SPSRB Common Standards Working Group

Standards, Guidelines and Recommendations for Writing Fortran 95 Code

S.-A. Boukabara and P. Van Delst

Draft

Version 0.2D

September 26, 2007

STANDARDS, GUIDELINES AND RECOMMENDATIONS FOR WRITING FORTRAN 95 CODES

S.-A. Boukabara and P. van Delst

[1]NOAA/NESDIS/STAR
[2]SAIC @ NOAA/NCEP/EMC

Introduction:

The purpose of this document is to ensure that new Fortran95 code 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 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 this document are divided into three components; Standards, Guidelines and Recommendations (SGRs):

- *Standards:* Aimed at ensuring portability, readability and robustness. Compliance with this category is mandatory. Identified by the (**) sign.
- *Guidelines:* Good practices. Compliance with this category is strongly encouraged. The case for deviations will need to be argued by the programmer. Identified by the (*) sign.
- *Recommendations:* Compliance with this category is optional, but is encouraged for consistency purposes. No sign for these items.

Depending on the projects, leads may opt to adhere to all three levels or just the two first. All projects must adhere at least to the mandatory standards.

This document also lists a set of Fortran features that are not to be used in any new code. These features are known to be error-prone, difficult to maintain/read or listed as obsolescent in the Fortran95 standard. In many cases, newer Fortran95 constructs provide the same functionality in a more readable and robust fashion

Disclaimers:

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. Encouraged Features

These usually help in the robustness of the code (by checking interface compatibility for example) and in the readability, maintainability and portability. They are reminded here:

- Encapsulation: Use of modules for procedures, functions, 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.
- Intrinsinc functions: bits, arrays manipulations, kinds definitions, etc.

2. Portability / Inter-systems Compatibility

Standards:

- (**) Source code shall conform to the ISO Fortran95 standard.
- (**) No compiler- or platform-dependent extensions shall be used.
- (**) No use shall be made of compiler-dependent error specifier values (e.g. IOSTAT or STAT values).

Guidelines:

- (*) 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.
- (*) 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.
- (*) 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.

Recommendations:

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

3. Readability / Maintainability

Standards:

- (**) Use free format syntax.
- (**) Use consistent indentation across the code. Each level of indentation should use at least two spaces.
- (**) Use modules to organize source code.
- (**) Use meaningful, understandable names for variables and parameters. Recognized abbreviations are acceptable as a means of preventing variable names getting too long.
- (**) 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.

Guidelines:

- (*) Use construct names to name loops, to increase readability, especially in nested loops.
- (*) Similarly, use construct names in subroutines, functions, main programs, modules, operator, interface, etc.
- (*) 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.
- (*) Use comments as required to delineate significant functional sections of code.
- (*) Do not use FORTRAN statements and intrinsinc function names as symbolic names.
- (*) Use named parameters instead of "magic numbers"; REAL, PARAMETER :: PI=3.14159, ONE=1.0
- (*) 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.

Recommendations:

- 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.
- Use the same indentation for comments as for the rest of the code.
- Functions, procedures, data that are naturally linked should be grouped in modules.
- Limit to 80 the number of characters per line (maximum allowed under ISO is 132)
- Use of operators <, >, <=, >=, ==, /= is encouraged (for readability) instead of .lt., .gt., .le., .ge., .eq., .ne.
- Modules should be named the same name as the files they reside in: To simplify the makefiles that compile them. Consequently, multiple modules in a single file are to be avoided where possible.
- Use blanks to improve the appearance of the code, to separate syntactic elements (on either side of equal signs, etc) in type declaration statements
- Always use the :: notation, even if there are no attributes.
- Line up vertically: attributes, variables, comments within the variables declaration section.
- Remove unused variables
- Remove code that was used for debugging once this is complete.

4. Robustness

Standards:

- (**) Use Implicit NONE in all codes: main programs, modules, etc. To ensure correct size and type declarations of variables/arrays.
- (**) 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).
- (**) Initialize all variables. Do not assume machine default value assignments.
- (**) Do not initialize variables of one type with values of another.

