

Fortran coding standards for new JULES code

Written by Matt Pryor

Last updated 08 June 2010

Table of contents

1	Introduction.....	2
1.1	Why have standards?	2
1.2	Units.....	2
2	Guidelines for Fortran code	3
2.1	Layout and formatting.....	3
2.2	Style	5
2.3	Fortran features	8
2.4	Floating-point arithmetic	11
2.4.1	Comparing real numbers.....	11
2.4.2	Non-distributive arithmetic.....	13
2.5	Further code guidance and best practices	13
3	Standard code templates	14
3.1	Code templates.....	14
3.2	Copyright notice.....	14

1 Introduction

This document specifies the software standards and coding styles to be used when writing new code files for JULES. When making extensive changes to an existing file, a rewrite should be done to ensure that the routine meets the JULES standard and style.

1.1 Why have standards?

This document is intended for new as well as experienced programmers, so a few words about why there is a need for software standards and styles may be in order.

Coding standards specify a standard working practice for a project with the aim of both reducing portability and maintainability problems and improving the readability of code. This process makes code development and reviewing easier for all developers involved in the project. Remember that software should be written for people and not just for computers! As long as the syntax rules of the programming language (e.g. Fortran 90) are followed, the computer does not care how the code is written. You could use archaic language structures, add no comments, leave no spaces etc. However, another programmer trying to use, maintain or alter the code will have trouble working out what the code does and how it does it. A little extra effort whilst writing the code can greatly simplify the task of this other programmer (which might be the original author a year or so after writing the code, when details of it are bound to have been forgotten). In addition, following these standards may well help you to write better, more efficient, programs containing fewer bugs.

1.2 Units

All routines and documentation must be written using SI units. Standard SI prefixes may be used. Where relevant, the units used must be clearly stated in both code and documentation.

2 Guidelines for Fortran code

These are guidelines you should adhere to when you are developing new code for inclusion in the official release of JULES. If you are modifying existing code, you should adhere to its existing standard and style where possible. If you want to change its standard and style, you should seek prior agreement with the JULES office. Where possible, you should try to maintain the same layout and style within a source file.

When reading these guidelines, it is assumed that you already have a good understanding of modern Fortran terminology. It is understood that these guidelines may not cover every aspect of your work. In such cases, use common sense and always bear in mind that other people may have to maintain the code in the future.

Always test your code before releasing it. Do not ignore compiler warnings, as they may point you to potential problems.

Some standard templates are given in Section 3.1 of this document.

2.1 Layout and formatting

The following is a list of recommended practices for layout and formatting when you write code in Fortran.

- Use the Fortran 90 free format syntax.
- Indent blocks by 2 characters. Where possible, comments should be indented with the code within a block.
- Use space and blank lines where appropriate to format your code to improve readability (use genuine spaces rather than tabs, as the `tab` character is not in the Fortran character set). For example:

Common practice

```
DO i=1,n
    a(i)%c=10*i/n
ENDDO

IF (this==that) THEN
    distance=0
    time=0
ENDIF
```

Better approach

```
DO i = 1, n
    a(i)%c = 10 * i / n
END DO

IF (this == that) THEN
    distance = 0
    time      = 0
END IF
```

- Try to confine your line width to 80 characters. This means that your code can be viewed easily in any editor on any screen, and can be printed easily on A4 paper.
- Line up your statements, where appropriate, to improve readability. For example:

Common practice

```
REAL, INTENT(IN) :: my_in(:)
REAL, INTENT(INOUT) :: my_inout(:)
REAL, INTENT(OUT) :: my_out(:)
! ...
CHARACTER(LEN=256) :: my_char
! ...
my_char = 'This is a very very very very very very ' // &
        'very very very very very very very very very ' // &
        'long character assignment'
```

Better approach

```
REAL, INTENT(IN) :: my_in(:)
REAL, INTENT(INOUT) :: my_inout(:)
REAL, INTENT(OUT) :: my_out(:)
! ...
CHARACTER(LEN=256) :: my_char
! ...
my_char                                     &
    = 'This is a very very very very very very ' &
    // 'very very very very very very very very ' &
    // 'long character assignment'
```

- Short and simple Fortran statements are easier to read and understand than long and complex ones. Where possible, avoid using continuation lines in a statement.
- Avoid putting multiple statements on the same line. It is not good for readability.
- Each program unit (module, subroutine, function etc.) should follow a structure similar to the templates supplied with this document. The intended behaviour of the unit should be clearly described in the header.
- Comments should start with a single '!' at beginning of the line. A blank line should be left before (but not after) the comment line. The only exception is for one line comments which can be indented within the code or placed after the statement.
- Each subroutine, function and module should be in a separate file. Modules may be used to group related variables, subroutines and functions.

