`ModelE` Coding Conventions

Tom Clune
NASA Goddard Space Flight Center
email: *thomas.l.clune@nasa.gov*

January 24, 2011

Contents

1 Introduction

This document establishes certain common coding conventions within `ModelE` software. With the overarching goal of improving science productivity, these coding conventions are intended to

- reduce common causes of bugs and/or inscrutible software,
- improve overall software quality,
- reduce differences in coding styles that limit legibility, and
- enable the use of automatic sofware development tools (e.g. Photran).

Documents analogous to this are an increasingly common practice among commercial software development organizations and are widely believed to improve productivity through a number of direct and indirect impacts. No doubt the balance of these drivers are somewhat different in research organizations, and the set of conventions below are intended to be a compromise among conflicting ideals of best practices, existing coding conventions, and other unique requirements of `ModelE`. Where possible, each requirement and/or recommendation is provided with rationale in the hopes of providing a compelling motivation.

The conventions in this document will be periodically reviewed, updated, and extended to ensure maximum benefit.

1.1 Mandatory vs. Voluntary

For the most part the establishment of these conventions these conventions is *not intended to be disruptive* to ongoing work, but rather to guide a gradual transformation as the community becomes more comfortable with the various elements. In that spirit, please note that *most* conventions in this document are *not* considered to be mandatory for `ModelE` developers. Developers who find themselves uncomfortable with any items should continue with their existing coding style and/or discuss their concerns with any member of the core software engineering team.

2 Naming conventions

Named software entities (variables, procedures, etc) are perhaps the most important mechanism by which one communicates and understands the intent an implementation. The choice of a good name can be challenging in many instances, but is also often a rewarding opportunity for creativity.

2.1 General Guidelines

Explicit absolute naming rules would be difficult to produce and most likely counter-productive in practice. Instead `ModelE` developers should focus on general principles for good names and use their own experience and judgment in the final selection.

The importance of name selection generally increases with the scope for an entity. Thus, names for input parameters are the most critical followed by names for public module variables, subroutines, and functions. Next lower in priority would be names of dummy arguments. And lowest in priority would be names of local variables and private module variables. Names that are used more frequently are worth greater investment than names that are used very infrequently.

2.1.1 Communicate intention

Good names should communicate the intention of a given software entity unambiguously to other developers. The name should give a useful indication of the role that the entity serves in the software using terminology that is understandable by other developers. Generally this guideline implies a preference for full English words and phrases, with the understanding that numerous caveats and exceptions exist. As an example consider the choice for naming a variable which contains the heat flux at the bottom of a grid-cell. The variable names "f" or "Q" are common in this situation, but within a large routine do not generally provide much insight to other developers. The name "flux" is better, but still lacks a certain degree of specificity. The name "heatFlux" or "heatFluxQ" or "lowerFlux" are better and depending on context might be sufficient. In a large routine with multiple types of fluxes at various boundaries, a better name would be "lowerHeatFlux" or "heatFluxAtBottom".

Perhaps the worst offense against the guideline here would be reusing a variable with a perfectly fine name for a second very different purpose. E.g. reusing "heatFlux" at a later point to represent something like the total mass.

2.1.2 Consistency, predictability, and ambiguity

Developers should not need to unnecessarily spend time determining the correct spelling of a given name. Abbreviations of long words are perfectly natural so long as they are *consistently applied and predictable*. When multiple abbreviations or alternate spellings (or even misspellings!) are in common use,

developers *must* frequently check other pieces of code to ensure they are using the correct spelling. For example the name "trop" is probably poor for indicating a tropospheric quantity if "tropo" and "troposph" are also used. At some point in the future, a table of abbreviations will be added to this document.

Encouraged
Avoid abbreviations in names unless consistent throughout `ModelE`.

Encouraged
Use correct spelling. Where English and American spellings differ, the American spelling should be preferred.

2.1.3 Generic terminology

Although very tempting, certain common words are too generic to confer any useful information to other developers and are generally poor candidates for parts of a variable name. Examples of such bland terms include:

- variable
- parameter
- buffer
- string
- array
- table

When tempted to use such terms in a name, consider other aspects of the functionality to come up with alternatives.