Guidelines:

- (*) 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*).
- (*) In mixed mode expressions and assignments (where variables of different types are mixed), the type conversions should be written explicitly (not assumed). Do not compare expressions of different types for instance. Explicitly perform the type conversion first.
- (*) No include files should be used. Use modules instead, with USE statements in calling programs.
- (*) 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.
- (*) Procedures should be logically flat (should focus on a particular functionality, not several ones)
- (*) Module PUBLIC variables (global variables) should be used with care and mostly for static or infrequently varying data.

Recommendations:

- Use parentheses at all times to control evaluation order in expressions.
- Use of structures is encouraged for a more stable interface and a more compact design. Refer to structure contents with the % sign (e.g. Absorbents%WaterVapor).

5. Vectors and Arrays

Standards:

- (**) Subscript expressions should be of type integer only.
- (**) When arrays are passed as arguments, code should not assume any particular passing mechanism.

Guidelines:

- (*) Use of arrays is encouraged as well as intrinsinc functions to manipulate them.
- (*) Use of assumed shapes is fine in passing vectors/arrays to functions/arrays.

Recommendations:

- Declare DIMENSION for all non-scalars

6. Dynamic Memory Allocation / Pointers

Standards:

- (**) Use of allocatable arrays is preferred to using pointers, when possible. To minimize risks of memory leaks and heap fragmentation.
- (**) Use of pointers is allowed when declaring an array in a subroutine and making it available to a calling program.
- (**) Always initialize pointer variables in their declaration statement using the NULL() intrinsinc. INTEGER, POINTER :: x=> NULL()

Guidelines:

- (*) Always deallocate allocated pointers and arrays. This is especially important inside subroutines and inside loops.
- (*) Always test the success of a dynamic memory allocation and deallocation.

Recommendations:

- Use of dynamic memory allocation is encouraged. It makes code generic and avoids declaring with maximum dimensions.
- For simplicity, use Automatic arrays in subroutines whenever possible, instead of allocatable arrays.

7. Loops

Standards:

- (**) Do not use GOTO to exit/cycle loops, use instead EXIT or CYCLE statements.

Recommendations:

- No numbered DO loops (DO 10 ...10 CONTINUE).

8. Functions/Procedures

Standards:

- (**) Use the save declaration where appropriate. Do not assume the value of the variable will be kept by the processor.
- (**) Do not use an entry in a function subprogram.
- (**) Functions must not have pointer results.

Guidelines:

- (*) All dummy arguments, except pointers, should include the INTENT clause in their declaration.

Recommendations:

- 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.
- Functions/procedures that perform the same function but for different types/sizes of arguments, should be overloaded, to minimize duplication and ease the maintainability.
- When explicit interfaces are needed, use modules, or contain the subroutines in the calling programs (through CONTAINS statement), for simplicity.
- Do not use external routines as these need interface blocks that would need to be updated each time the interface of the external routine is changed.

9. Inputs/Outputs

Standards:

- (**) I/O statements on external files should contain the status specifier parameters err=, end=, iostat=, as appropriate.

Recommendations:

- Use write rather than print statements for non-terminal I/O.
- Use Character parameters or explicit format specifiers inside the Read or Write statement. DO not use labeled format statements (outdated).

Fortran Features that are obsolescent and/or discouraged:

Standards:

- (**) No Common blocks. Modules are a better way to declare/store static data, with the added ability to mix data of various types, and to limit access to contained variables through use of the ONLY and PRIVATE clauses.
- (**) No assigned and computed GO TOs - use the CASE construct instead
- (**) No arithmetic IF statements - use the block IF construct instead

Guidelines:

- (*) Do not make use of the equivalence statement, especially for variables of different types. Use pointers or derived types instead.

Recommendations:

- No implicitly changing 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.

References:

- PennState University (PSU) Center for Environmental Informatics Database Fortran-77 Coding Standard
- UCAR Coding Standard for CCM4
- European Standards for Writing and Documenting Exchangeable Fortran 90 Code.
- Draft version of the STAR F90/95 Coding Standard developed for IASI, CrIS/ATMS projects [W. Wolf et al.]
- Fortran Coding Guidelines for CRTM [P. Van Delst]