Joint Center for Satellite Data Assimilation

CRTM: Fortran95 Coding Guidelines

Paul van Delst[a]
JCSDA/EMC/SAIC

January, 2008

---

[a]paul.vandelst@noaa.gov

1 Introduction

The reason for putting this document together, apart from establishing some minimum standard for code quality from developers outside EMC or JCSDA, is to provide a basis for consistency amongst the many CRTM developers. That is, some guidelines that allow for personal styles and preferences but still supply the visual cues that allow developers to easily read code from disparate sources.

Most of the controversial items (as far these things are controversial in the grand scheme of the world) tend to be those suggestions that make code easier to read for some people, e.g. lining up attributes within variable declaration blocks, or using all lowercase, or mixed case, etc. These sorts of items will always be subjective - what is intended is to mould the guidelines below to satisfy as many people's predilections as possible so as to maintain a "consistent look and feel" of the code.

One thing to remember is that other people will also be looking at, reading, and trying to understand your code - be nice to them.

2 Naming Conventions

- Identify derived data types and associated procedures[1], or a logical group of module procedures (as well as the containing module), with appropriate prefixes or suffixes to minimise namespace clashes.
- All tangent-linear variables and procedures should be suffixed with "`_TL`" and all adjoint variables and procedures should be suffixed with "`_AD`".

3 Style

- Use free format syntax.
- Indentation: begin in first column for statements such as `PROGRAM`, `MODULE` and `CONTAINS`, and recursively indent all subsequent blocks by at least two spaces.
- Do not use tab characters - they are not part of the Fortran character set.
- Name `END`s fully, including the program unit name.
- When creating new code (this includes refactoring[2] old code), use the style guidelines above within the context of your personal style. However, keep in mind that others will have to read your code. If you use a syntax sensitive editor, as an experiment, turn off the syntax colouring to see if your code is still easily readable.
- When modifying old code, adhere to the style of the existing code.

4 Comments and Documentation

- For cryptic variable names, state description in a comment immediately preceding declaration or on end of the declaration line. Better yet, try not to use cryptic variable names.
- For procedures and modules, insert a contiguous documentation header immediately preceding its declaration containing a *brief* overview followed by an optional detailed description.

[1]The word "procedure" is used as a catch all to refer to both subroutines and functions.
[2]Refactoring involves improving the design of existing code. It doesn't change the observable behaviour of the software; it improves its internal structure. Refactoring does not fix bugs or add new functionality. See http://en.wikipedia.org/wiki/Refactoring or Fowler,M., "Refactoring", 2000, Addison-Wesley.

- Procedure argument documentation in the documentation header should briefly describe what are the arguments and their units. In some cases, this level of documentation may be unnecessary (e.g. the arguments to a generic interpolation procedure.) If in doubt, err on the side of documenting the argument list.
- Ensure procedure argument documentation in the documentation header is consistent with additions and/or deletions from the calling list.
- Do not document changes within the code with comments that include the user's name or initials.
- Document any modifications made by using a short, but descriptive, log message when checking the modified code into the software repository. Don't just say *what* has changed - since differencing versions provides that information - but *why*.

5 Variable Declarations

- Use meaningful, understandable names for variables and parameters. That is, rather than,

  ```
  INTEGER, PARAMETER :: NN = 10
  REAL :: CldP
  ```

  use

  ```
  INTEGER, PARAMETER :: N_LAYERS = 10
  REAL :: CloudPressure
  ```

  At the same time, recognize that common programming idioms should be used. For example, it is common to use single letter variable names such as `i` as an array loop index counter rather than a variable named `LoopIndex`.
- Declare the kind for all reals, including literal constants, by using a kind definition module.
- Do not use Fortran intrinsic function names for variable names.
- Declare `INTENT` on all dummy arguments.
- Line up attributes within variable declaration blocks.
- Any scalars used to define extent must be declared prior to use.
- Declare a variable name only once in a scope, including `USE MODULE` statements.

6 Modules

- Use modules to group related procedures and/or shared data.
- Use the `ONLY` attribute on `USE` statements as required.
- Declare `IMPLICIT NONE`.
- Include a `PRIVATE` statement and explicitly declare public entities.

7 Subroutine and Functions

- Group all dummy argument declarations first, followed by local variable declarations.
- Declare `INTENT` on all dummy arguments.
- All subroutines and functions should be contained within a module. Using module procedures rather than external procedures ensures that the procedure interface is explicit and checked for consistency during compilation[3].
- To avoid null or undefined pointers, pointers passed through an argument list must be allocated.
- Functions must not have pointer results.

