limit frequency and number of connections via ftp/sftp or scp.*

Failure to do so can consume bandwidth on a remote server creating an unintended denial of service attack. Scripts are often used to automate these connections and transfers. A script can be constructed so it sleeps (using the *sleep* system command) between cycles in which is initiates connections. This will slow the frequency of access. Also, the software can be written so that it limits the total number of connections allowed at any given time.

***Standard:*** *Tailor error messages to the appropriate audience.*

This security issue is identified in references to prevent hackers from learning about system paths, or other sensitive details. But it is a good practice for application development. Too much information confuses the recipient of the message and tends to be ignored. Messages to be send to the operators need less technical information than messages intended for the application programmer.

***Standard:***  *Use compiler options; turn on all checking during development.*

Eliminate warnings by modifying the code – do not just look at errors that prevent the compile. This can catch many flaws. To improve operational performance, the final recompile usually removes most of this checking for operational processing. But compiler options should be turned on again when verifying maintenance changes.

***Standard:*** *Own the source code.*

Do not use executable files obtained from outside sources. If there is a requirement to use executable files without obtaining the source code, security must verify that they were obtained from a trusted source.

***Standard:*** *Monitor all changes.*

When developing on a shared system, or after making a change on an operational system, always verify that something running via a scheduler is working as expected. Monitor at least one or more runs of the software to ensure proper function. Changes made and not monitored are a frequent source of problems in operations.

***Standard:*** *Check ownership and permissions.*

Make sure consideration is made to allow only the appropriate user ID (the production ID for operational systems) to modify and run the application.

***Standard:*** *Be careful with usage of* rm -f.

If you supply this command with a variable that holds a path name you want removed, make sure that path exists. If the variable is undefined, or simply defined as “/”, this command will recursively erase everything in the entire file system owned by that user and everything to which that user has write permission.

***Standard:*** *Avoid system calls if there is a system function interface call available in the language you are using.*

For example, do not use a system call to ftp when you can use the Perl ftp module instead. This improves portability of the software.

***Standard:*** *If you must use system calls (referred to as backtick calls), always check that the system returned the expected value*.

For example, for calls such as “$date =`date +%y%m%d`;” always verify that the system returned a reasonable value. Or, for Perl, use and check return codes.

***Standard:*** *If multi-threading is in your design, a simple and useful way to run multiple instances of the same program at one time is to use the* fork exec *approach.*

This concept is illustrated nicely at: <http://beej.us/guide/bgipc/output/html/singlepage/bgipc.html>. When using this approach, software should always:

1. limit the total number of such processes
2. free zombies (completed child processes whose status has not been checked).

Failure to do so will consume system resources on the system (e.g. CPUs, memory) as well as for the given user.

***Guideline:*** *Watch the use of multiple thread situations as this can be dangerous.*

An application can behave differently than expected if more than one instance is kicked off at once. Satellite data can arrive in spurts. Think about whether shared intermediate files and ancillary files, or simultaneous writing to logs or output could be a problem. Scripts or control software (such as OPUS) can be used to ensure an application remains single threaded.

***Recommendation***: *Root configure the system so that each user is also allowed a certain number of processes he is allowed to run (defined as maxprocs) at any instant.*

# **APPENDIX A**

## **Sample Module compliant with General Programming Principles**

This example includes all modules and subroutines contained with Noise.

!----------------------------------------------------------------------------! Name: Noise

!

! Type: F90 module

!

!Function:

! This module contains the various subroutines related to the

! handling of the instrument noise.

!

! Description:

! N/A

!

!Files Needed:

! N/A

!

! Modules Needed:

! - misc

! - IO\_Noise

!

! Subroutines Contained:

! - BuildMatrxFromDiagVec

! - LoadNoise

! - ComputeNoise

! - NoiseOnTopOfRad

! - GenerateNoiseErr

! - GRNF

!

!Calling Sequence:

! N/A

!

!Inputs:

! N/A

!

!Outputs:

! N/A

!

!System Calls:

! N/A

!

! History:

! 2007-03-15 S.A. Boukabara IMSG Inc. @ NOAA/NESDIS/ORA

! -Original Code

!

