**SPSRB Common Standards Working Group**

**Standards, Guidelines and Recommendations for**

**Writing Fortran 90/95 Code**

**S. –A. Boukabara**

**SAIC for NOAA/NESDIS/STAR**

**P. Van Delst**

**NOAA/NCEP/EMC**



**Version 2.0**

**August, 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 for unpublished documents.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 22 2009 | Initial Release by CSWG | SPSRB approved  June 17 2009 |
| 2.0 | September 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. |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**TABLE OF CONTENTS**

1. **Introduction ………………………………………………….**
   1. Programming Standards and Guideline Definitions …………….......
   2. Reference Documents ….....…………………………………………
2. **Language Features …………………………………..............**
   1. Encouraged Practices and Features ………….………….…………...
   2. Procedures ….………………………………..………………….…...
3. **Formatting …..……………………………………………….** 
   1. Organization ………………………………………………………….
   2. Size …………………………..……………………………………….
   3. Naming Conventions ………………………...………………………
   4. Headers ……………………………………...……………………….
   5. Indentations …………………………….…………………………….
   6. Comments …………………………………..……………….……….
4. **Statement and Programming Guidance ……..…………….** 
   1. Robustness ……………….…………………………………….…….
   2. Vectors and Arrays ………………………………………….……….
   3. Loops …………………………………………………………………
   4. Inputs and Outputs (I/O) ………………………………………..……
5. **Readability and Maintainability …..……………………….** 
   1. Maintainability with Functions and Attributes ……………...……….
   2. Readability ………………………………………………..………….
   3. Readability Items to Avoid ……………………………………..……
6. **Memory Allocation and Pointers …..……………………….** 
   1. Written Section 1 …………………………………………………….
   2. Written Section 2 …………………………………………………….
7. **Interoperability …..………………………………………….** 
   1. Portability …………………………………………………………….
   2. Inter-system Compatibility ………………………………….……….
8. **Features to Avoid in Fortran 90/95 …..…………………….** 
   1. Obsolete Features ..……………………………………..…………….
   2. Discouraged Elements ……………………………………………….

**6**

7

8

**9**

9

9

**10**

10

10

11

11

11

11

**12**

12

13

13

13

**13**

13

14

14

**14**

14

15

**15**

15

16

**16**

16

16

# **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

Specifically, the aim of this document is to ensure that new Fortran95 codes will be as portable and robust as possible, as well as consistent throughout the system. It builds upon commonly shared experience to avoid error-prone practices and gathers guidelines that are known to make codes more robust. This document covers items in order of decreasing importance (see below), deemed to be important for any code. It is recognized in the spirit of this standard that certain suggestions which make code easier to read for some people (e.g. lining up attributes, or using all lower/upper 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.

Moreover, this document deals exclusively with codes that are written in Fortran 95. It is also applicable to many, if not most, Fortran90 codes. It is not intended for this document to be applied to existing Fortran77 codes. This standard document is made as short as possible with the intent to be a quick reference document for those routinely working on Fortran 90/95 codes. As a consequence, the descriptions of why these standards were chosen, are sometimes assumed self-explanatory. For certain other standards, to-the-point descriptions are provided. For the same reasons, practical examples have been kept to a minimum. All comments, suggestions are continuously welcome as this standard document is envisioned to get updated regularly.

**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**

Five documents were used in creating this Fortran standard. Please refer to them for any clarification or additional information as needed.

P. Andrews (UKMO), G. Cats (KNMI/HIRLAM), D. Dent (ECMWF), M. Gertz (DWD), J. L. Ricard (Meteo France). ***European Standards for Writing and Documenting Exchangeable Fortran 90 Code*.** Crown Copyright, Version 1.1, updated 23 October 1996.

P. Van Delst. ***Fortran Coding Guidelines for CRTM*.**

W. Wolf et al., ***STAR F90/95 Coding Standard* [Draft].** developed for IASI, CrIS/ATMS projects.

***Fortan 95 Standards* [Working Draft]. Last updated October 21, 1997.** Found at http://j3- fortran.org/doc/year/97/97-007r2/pdf/97-007r2.pdf*.*

***UCAR Coding Standard for CCM4.*** Found athttp://www.cgd.ucar.edu/cms/ccm4/codingstandard.shtml.

**2. Language Features**

* 1. **Encouraged Practices and Features**

To increase robustness, it is recommended to compile the code with as many compilers as available, to minimize the errors that may be missed by using a single compiler.

***Guideline****: Use of the –ISO (International Standardization Organization) compiler option is encouraged.*

This ensures that no extension is used within the code.

***Guideline:*** *The following may aid in the robustness, readability, maintainability and portability of the code:*

* Encapsulation via the use of modules for procedures, functions, and data.
* Dynamic memory allocation for optimal memory usage.
* Derived types or structures which generally lead to stable interfaces, optimal memory usage, compactness, etc.
* Optional and keyword arguments in using routines.
* Definition of new operators, which helps readability and compactness.
* Functions/subroutines/operators overloading capability.
* Intrinsic functions: bits, arrays manipulations, kinds definitions, etc.