2.2 Style

The following is a list of recommended styles when you write code in Fortran.

- New code should be written using Fortran 95 syntax. Avoid un-portable vendor/compiler extensions. Avoid Fortran 2003 features for the moment, as they will not become widely available in the near future (however, there is no harm in designing your code with the future in mind. For example, if there is a feature that is not in Fortran 95 and you know that it is in Fortran 2003, you may want to write your Fortran 95 code to make it easier for the future upgrade).
- Write your program in UK English, unless you have a very good reason for not doing so. Write your comments in simple UK English and name your program units and variables based on sensible UK English words. Always bear in mind that your code may be read by people who are not proficient English speakers.
- When naming your variables and program units, always keep in mind that Fortran is a case-insensitive language (e.g. *EditOrExit* is the same as *EditorExit*.)
- Use only characters in the Fortran character set. In particular, accent characters and tabs are not allowed in code, although they are usually OK in comments. If your editor inserts tabs automatically, you should configure it to switch off the functionality when you are editing Fortran source files.
- Although Fortran has no reserved keywords, you should avoid naming your program units and variables with names that match an intrinsic `FUNCTION` or `SUBROUTINE`. Similarly, you should avoid naming your program units and variables with names that match a *keyword* in a Fortran statement.
- Be generous with comments. State the reason for doing something, instead of repeating the Fortran logic in words.
- To improve readability, write your code using the *ALL CAPS Fortran keywords* approach. This is the style used in most of the examples in this document, where Fortran keywords and intrinsic procedures are written in ALL CAPS. The rest of the code is written in either lowercase with underscores or CamelCase. This approach has the advantage that Fortran keywords stand out.
- Use the new and clearer syntax for LOGICAL comparisons, i.e.:
 - `==` instead of `.EQ.`
 - `/=` instead of `.NE.`
 - `>` instead of `.GT.`
 - `<` instead of `.LT.`
 - `>=` instead of `.GE.`
 - `<=` instead of `.LE.`

- Where appropriate, simplify your LOGICAL assignments, for example:

Common practice

```

IF (my_var == some_value) THEN
    something      = .TRUE.
    something_else = .FALSE.
ELSE
    something      = .FALSE.
    something_else = .TRUE.
END IF

! ...

IF (something .EQV. .TRUE.) THEN
    CALL do_something()
    ! ...
END IF

```

Better approach

```

something      = (my_var == some_value)
something_else = (my_var /= some_value)

! ...

IF (something) THEN
    CALL do_something()
    ! ...
END IF

```

- Positive logic is usually easier to understand. When using an IF-ELSE-END IF construct you should use positive logic in the IF test, provided that the positive and the negative blocks are about the same length. It may be more appropriate to use negative logic if the negative block is significantly longer than the positive block. For example:

Common practice

```

IF (my_var != some_value) THEN
    CALL do_this()
ELSE
    CALL do_that()
END IF

```

Better approach

```

IF (my_var == some_value) THEN
    CALL do_that()
ELSE
    CALL do_this()
END IF

```

- To improve readability, you should always use the optional space to separate the following Fortran keywords:

```
ELSE IF      END DO      END FORALL    END FUNCTION
END IF      END INTERFACE  END MODULE  END PROGRAM
END SELECT   END SUBROUTINE END TYPE     END WHERE
SELECT CASE
```

- If you have a large or complex code block embedding other code blocks, you may consider naming some or all of them to improve readability.
- If you have a large or complex interface block or if you have one or more sub-program units in the `CONTAINS` section, you can improve readability by using the full version of the `END` statement (i.e. `END SUBROUTINE <name>` or `END FUNCTION <name>` instead of just `END`) at the end of each sub-program unit. For readability in general, the full version of the `END` statement is recommended over the simple `END`.
- Where possible, consider using `CYCLE`, `EXIT` or a `WHERE`-construct to simplify complicated `DO`-loops.
- When writing a `REAL` literal with an integer value, put a `0` after the decimal point (i.e. `1.0` as opposed to `1.`) to improve readability.
- Where reasonable and sensible to do so, you should try to match the names of dummy and actual arguments to a `SUBROUTINE/FUNCTION`.
- In an array assignment, it is recommended that you use array notations to improve readability. E.g.:

Common practice

```
INTEGER :: array1(10, 20), array2(10, 20)
INTEGER :: scalar
! ...
array1 = 1
array2 = array1 * scalar
```

Better approach

```
INTEGER :: array1(10, 20), array2(10, 20)
INTEGER :: scalar
! ...
array1(:, :) = 1
array2(:, :) = array1(:, :) * scalar
```

- Where appropriate, use parentheses to improve readability. E.g.:
`a = (b * i) + (c / n)` is easier to read than `a = b * i + c / n`.