!----------------------------------------------------------------------------

MODULE Noise

USE misc

USE IO\_Noise

IMPLICIT NONE

PRIVATE

!---Publicly available subroutine

PUBLIC :: LoadNoise,ComputeNoise,NoiseOnTopOfRad,GRNF

PUBLIC :: GenerateNoiseErr,BuildMatrxFromDiagVec

!---Publicly available data/types, etc

PUBLIC :: Noise\_type,noiseInfo

!---Declaration sections

TYPE :: Noise\_type

INTEGER :: nchan !Number of channels

REAL, DIMENSION(:), POINTER :: CentrFreq !Center Frequencies

REAL, DIMENSION(:), POINTER :: rms !rms

REAL, DIMENSION(:), POINTER :: nedt !nedt vector

END TYPE Noise\_type

TYPE(Noise\_type) :: noiseInfo

!---Module-wide visible variables

INTEGER :: ISEED=2

!---INTRINSIC functions used in this module

INTRINSIC :: RANDOM\_SEED,ALOG,ATAN,COS,RANDOM\_NUMBER,SQRT

CONTAINS

!===============================================================

! Name: BuildMatrxFromDiagVec

!

! Type: Subroutine

!

!Function:

! Returns a diagonal matrix

!

! Description:

! Builds a matrix from diagonal elements

! contained in a vector. Off-diagonal are set to 0.

!  
!Files Needed:

! - None

!

! Modules Needed:

! - None

!

! Subroutines Contained:

! N/A

!

!Calling Sequence:

! CALL BuildMatrx(nchan, noiseRMS, Se)

!

! Input:

! Name Type Description

! ---------------------------------------------------

! - nchan I Number of channels

! - noiseRMS I Vector of diagonal elements

!

!Output:

! Name Type Description

! ---------------------------------------------------

! - Se O Matrix with diagonal elmts

!

!System Calls:

! None

!

! History:

! 03-22-2007 Sid Ahmed Boukabara, IMSG Inc @ NOAA/NESDIS/ORA

!

!===============================================================

SUBROUTINE BuildMatrxFromDiagVec(nchan,noiseRMS,Se)

REAL, DIMENSION(:) :: noiseRMS

REAL, DIMENSION(:,:) :: Se

INTEGER :: nchan,ichan

Se = 0.

DO ichan=1,nchan

Se(ichan,ichan) = noiseRMS(ichan)\*\*2.

ENDDO

RETURN

END SUBROUTINE BuildMatrxFromDiagVec

!===============================================================

! Name: LoadNoise

!

! Type: Subroutine

!

!Function:

! Reads an ascii file contain instrument noise values

!

! Description:

! Reads noise values from noise file and loads

! them into Noise structure called NoiseInfo

! (see definition on top of module)

!  
!Files Needed:

! - Noise file (path/satId\_sensor1\_nedt\*.dat)

!

! Modules needed:

! - ReadNoise

!

! Subroutines Contained:

! N/A

!

!Calling Sequence:

! CALL LoadNoise(NoiseFile)

!

!Input:

!

! Name Type Description

! ---------------------------------------------------

! - NoiseFile I Name of noise file

!

!Output:

! - None

!

!System Calls:

! None

!

! History:

! 03-22-2007 Sid Ahmed Boukabara, IMSG Inc @ NOAA/NESDIS/ORA

!

!===============================================================

SUBROUTINE LoadNoise(NoiseFile)

CHARACTER(LEN=\*) :: NoiseFile

INTEGER :: iu

!---Open file containing radiance measurements

iu=get\_lun()

OPEN(iu,file=NoiseFile,form='formatted',status='old')

READ(iu,'(25x,i8)') noiseInfo%nChan

ALLOCATE(noiseInfo%CentrFreq(noiseInfo%nChan),noiseInfo%rms(noiseInfo%nChan),&

noiseInfo%nedt(noiseInfo%nChan))

CLOSE(iu)

CALL ReadNoise(NoiseFile,noiseInfo%CentrFreq,noiseInfo%nedt,noiseInfo%nchan)

noiseInfo%rms(1:noiseInfo%nChan)=noiseInfo%nedt(1:noiseInfo%nChan)

RETURN

END SUBROUTINE LoadNoise

!===============================================================

! Name: ComputeNoise

!

! Type: Subroutine

!

!Function:

! Returns a vector contain instrument noise values

!

! Description:

! This subroutine stores the noise elements contained in

! the noise structure into an independent vector

!

!Files Needed:

! - None

!

! Modules needed:

! - None

!

!

! Subroutines Contained:

! N/A

!

!Calling Sequence:

! CALL ComputeNoise(noiseRMS)

!

!Input:

! -None

!