Encouraged
Avoid bland or generic terms in names of variables.

2.1.4 Name length

The guidelines given above generally drive selection toward longer names which convey more information. Clearly there are advantages to shorter names as well, and a good compromise is a bit of an art form. Note that concern about time spent (wasted) typing longer names is generally misplaced, as numerous studies have shown that source code is read many more times than than it is written. Further, many modern software editors provide means to "auto-complete" names, which further reduces concerns over typing long names.

Names which are *too* long can reduce clarity, especially in long expressions. When the discrepancy is severe, there are several alternatives:

1. Split the long expression into multiple statements by introducing intermediate variables for subexpressions. This often improves the clarity in a number of ways with the intermediate names providing new avenues for communication.

2. Introduce a local variable with a shorter name to be used as an alias. Because the new name has a smaller scope and is directly associated with

the original variable, a very short string is very sensible.

3. In the near future the F2003 `associate` construct will provide a formal mechanism for using a short name (alias) to represent repeated subexpressions within a longer expression.

Mandatory
In no event shall a name exceed 31 characters which is the maximum under the F2003 standard.

The F2008 standard will extend this to 63 characters, but this is motivated by the need to support automatically generated source code, and should *not* be seen as guidance for human-generated software.

2.2 Specific conventions

2.2.1 Multi-word Names

Encouraged
`ModelE` will use the common so-called mixed-case convention for concatenating multiple words in a variable name.

In this convention the beginnings of words are indicated by starting them with capital letters, e.g. "potentialTemperature" and "numTracers". Capitalization of the first word is context dependent and discussed in more detail below. Although this convention is somewhat arbitrary, many groups have adjusted to this convention and grow to prefer it. It is important that a single convention be established as it eliminates time spent determining whether a given variable uses some other mechanism to append words. Also, although Fortran is case-insensitive, consistent capitalization aids in reading code and finding other instances of the same variable. (Not to mention simply eliminating debate about which capitalization to use in the first place.)

2.2.2 File names

As with variable names, file names should communicate their intent which should be their contents. In this sense, files should ideally contain only one entity which will either be a program, a subroutine, a function or a module. The current implementation of `ModelE` is far from this ideal, and adoption is expected to be very gradual.

Encouraged
Choose file names to coincide with its contents.

The suffix of a file name is to be used to indicate whether the overall format is *fixed* or *free*.

Mandatory
Fixed format files *must* end with the `.f` or `.F` suffix, while free format files *must* end with `.F90`.

For example, given a software entity named `foo`, the corresponding free-format file name should be `foo.F90`.

2.2.3 Derived type names

Derived type names should end with the `_type` suffix to indicate their role. This convention might change once F2003 becomes more widespread and other object-orient conventions will be more appropriate. Fortran 95 did not permit module procedures to have the same name as derived types which is a natural situation for constructor methods. F2003 relaxes this restriction.

Encouraged
Use the `_type` suffix for names of Fortran derived types.

In analogy with object-oriented languages where developers typically capitalize class names, derived type names should be capitalized. The issue is less important in Fortran since the `type` keyword is always present for derived types.

2.2.4 Module names

Modules are sufficiently fundamental that reserving a special suffix to indicate their names is a sensible and common convention. Most communities have opted to use `Mod` suffix for this purpose. This is also the recommendation for `ModelE`, but with special exemptions related to existing conventions for physical components within the model. Files containing a module should also follow the convention of dropping the `Mod` suffix in the file name. In that context the suffix is somewhat redundant, and dropping the suffix is more consistent with the style of other community software. As with derived type names, it is generally appropriate to capitalize module names.

Encouraged
Most module names should use the `Mod` suffix.

Encouraged
The `Mod` suffix should be omitted from the name of a file containing a module.

Encouraged
Capitalize module names.