2.3 Fortran features

The following is a list of Fortran features that you should use or avoid.

- Use `IMPLICIT NONE` in all program units. This forces you to declare all your variables explicitly. This helps to reduce bugs in your program that will otherwise be difficult to track.
- Design any derived data types carefully and use them to group related variables. Appropriate use of derived data types will allow you to design modules and procedures with simpler and cleaner interfaces.
- Where possible, module variables and procedures should be declared `PRIVATE`. This avoids unnecessary export of symbols, promotes data hiding and may also help the compiler to optimise the code.
- When you are passing an array argument to a `SUBROUTINE/FUNCTION`, and the `SUBROUTINE/FUNCTION` does not change the `SIZE/DIMENSION` of the array, you should pass it as an assumed shape array. Memory management of such an array is automatically handled by the `SUBROUTINE/FUNCTION`, and you do not have to worry about having to `ALLOCATE` or `DEALLOCATE` your array. It also helps the compiler to optimise the code.
- Use an array `POINTER` when you are passing an array argument to a `SUBROUTINE`, and the `SUBROUTINE` has to alter the `SIZE/DIMENSION` of the array. You should also use an array `POINTER` when you need a dynamic array in a component of a derived data type. (Note: Fortran 2003 allows passing `ALLOCATABLE` arrays as arguments as well as using `ALLOCATABLE` arrays as components of a derived data type. Therefore, the need for using an array `POINTER` should be reduced once Fortran 2003 becomes more widely accepted.)
- Where possible, an `ALLOCATE` statement for an `ALLOCATABLE` array (or a `POINTER` used as a dynamic array) should be coupled with a `DEALLOCATE` within the same scope. If an `ALLOCATABLE` array is a `PUBLIC MODULE` variable, it is highly desirable if its memory allocation and deallocation are only performed in procedures within the `MODULE` in which it is declared. You may consider writing specific `SUBROUTINES` within the `MODULE` to handle these memory managements.
- To avoid memory fragmentation, it is desirable to `DEALLOCATE` in reverse order of `ALLOCATE`.

Common practice

```
ALLOCATE (a(n))
ALLOCATE (b(n))
ALLOCATE (c(n))
! ... do something ...
DEALLOCATE (a)
DEALLOCATE (b)
DEALLOCATE (c)
```


Better approach

```
ALLOCATE (a(n))
ALLOCATE (b(n))
ALLOCATE (c(n))
! ... do something ...
DEALLOCATE (c)
DEALLOCATE (b)
DEALLOCATE (a)
```

- Always define a `POINTER` before using it. You can define a `POINTER` in its declaration by pointing it to the intrinsic function `NULL()`. Alternatively, you can make sure that your `POINTER` is defined or nullified early on in the program unit. Similarly, `NULLIFY` a `POINTER` when it is no longer in use, either by using the `NULLIFY` statement or by pointing your `POINTER` to `NULL()`.
- Avoid the `DIMENSION` attribute or statement. Declare the `DIMENSION` with the declared variables. E.g.:

Common practice

```
INTEGER, DIMENSION(10) :: array1
INTEGER                  :: array2
DIMENSION                :: array2(20)
```

Better approach

```
INTEGER :: array1(10), array2(20)
```

- Avoid `COMMON` blocks and `BLOCK DATA` program units. Instead, use a `MODULE` with `PUBLIC` variables.
- Avoid the `EQUIVALENCE` statement. Use a `POINTER` or a derived data type, and the `TRANSFER` intrinsic function to convert between types.
- Avoid the `PAUSE` statement, as your program will hang in a batch environment. If you need to halt your program for interactive use, consider using a `READ*` statement instead.
- Avoid the `ENTRY` statement. Use a `MODULE` or internal `SUBROUTINE`.
- Avoid the `GOTO` statement. The only commonly acceptable usage of `GOTO` is for error trapping. In such case, the jump should be to a commented `9999 CONTINUE` statement near the end of the program unit. Typically, you will only find error handlers beyond the `9999 CONTINUE` statement.
- Never use a `GOTO` to jump upwards in the code.
- Any `GOTO` must be commented to explain why it is there and what it is doing.
- Avoid assigned `GOTO`, computed `GOTO`, arithmetic `IF`, etc. Use the appropriate modern constructs such as `IF`, `WHERE`, `SELECT CASE`, etc..
- Avoid numbered statement labels. `DO ... label CONTINUE` constructs should be replaced by `DO ... END DO` constructs. Every `DO` loop must be terminated with a corresponding `END DO`.