8 Control Constructs

- Name control constructs (e.g., `DO`, `IF`, `SELECT CASE`) which span a significant number of lines or form nested code blocks.
- No numbered do-loops.
- Name loops that contain `CYCLE` or `EXIT` statements.
- Use `CYCLE` or `EXIT` rather than `GOTO`.
- Use Fortran95-style relational symbols, e.g., `>=` rather than `.GE.`, `/=` rather than `.NE.`.
- For multiple selection tests, use case statements with case defaults rather than if-constructs wherever possible. For example, rather than

  ```
  IF (i == 1) THEN
    ...
  ELSE IF (i == 2) THEN
    ...
  ELSE IF (i == 3) THEN
    ...
  ELSE
    ...
  END IF
  ```

  use

  ```
  SELECT CASE(i)
    CASE(1)
      ...
    CASE(2)
      ...
    CASE(3)
      ...
    CASE DEFAULT
      ...
  END SELECT
  ```

[3]Note that one can take advantage of explicit interfaces for external procedures by using an interface block, but doing so means any changes to the external procedure interface also requires the interface block to be updated.

9 Miscellaneous

- Write only standard conforming Fortran95. Do not use, if at all possible, compiler specific features, functions or subroutine calls. Doing so limits portability of the code. If compiler specific features must be used, localise the impact by wrapping the compiler extensions within a generic procedure and call that generic procedure. Document the potential portability problem in the calling code.
- Always use a kind definition module with parameterized kind types.
- Literal kind types shall not be used. For example, do not do the following:

  ```
  REAL(4) :: x
  REAL(8) :: y
  x = 1.0_4
  y = 2.0_8
  ```

  The above example is a common way to specify single and double precision reals (the 4 and 8 kind type values being synonymous with the common Fortran77 extension *4 and *8). However, kind type values are not portable. Some compilers use a kind value of 1 for single precision and 2 for double precision reals. Always use a kind definition module with parameterized kind types.
- Do not use "magic numbers", i.e. literal constants, in variable assignments or expressions. Use named parameters. For example, rather than

  ```
  USE Type_Kinds, ONLY: rk
  REAL(rk) :: ppmv
  REAL(rk) :: Mixing_Ratio
  REAL(rk) :: Molecular_Weight
  ...
  ppmv = 1.0e+03*Mixing_Ratio*28.9648/Molecular_Weight
  ```

  consider the following,

  ```
  USE Type_Kinds, ONLY: rk
  REAL(rk), PARAMETER :: G_TO_KG      = 1.0e-03_rk
  REAL(rk), PARAMETER :: PPV_TO_PPMV  = 1.0e+06_rk
  REAL(rk), PARAMETER :: SCALE_FACTOR = G_TO_KG * PPV_TO_PPMV
  REAL(rk), PARAMETER :: MW_DRYAIR    = 28.9648_rk
  ...
  REAL(rk) :: ppmv
  REAL(rk) :: Mixing_Ratio
  REAL(rk) :: Molecular_Weight
  ...
  ppmv = SCALE_FACTOR*Mixing_Ratio*MW_DRYAIR/Molecular_Weight
  ```

  Now all of the "magic numbers" in the expression have been replaced with named parameter constants. Simply looking at the code tells us what the scaling factor is for and that the other number is actually the molecular weight of dry air.
- Always use the kind type when defining and assigning real literal constant parameters. See example above. Note the suffix `_rk` on all the literal constants in the parameter definitions. This ensures that the literal constant has the same precision as its data type.
- Always initialise pointer variables in their declaration statement using the `NULL()` intrinsic, e.g.

  ```
  INTEGER, POINTER :: x => NULL()
  ```

- Use modules for sharing large segments of data.
- Remove unused variables.
- Do not use common blocks or includes for new code.
- Always use generic, not specific, intrinsic functions, e.g. `COS` rather than `DCOS`.
- Remove code that was used for debugging purposes once the debugging is complete.

10 Revision Control Information Access

This section is here for those cases where software revision control information needs to be accessed dynamically; for example, if the code history is to be written into a data file. This allows the history of the data file contents to be associated with particular code releases.

- In programs, define a character parameter named `PROGRAM_RCS_ID` containing the CVS/svn[4] `$Id:$` tag.
- In modules, define a private character parameter named `MODULE_RCS_ID` containing the CVS/svn `$Id:$` tag.
- In subroutines and functions, include an `OPTIONAL, INTENT(OUT)` argument `RCS_Id` that, if supplied, returns the module CVS/svn `$Id:$`.