Subsystem global entities module _COM A consistent existing convention within `ModelE` is for modules which provide the various global variables associated with a given physical component. The modules are currently named with the `_COM` suffix, and warrant an exception from the usual naming convention for modules. In most instances this convention is already consistent with the corresponding file name, but will eventually require a fix for th exceptions.

Subsystem driver module _DRV In `ModelE` a consistent existing convention for most physical components is to have a top level file containing the

suffix _DRV. This convention is also to be continued, but the corresponding procedure names are generally quite inconsistent with this convention. E.g. the file RAD_DRV.f contains the top-level procedure RADIA()

Both of the preceeding two exceptions are likely to be revisited if and when these physical components are re-implemented as ESMF components.

2.2.5 Procedure names

Subroutines and functions perform actions and are generally best expressed with names corresponding to English verbs. E.g. print() or accumulate(). Many routines are intended to put or retrieve information from some sort of data structure, possibly indirectly. The words "put" and "get" are useful modifiers in such instances. E.g. putLatitude or getSurfaceAlbedo(). Although these conventions are fairly natural, actual awareness of them of can be beneficial when creating names.

2.2.6 Variable names

Variable names represent objects and as such are generally best represented with names corresponding to English nouns. A good rule-of-thumb is to use singular nouns for scalars and plurals for lists/arrays. Note, however, that this rule-of-thumb has a very important exception for arrays which represent spatially distributed quantities such as temperature(i,j,k) which are referred to in the singular by common convention.

3 Fortran language constructs

3.1 Which Fortran version

In an ideal world, ModelE would to be implemented in strict compliance with the Fortran standard. However, allowance *must* be given to the evolution of the Fortran standard itself as well as to a very small number of nonstandard, yet highly portable extension to the Fortran language. At the time of this writing (January 2010), the current standard is Fortran 2003 (F2003) and the Fortran 2008 (F2008) standard is expected to be fully ratified later this year. In reality, few Fortran compilers have implemented the full F2003 standard and the interests of ModelE portability require that source code be restricted to a more portable subset of F2003 defined as that which is supported by current version of both GFortran *and* Intel Fortran compilers. ModelE execution under GFortran guarantees a strong degree of portability, while Intel guarantees continuity and high performance for GISS's primary computing environments. Note that some other compilers most likely also support this subset of F2003 (and beyond), so this constraint is not as severe as it might first appear.

Mandatory
ModelE is implemented in the subset of Fortran 2003 that is robustly implemented by both current Intel and GFortran compilers.

3.1.1 Non standard extensions in ModelE

CPP The build process of ModelE relies upon the C preprocessor (CPP), which is technically not part of the Fortran standard. This capability is essential for enabling multiple configurations of the model.

real*8 Although the Fortran 90 standard introduced portable syntax for controlling the precision of floating point quantities, the widespread extension (`real*8, real*4`) is portable on virtually all Fortran compilers and deeply embedded in ModelE. The Fortran `KIND=` mechanism is of course permitted and encouraged in software sections where support of multiple precisions is required.

3.2 Obsolete and discouraged features

Due to the desire to support legacy software, the Fortran standard rarely actually removes language features even when superior mechanisms have been introduced. ModelE developers are strongly encouraged to avoid the following language features:

`entry` statement At best this mechanism has always been confusing, and far better mechanisms now exist to share functionality across multiple interfaces. This feature is strictly forbidden from being added to ModelE, and all existing uses will soon be eliminated. This change is further motivated by some software tools which do not support this language "feature".

> **Mandatory**
> The `entry` statement should not be used in ModelE.

arithmetic `if` Although compact, this construct generally obfuscates code.

> **Mandatory**
> The arithmetic `if` construct should not be used in ModelE.

computed `goto` This feature is generally inferior to the newer `select case` construct which shows the conditions for execution at the top of each case.

> **Mandatory**
> The computed `goto` construct should not be used in ModelE.

`goto` statement Although there are still certain situations where the use of `goto` is the clearest expression of an algorithm, such situations are vanishingly rare in practice. The `cycle` and `exit` statements generally communicate intent in a superior manner within loops, and `select case` and plain old `if` statements cover most other cases.