- Never use a `FORMAT` statement - they require the use of labels, and obscure the meaning of the I/O statement. The formatting information can be placed explicitly within the `READ`, `WRITE` or `PRINT` statement, or be assigned to a `CHARACTER` variable in a `PARAMETER` statement in the header of the routine for later use in I/O statements. Never place output text within the format specifier, i.e. only format information may be placed within the `FMT=` part of an I/O statement. All variables and literals, including any character literals, must be 'arguments' of the I/O routine itself. This improves readability by clearly separating what is to be read/written from how to read/write it.
- Avoid the `FORALL` statement/construct. Despite what it is supposed to do, `FORALL` is often difficult for compilers to optimise (see, for example, [Implementing the Standards including Fortran 2003](#) by NAG). Stick to the equivalent `DO` construct, `WHERE` statement/construct or array assignments unless there are actual performance benefits from using `FORALL`.
- A `FUNCTION` should be `PURE`, i.e. it should have no side effects (e.g. altering an argument or module variable, or performing I/O). If you need to perform a task with side effects, you should use a `SUBROUTINE` instead.
- Declare the `INTENT` of all arguments to a subroutine or function. This allows checks against unintended access of variables to be done at compile time. The above point requiring functions to be pure means that all arguments of a `FUNCTION` should be declared as `INTENT(IN)`.
- Avoid `RECURSIVE` procedures if possible. `RECURSIVE` procedures are usually difficult to understand, and are always difficult to optimise in a supercomputer environment.
- Avoid using the specific names of intrinsic procedures. Use the generic names of intrinsic procedures where possible.
- Use the `ONLY` clause in a `USE <module>` statement to declare all imported symbols (i.e. parameters, variables, functions, subroutines, etc). This makes it easier to locate the source of each symbol, and avoids unintentional access to other `PUBLIC` symbols within the `MODULE`.
- Function or subroutine arguments should be declared separately from local variables.
- The standard delimiter for namelists is `\.`. In particular, note that `&END` is non-standard and should be avoided.
- The use of operator overloading is discouraged, as it can lead to confusion. The only acceptable use is to allow the standard operators (`+`, `-` etc.) to work with derived data types, where this makes sense.
- Avoid using archaic Fortran 77 features and features deprecated in Fortran 90.

2.4 Floating-point arithmetic

When writing scientific code, it is important to understand the differences between normal arithmetic and the floating-point arithmetic used by computers. Due to the limited precision available to represent real numbers, many things that are true for normal arithmetic no longer hold in floating-point arithmetic. Special care must also be given to treating variables in a way that is physically realistic, not just mathematically correct. Failure to do so can result in a model that is over-sensitive to both changes in computing platform and small perturbations in initial conditions.

2.4.1 Comparing real numbers

A common place for errors of this kind to arise is when comparing real numbers to each other. Consider the following code:

```
IF ( snow_tile > 0.0 ) THEN
    ...
ELSE
    ...
END IF
```

This code is meant to be modelling two different physical situations – the case where snow is present on a tile and the case where it is not. You may expect that a tile with no snow would have `snow_tile` set to 0. However, due to differences in compilers, optimisation options and floating-point rounding errors, `snow_tile` could end up being tiny (say 10^{-20}) rather than 0. This snow amount is physically negligible, so we would want the model to behave as if there is no snow. This is not the case, however, since 10^{-20} is greater than 0. This kind of unphysical branching can lead to significant errors.

Other than avoiding this kind of branching `IF` statement entirely, there are two ways around this problem:

1. Specify a physically realistic tolerance level, e.g.

```
IF ( snow_tile > tolerance ) THEN
    ...
ELSE
    ...
END IF
```

where `tolerance` is some small value greater than 0.

One problem with this solution is that you may end up with several different tolerances defined in various places in the code. For this reason, `tolerance` should be obtained using the Fortran intrinsic functions `EPSILON` (`EPSILON(X)` returns a nearly negligible number relative to 1) or `TINY` (`TINY(X)` returns the smallest positive (non zero) number of the same type as `x` that the computer can represent) where possible. If the values provided by these intrinsic functions are inappropriate, the tolerance should be declared as a `PARAMETER` in the variable declarations section of the programming unit.

When defining a value for `tolerance`, bear in mind that if the value is too large it could lead to problems with conservation of variables (e.g. water, energy, carbon).