**2.2 Procedures**

***Standard***: *Use the save declaration where appropriate. Do not assume the value of the variable will be kept by the processor.*

***Standard****: Do not use an entry in a function subprogram.*

***Standard****: Functions must not have pointer results.*

***Guideline****: All dummy arguments, except pointers, should include the* INTENT *clause in their declaration.*

***Recommendation***: *Error conditions; when an error condition occurs inside a function/procedure, a message describing what went wrong should be printed.*

The name of the routine in which the error occurred must be included. It is acceptable to terminate execution within a package, but the developer may instead wish to return an error flag through the argument list.

***Recommendation***: *Functions/procedures that perform the same function but for different types/sizes of arguments, should be overloaded, to minimize duplication and ease the maintainability.*

***Recommendation****: When explicit interfaces are needed, use modules, or contain the subroutines in the calling programs (through* CONTAINS *statement), for simplicity.*

***Recommendation****: Do not use external routines (e.ge. subroutine not contained within a module and not within the* CONTAINS *statement of the main program).*

As in some cases, these functions need interface blocks that would need to be updated each time the interface of the external routine is changed.

**3. Formatting**

**3.1 Organization**

***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***: *Use free format syntax.*

***Standard****: Use modules to organize source code.*

**3.2 Size**

***Recommendation****: Limit to 80 the number of characters per line (maximum allowed under ISO is 132 characters).*

**3.3 Naming Conventions**

***Standard***: *Use meaningful, understandable names for variables and parameters.*

Recognized abbreviations are acceptable as a means of preventing very long variable names.

***Guideline***: *Use construct names to name loops, to increase readability, especially in nested loops.*

***Guideline****: Similarly, use construct names in subroutines, functions, main programs, modules, operator, interface, etc.*

***Recommendation****: Modules should be named the same name as the files they reside in.*

To simplify the makefiles, compile them. Consequently, multiple modules in a single file are to be avoided where possible.

**3.4 Headers**

***Standard****: Use the header guidelines and standards set in the General Programming Standards Document.*

***Standard****: Each externally-called function, subroutine, should contain a header.*

The content and style of the header should be consistent across the system, and should include the functionality of the function, as well as the description of the arguments, the author(s) names. A header could be replaced by a limited number of descriptive comments for small subroutines.

**3.5 Indentations**

***Standard****: Use consistent indentation across the code. Each level of indentation should use at least two spaces.*

***Recommendation****: Use the same indentation for comments as for the rest of the code.*

**3.6 Comments**

***Guideline****: Include comments to describe the input, output and local variables of all procedures.*

Grouping comments for similar variables is acceptable when their names are explicit enough.

***Guideline****: Use comments as required to delineate significant functional sections of code.*

**4. Statement and Programming Guidance**

**4.1 Robustness**

***Standard***: *Use* IMPLICIT NONE *in all codes (e.g. main programs, modules, etc.) to ensure correct size and type declarations of variables/arrays.*

***Standard***: *Use* PRIVATE *in modules before explicitly listing data, functions, procedures to be* PUBLIC.

This ensures encapsulation of modules and avoids potential naming conflicts. Exception to previous statement is when a module is entirely dedicated to PUBLIC data/functions (e.g. a module dedicated to constants).

***Standard***: *Initialize all variables and do* *not assume machine default value assignments.*

***Standard***: *Do not initialize variables of one type with values of another.*

***Guideline***: *Do not use the operators* == *and* /= *with floating-point expressions as operands.*

Check instead the departure of the difference from a pre-defined numerical accuracy threshold (e.g. epsilon comparison).

***Guideline***: *In mixed mode expressions and assignments, where variables of different types are mixed within an expression or in an assignment, type conversions should be written explicitly (e.g. not assumed).*

Do not compare expressions of different types for instance. Explicitly perform the type conversion first.

***Guideline***: *No include files should be used.*

Use modules instead, with USE statements in calling programs.

***Guideline***: *Structures (derived types) should be defined within their own module*.

Procedures, Functions to manipulate these structures should also be defined within this module, to form an object-like entity.

***Guideline***: *Procedures should be logically flat and should focus on a particular functionality, not several ones.*

***Guideline****: Module* PUBLIC *variables (e.g. global variables) should be used with care and mostly for static or infrequently varying data.*

***Recommendation***: *Use parentheses at all times to control evaluation order in expressions.*

***Recommendation***: *Use of structures is creates a more stable interface and a more compact design.*

Refer to structure contents with the % sign (e.g. Absorbents%WaterVapor).