[4]CVS (Concurrent Version System) and svn (Subversion) are two popular, open-source software configuration management (SCM) tools that recognise and expand RCS (Revision Control System) keywords (such as `$Id:$`, `$Date:$`, `$Revision:$`, etc.) in source code. In the character parameters `PROGRAM_RCS_ID` and `MODULE_RCS_ID`, and the optional argument `RCS_Id`, the acronym RCS is used in a generic sense, rather than in reference to the Revision Control System SCM tool.

A Example Code

Below is some example code demonstrating the implementation of the coding guidelines. Note that the example is not meant to represent the only acceptable style (with capitalized Fortran statements and specifiers). As one reviewer of this document stated: “Variations in use of capitalization are just too numerous for users to agree on a single standard”. In addition, regardless of your personal coding style, recall the guideline in the style section: When modifying old code, adhere to the style of the existing code.

```
! Define the kinds to use for integers and reals
! including generic kinds to allow simple alteration
! of required precisions.

MODULE Type_Kinds

  ! No implicit typing
  IMPLICIT NONE

  ! Explicit visibility declaration
  PRIVATE
  PUBLIC :: Byte, Short, Long
  PUBLIC :: Single, Double
  PUBLIC :: ik, rk

  ! Integer kinds
  INTEGER, PARAMETER :: Byte  = SELECTED_INT_KIND(1)  ! Byte  integer
  INTEGER, PARAMETER :: Short = SELECTED_INT_KIND(4)  ! Short integer
  INTEGER, PARAMETER :: Long  = SELECTED_INT_KIND(8)  ! Long  integer

  ! Floating point kinds
  INTEGER, PARAMETER :: Single = SELECTED_REAL_KIND(6)  ! Single precision
  INTEGER, PARAMETER :: Double = SELECTED_REAL_KIND(15) ! Double precision

  ! Generic kinds
  INTEGER, PARAMETER :: ik = Long    ! Generic integer kind
  INTEGER, PARAMETER :: rk = Double  ! Generic real kind

END MODULE Type_Kinds


! A pretend module containing forward, tangent-linear
! and adjoint component subroutines.
!
! You may also want to list information about what
! procedures are available from this module, and
! what other dependencies this module has, i.e.
! other module usage.

MODULE My_Module

  ! Only use the required entities of a USEd module
  USE Type_Kinds, ONLY: rk

  ! No implicit typing
  IMPLICIT NONE
```

```fortran
  ! Explicit visibility declaration
  PRIVATE
  PUBLIC :: My_Sub, My_Sub_TL, My_Sub_AD

  ! Module RCS source control identifier
  CHARACTER(*), PARAMETER :: MODULE_RCS_ID = &
  '$Id:$'

  ! Literal constants. Note the use of the _rk
  ! suffix to ensure the constants have the
  ! correct precision.
  REAL(rk), PARAMETER :: ZERO  =  0.0_rk
  REAL(rk), PARAMETER :: THREE =  3.0_rk
  REAL(rk), PARAMETER :: FIVE  =  5.0_rk
  REAL(rk), PARAMETER :: NINE  =  9.0_rk
  REAL(rk), PARAMETER :: TEN   = 10.0_rk

CONTAINS

  ! Forward model to compute
  !   z = 5x^2 + 3y^3
  !
  ! For more complicated models a little bit
  ! more information, such as calls made or
  ! dummy argument side effects could be
  ! listed here. Basically anything that would
  ! make a reader of the code able to more
  ! quickly understand what this routine does.
  !
  ! Note that the comment block is contiguous

  SUBROUTINE My_Sub(x, y, z, RCS_Id)
    REAL(rk),               INTENT(IN)  :: x, y
    REAL(rk),               INTENT(OUT) :: z
    CHARACTER(*), OPTIONAL, INTENT(OUT) :: RCS_Id

    ! Process optional arguments
    IF ( PRESENT(RCS_Id) ) RCS_Id = MODULE_RCS_ID

    ! Forward model computation
    z = (FIVE*(x**2)) + (THREE*(y**3))
  END SUBROUTINE My_Sub


  ! Tangent-linear model of
  !   z = 5x^2 + 3y^3
  !
  ! Again, any further information that would
  ! faciliate another user's understanding of
  ! code and its side-effects should be listed
  ! here.
  !
  ! Note that the comment block is contiguous
  SUBROUTINE My_Sub_TL(x, y, x_TL, y_TL, z_TL, RCS_Id)
    REAL(rk),               INTENT(IN)  :: x, y
    REAL(rk),               INTENT(IN)  :: x_TL, y_TL
    REAL(rk),               INTENT(OUT) :: z_TL
    CHARACTER(*), OPTIONAL, INTENT(OUT) :: RCS_Id

    ! Process optional arguments
    IF ( PRESENT(RCS_Id) ) RCS_Id = MODULE_RCS_ID

    ! Tangent-linear model computation
    z_TL = (TEN*x*x_TL) + (NINE*(y**2)*y_TL)
  END SUBROUTINE My_Sub_TL


  ! Adjoint model of
  !   z = 5x^2 + 3y^3
  !
  ! For adjoint code, there are usually side
  ! effects -- note that the z_AD input argument
  ! has intent IN OUT. Similarly for the x_AD and
  ! y_AD output arguments.
  !
  ! Note that the comment block is contiguous

  SUBROUTINE My_Sub_AD(x, y, z_AD, x_AD, y_AD, RCS_Id)
    REAL(rk),               INTENT(IN)     :: x, y        ! FWD input
    REAL(rk),               INTENT(IN OUT) :: z_AD        ! Input
    REAL(rk),               INTENT(IN OUT) :: x_AD, y_AD  ! Output
    CHARACTER(*), OPTIONAL, INTENT(OUT)    :: RCS_Id

    ! Process optional arguments
    IF ( PRESENT(RCS_Id) ) RCS_Id = MODULE_RCS_ID

    ! Adjoint model computation
    y_AD = y_AD + (NINE*(y**2)*z_AD)
    x_AD = x_AD + (TEN*x*z_AD)
    z_AD = ZERO
  END SUBROUTINE My_Sub_AD

END MODULE My_Module
```