2. Set the variable in question to 0.0 whenever it makes sense physically, e.g.

```
IF ( snow_tile should be 0 physically ) THEN
    snow_tile = 0.0
END IF
```

```
IF ( snow_tile > 0.0 ) THEN
    ...
ELSE
    ...
END IF
```

This solution avoids the problem of having several different tolerances defined in the code. However, depending on how the condition ‘snow_tile should be 0 physically’ is defined, it could lead to problems with conservation of variables (in this case, water, due to loss of snow).

Which solution to use is highly dependent on the particular problem, and requires careful thought. In general, the 2nd solution is preferable, since it avoids having several different tolerances defined in the code.

Although the above conversation focuses on comparing values to zero, the same concepts apply in the general case of comparing any real number to any other. For example:

Common practice

```
IF ( real1 > real2 ) THEN
    ...
ELSE
    ...
END IF
```

Better approach

```
IF ( real1 - real2 > tolerance ) THEN
    ...
ELSE
    ...
END IF
```

where, as above, `tolerance` is some suitably small number.

The same concerns about rounding errors must also be considered when comparing two real numbers for equality, e.g.

Common practice

```
IF ( real1 == real2 ) THEN
    ...
END IF
```

Better approach

```
IF ( ABS(real1 - real2) < tolerance ) THEN
    ...
END IF
```

where, again, `tolerance` is some suitably small number. Such concerns with rounding errors do not apply to `INTEGER` and `LOGICAL` comparisons, hence the form `value1 == value2` can be used for these.

2.4.2 Non-distributive arithmetic

Scientific programmers should also be aware that some algebraic identities that hold for normal arithmetic no longer hold in floating-point arithmetic, mainly due to rounding errors. In particular, unlike normal arithmetic, the order in which calculations are performed will affect the answer given. For example, the following statements are true for normal arithmetic, but *do not hold in floating-point arithmetic*:

```
x + (y + z) ≡ (x + y) + z
x * (y * z) ≡ (x * y) * z
x * (y / z) ≡ (x * y) / z
```

This can lead to extremely subtle errors that are hard to spot, so developers should bear this in mind at all times. Trying to enforce the order of calculation using brackets will not necessarily solve this problem, since the compiler will make decisions for itself depending on optimisation options, etc.

A more in depth discussion of these kinds of rounding errors can be found in the accompanying document ‘Dealing with rounding issues’ from the Met Office UM perturbation sensitivity project. The document also includes some real world examples of rounding issues from the Met Office Unified Model.

2.5 Further code guidance and best practices

- Avoid the use of ‘magic numbers’. A ‘magic number’ is a numeric constant that is hard wired into the code. These are very hard to maintain and obscure the function of the code. It is much better to assign the ‘magic number’ to a variable or constant with a meaningful name and use this throughout the code. In many cases the variable may be best placed in a module. This ensures that all subroutines will use the correct value of the numeric constant and that alteration of it in one place will be propagated to all its occurrences. For example:

Common practice

```
IF (ObsType == 3) THEN
```

Better approach

```
INTEGER, PARAMETER :: SurfaceWind = 3 ! No. for surface wind
...
IF (ObsType == SurfaceWind) THEN
```

- Write well structured code making use of subroutines to separate specific subtasks. In particular all file I/O should be done through subroutines. This greatly facilitates the portability of the code. Subroutines should be kept

reasonably short, but don't forget there are start up overheads involved in calling an external subroutine, so they should do a reasonable amount of work.

- If you find yourself copying and pasting the same code, consider making it a `SUBROUTINE` or `FUNCTION` that can be called from the different places in the code.
- Any code that introduces new physics to JULES should have a switch to enable it to be turned off. This makes it possible to run the model in a configuration that is identical to the model before the new physics went in, in order to check that nothing unexpected has been broken.
- Code should be accompanied by technical documentation describing the physical processes that the additional code is intended to model and how this is achieved. Any equations used should be documented (in their continuous form where appropriate) along with the methods used to discretise these equations. In the case where new subroutines/functions have been added, a calling tree should be included. Any changes to the JULES control file should also be clearly documented.

The most important thing is to *always* bear in mind that somebody will have to maintain your code in the future. That person could even be you several years later! Make sure you comment code thoroughly and use some common sense where procedures are not clear from this document.

3 Standard code templates

Some standard templates for copyright notices and code files are provided as text files with this document. When writing new code, these templates should be used as a guide for the structure of the new files.

3.1 Code templates

Templates are provided for a `SUBROUTINE`, a `FUNCTION` and a simple `MODULE`. Text in angle brackets provides placeholders for Fortran statements.

3.2 Copyright notice

Two different copyright notices are provided, one of which should be included at the top of each source file that you write. Met Office contributors should use the Crown Copyright notice, while other developers should use the Consortium Member Copyright notice.