***Guideline****: All subroutines not contained within a module should have a module interface to explicitly identify to the compiler the arguments, their types, dimensions, and whether they are input, output, (or both input and output) or optional.*

**4.2 Vectors and Arrays**

***Standard****: Subscript expressions should be of type integer only.*

***Standard****: When arrays are passed as arguments, code should not assume any particular passing mechanism.*

***Guideline****: Use of arrays is encouraged as well as intrinsic functions to manipulate them.*

***Guideline****: Use of assumed shapes is fine in passing vectors/arrays to functions/arrays.*

***Recommendation****: Declare* DIMENSION *for all non-scalars*

**4.3 Loops**

***Standard:***  *Do not use* GOTO *to exit/cycle loops, use instead* EXIT *or* CYCLE *statements.*

***Recommendation****: Do not use numbered* DO *loops (*DO 10 ...10 CONTINUE*).*

**4.4 Inputs and Outputs (I/O)**

***Standard****: I/O statements on external files should contain the status specifier parameters* err=, end=, iostat=*, as appropriate*

***Recommendation***: *Use write rather than print statements for non-terminal I/O.*

***Recommendation***: *Use Character parameters or explicit format specifiers inside the Read or Write statement.*

Do not use labeled format statements as they are outdated.

**5. Readability and Maintainability**

**5.1 Maintainability with Functions and Attributes**

***Guideline****: Use named parameters instead of “magic numbers”;* REAL, PARAMETER :: PI=3.14159, ONE=1.0

***Recommendation****: Functions, procedures, data that are linked and should be grouped in modules.*

***Recommendation****: Use of operators* <, >, <=, >=, ==, /= *is encouraged (for readability) instead of* .lt., .gt., .le., .ge., .eq., .ne.

***Recommendation****: Always use the* **::** *notation, even if there are no attributes.*

**5.2 Readability**

***Recommendation****: When writing new code, adhere to the style standards within your own coding style.*

When modifying an old code, adhere to the style of the existing code to keep consistency.

***Recommendation****: Use blanks to improve the appearance of the code, to separate syntactic elements (on either side of equal signs, etc) in type declaration statements*

***Recommendation****: Line up vertically: attributes, variables, comments within the variables declaration section.*

**5.3 Readability Items to Avoid**

***Guideline****: Do not use* FORTRAN *statements and intrinsic function names as symbolic names.*

***Guideline***: *Do not use* GOTO *statements.*

These are hard to maintain and complicate understanding the code. If absolutely necessary to use GOTO (if using other constructs complicates the code structure), thoroughly document the use of the GOTO.

***Recommendation****: Remove unused variables*

***Recommendation****: Remove code that was used for debugging once this is complete.*

**6. Memory Allocation and Pointers**

**6.1 Dynamic Memory Allocation**

***Standard****: Use of allocatable arrays is preferred to using pointers, when possible.*

To minimize risks of memory leaks and heap fragmentation.

***Guideline****: Always test the success of a dynamic memory allocation and deallocation.*

***Recommendation****: Use of dynamic memory allocation is encouraged. It makes code generic and avoids declaring with maximum dimensions.*

***Recommendation****: For simplicity, use Automatic arrays in subroutines whenever possible, instead of allocatable arrays.*

**6.2 Pointers**

***Standard***: *Use of pointers is allowed when declaring an array in a subroutine and making it available to a calling program.*

***Standard****: Always initialize pointer variables in their declaration statement using the* NULL()intrinsinc. INTEGER, POINTER :: x=> NULL()*.*

***Guideline****: Always deallocate allocated pointers and arrays. This is especially important inside subroutines and inside loops.*

**7. Interoperability**

**7.1 Portability**

***Standard****: Source code shall conform to the ISO Fortran95 standard.*

***Standard****: No compiler- or platform-dependent extensions shall be used.*

***Standard****: While use of* IOSTAT *and* STAT *is encouraged to handle I/O and other status errors, no interpretation of actual returned values shall be made, as these values are compiler-dependent.*

***Guideline***: *Do not use tab characters in the code to ensure it will look as intended when ported.*

*They are not part of the Fortran characters set.*

***Recommendation****: For applications requiring interaction with independently-developed frameworks, the use of* KIND *type for all variables declaration is encouraged to facilitate the integration*

**7.2 Intersystem Compatibility**

***Guideline***: *Note that* STOP *is a F90/95 standard.* EXIT(N) *is an extension and should be avoided.*

It is recognized that STOP does not necessarily return an error code. If an error code must be passed to a script for instance, then the extension EXIT could be used but within a central place, so that to limit its occurrences within the code to a single place

***Guideline***:Precision: *Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size.*

The F90/95 KIND feature should be used instead.

**8. Features to Avoid in Fortran 90/95**

**8.1 Obsolete Features**

***Recommendation****: No implicit changing of the shape of an array when passing it into a subroutine.*

Although actually forbidden in the standard it was very common practice in FORTRAN 77 to pass 'n' dimensional arrays into a subroutine where they would, say, be treated as a 1 dimensional array. This practice, though banned in Fortran 90, is still possible with external routines for which no Interface block has been supplied. This only works because of assumptions made about how the data is stored.

**8.2 Discouraged Elements**

***Standard****: Do not use Common blocks; modules are a better way to declare/store static data.*

They allow for the added ability to mix data of various types and to limit access to contained variables through use of the ONLY and PRIVATE clauses.

***Standard****: Do not use assigned and computed* GOTO*; use the* CASE *construct instead*

***Standard****: Do not use arithmetic* IF *statements; use the block* IF *construct instead*

***Guideline***: *Do not make use of the equivalence statement, especially for variables of different types.*

Use pointers or derived types instead an equivalence statement.