B Example Documentation Header

A template for documentation headers is shown below. Plans do exist to construct "scrapers" to search through source code for the documentation headers and replicate them in HTML form to automatically produce webpage documentation. As such, using a common documentation template will make that process simpler.

```
!--------------------------------------------------------------------------------
!
! NAME:
!
! PURPOSE:
!
! CALLING SEQUENCE:
!
! INPUT ARGUMENTS:
!
! OPTIONAL INPUT ARGUMENTS:
!
! OUTPUT ARGUMENTS:
!
! OPTIONAL OUTPUT ARGUMENTS:
!
! FUNCTION RESULT:
!
! SIDE EFFECTS:
!
! RESTRICTIONS:
!
! COMMENTS:
!
! PROCEDURE:
!
! CREATION HISTORY:
!   Written by:  Joe Bloggs, Institution, 01-Jan-2008
!                joe.bloggs@institution.org
!
!--------------------------------------------------------------------------------
```

Figure B.1: Documentation header template

An example of using the documentation header template for the `My_Sub` subroutine from appendix A is shown in figure B.2. Note that unused headings are deleted - you should just document the important elements (and potential pitfalls) that *you* would like to know about if you were receiving the code.

Also note the listing for each argument: `UNITS`, `TYPE`, `DIMENSION` and `ATTRIBUTES`. The only one that should be considered mandatory is the `UNITS` entry. All of the other entries are obtainable from inspection of the argument declaration in the procedure itself.

```
!------------------------------------------------------------------------------
!
! NAME:
!   My_Sub
!
! PURPOSE:
!   Forward model to compute  z = 5x^2 + 3y^3
!
! CALLING SEQUENCE:
!   CALL My_Sub(x, y, z, RCS_Id=RCS_Id)
!
! INPUT ARGUMENTS:
!   x:       Brief description of x argument
!            UNITS:      Units of x-argument
!            TYPE:       REAL(rk)
!            DIMENSION:  Scalar
!            ATTRIBUTES: INTENT(IN)
!
!   y:       Brief description of y argument
!            UNITS:      Units of y-argument
!            TYPE:       REAL(rk)
!            DIMENSION:  Scalar
!            ATTRIBUTES: INTENT(IN)
!
! OUTPUT ARGUMENTS:
!   z:       Brief description of z argument
!            UNITS:      Units of z-argument
!            TYPE:       REAL(rk)
!            DIMENSION:  Scalar
!            ATTRIBUTES: INTENT(IN)
!
! OPTIONAL OUTPUT ARGUMENTS:
!   RCS_Id:  Character string containing the Revision Control
!            System Id field for the module.
!            UNITS:      N/A
!            TYPE:       CHARACTER(*)
!            DIMENSION:  Scalar
!            ATTRIBUTES: INTENT(OUT), OPTIONAL
!
! CREATION HISTORY:
!   Written by:  Joe Bloggs, Institution, 01-Jan-2008
!                joe.bloggs@institution.org
!
!------------------------------------------------------------------------------
```

Figure B.2: Example documentation header for the `My_Sub` subroutine from appendix A example source code.