!Output:

! Name Type Description

! ---------------------------------------------------

! - noiseRMS O Vector of noise values

!

!System Calls:

! -None

!

! History:

! 03-22-2007 Sid Ahmed Boukabara, IMSG Inc @ NOAA/NESDIS/ORA

!

!===============================================================

SUBROUTINE ComputeNoise(noiseRMS)

REAL, DIMENSION(:) :: noiseRMS

noiseRMS(:) = noiseInfo%rms(:)

RETURN

END SUBROUTINE ComputeNoise

!===============================================================

! Name: NoiseOnTopOfRad

!

! Type: Subroutine

!

!Function:

! Returns an array of brightness temperatures with added noise

!

! Description:

! Adds noise on top of the brightness temperatures.

! Useful when doing forward simulations with a forward model.

!

!Files Needed:

! - None

!

! Modules needed:

! - None

!

! Subroutines Contained:

! N/A

!

!Calling Sequence:

! CALL NoiseOnTopOfRad(nchan,TB,NoiseErr)

!

! Input:

! Name Type Description

! ---------------------------------------------------

! - nchan I Number of channels

! - TB I/O Vector of brightness temperatures

! - NoiseErr I Vector of channel-based noise

!

!Output:

! Name Type Description

! ---------------------------------------------------

! - TB I/O Vector of brightness temperatures

!

!System Calls:

! None

!

! History:

! 03-22-2007 Sid Ahmed Boukabara, IMSG Inc @ NOAA/NESDIS/ORA

!

!===============================================================

SUBROUTINE NoiseOnTopOfRad(nchan,TB,NoiseErr)

REAL, DIMENSION(:) :: TB,NoiseErr

INTEGER :: nchan,ichan

DO ichan=1,nchan

TB(ichan)= TB(ichan)+NoiseErr(ichan)

ENDDO

RETURN

END SUBROUTINE NoiseOnTopOfRad

!===============================================================

! Name: GenerateNoiseErr

!

! Type: Subroutine

!

!Function:

! Returns noise values based on a normal distrubution

!

! Description:

! Generates a random normal distribution noise based on a

! random number generator combined with a-priori noise

! information

!

!Files Needed:

! - None

!

! Modules needed:

! - None

!

! Subroutines Contained:

! N/A

!

!Calling Sequence:

! CALL GenerateNoiseErr(nchan,NoiseErr)

!

!Inputs:

! Name Type Description

! ---------------------------------------------------

! - nchan I Number of channels

!

!Outputs:

! Name Type Description

! ---------------------------------------------------

! - NoiseErr O Vector of randomly generated

! noise values

!

!System Calls:

! None

!

! History:

! 03-22-2007 Sid Ahmed Boukabara, IMSG Inc @ NOAA/NESDIS/ORA

!

!===============================================================

SUBROUTINE GenerateNoiseErr(nchan,NoiseErr)

REAL, DIMENSION(:) :: NoiseErr

REAL :: X

INTEGER :: nchan,ichan

DO ichan=1,nchan

CALL GRNF (X)

NoiseErr(ichan) = X\*noiseInfo%rms(ichan)

ENDDO

RETURN

END SUBROUTINE GenerateNoiseErr

!===============================================================

! Name: GRNF

!

! Type: Subroutine

!

!Function:

! Returns a random number between 0 and 1

!

! Description:

! Gaussian random number generated from uniform random number

!

!Files Needed:

! -None

!

! Modules needed:

! -None

!

! Subroutines Contained:

! N/A

!

!Calling Sequence:

! CALL GRNF(X)

!

!Input:

! -None

!

!Output:

! Name Type Description

! ---------------------------------------------------

! - X O Random number generated

!

!

!

!System Calls:

! None

!

! History:

! 03-22-2007 Sid Ahmed Boukabara, IMSG Inc @ NOAA/NESDIS/ORA

!

!===============================================================

SUBROUTINE GRNF (X)

REAL, INTENT (OUT) :: X

REAL :: PI,R1,R2,X0

PI = 4.0\*ATAN(1.0)

CALL RANDOM\_NUMBER(HARVEST=X0)

R1 = -ALOG(1.0-X0)

CALL RANDOM\_NUMBER(HARVEST=X0)

R2 = 2.0\*PI\*X0

R1 = SQRT(2.0\*R1)

X = R1\*COS(R2)

END SUBROUTINE GRNF

END MODULE Noise