> **Encouraged**
> Alternatives to the `goto` statement should be be used.

`continue` statement `END DO` is generally the preferred mechanism to close loops. For longer loops where the loss of a statement label might complicate finding the corresponding beginning of a loop, developers should use the F90 mechanism for labeling blocks. E.g.

```
outerLoop: do i = 1, 10
...
end do outerLoop
```

> **Encouraged**
> Avoid the use of the `continue` statement.

statement labels Although these are still necessary for `goto` statements which cannot yet be removed, other uses should rely on the F90 mechanism for labeling blocks.

> **Encouraged**
> Use F90 statement labels for long nested loops that extend more than one screen.

3.3 Required and encouraged features

Accidental misspelling of variables was once a common source of errors in Fortran programs. The introduction of `implicit none` has alleviated many such errors and fortunately has become widely used.

> **Mandatory**
> The `implicit none` statement *must* be used in all modules and all non-module subroutines and functions.

By default all Fortran module entities are "public" which can lead to problems with multiple paths by which those entities are accessed by higher level program units. The cascade of possible host association can lead to long and/or aborted compilation. Aside from these technical issues, one of the intents of the Fortran module construct is to encapsulate (i.e. hide) details of implementation from external program units. Fortunately, Fortran has the `private` statement which toggles this default.

> **Encouraged**
> Modules should use the `private` statement. Entities which should be accessible by other program units should be declared with the `public` attribute.

Even more than Fortran modules, derived types should hide the details of their internal implementation. Unfortunately, as with modules, the default public access leads to over-reliance on access to internal details. With F95 such structures must be entirely public or entirely private, but F2003 introduces finer control.

> **Encouraged**
> Fortran derived types should use the `private` statement where possible.

4 Formatting conventions

Formatting issues are far less substantive than the software elements that are discussed earlier in this document. However, a consistent "look-and-feel" can be a powerful aid to the readability of `ModelE`as well as preventing needless thrashing in CVS as one developer after another imposes their personal preference. Nonetheless, this section is intentionally minimalist and as much as possible reflects existing style within `ModelE`.

4.1 Free format

Although `ModelE`is at the time of this writing almost exclusively implemented in the older fixed-format Fortran convention, the new default format is exclusively free-format. Further, the existing code base will soon be thoroughly converted to free-format. While there are several minor advantages to free-format, the rationale for the wholesale conversion is to leverage a new generation of powerful software tools that do not support the older format.

Although, free-format permits source code to extend up to column 132, practical readability requires that source code be limited to column 80. Exceptional cases where the code marginally exceeds this threshold may be acceptable if additional line-splits have comparable consequences on appearance.

4.2 Indentation

The interior of each of the following categories of Fortran code blocks shall be indented in a consistent manner:

module <indented block> end module	subroutine <indented block> end subroutine
function <indented block> end function	program <indented block> end program
type <indented block> end type	interface <indented block> end interface

if (...) then
  <indented block>
else
  <indented block>
endif
	select case
	case (...)
	  <indented block>
	case (...)
	  <indented block>
	end select

do <indented block> end do	

At this time precisely 2 spaces shall be used for each level of indentation. Although a larger indentation is generally preferable for readability, existing reliance on very deep nesting is a dominant concern. If at some later time, deep nests have been eliminated from `ModelE`, the level of indentation will be raised.

Indentation should always be implemented with spaces, as the `<TAB>` character is not legal in Fortran source code. Unfortunately, some common editors will permit the insertion of `<TAB>` characters, so some caution is appropriate. Note to Emacs users: Although the `<TAB>` key is used to auto-indent lines of source code in Fortran mode, the editor actually only inserts (or removes) spaces to achieve indentation.

4.2.1 Indentation of documentation

Documentation in the header of procedures and modules should not be indented, while documentation lines in executable sections should be indented at the same level as the surrounding code. End-of-line not extend beyond column 80.

4.3 Spacing

4.3.1 Two word Fortran keywords

Although spaces are generally significant under the free-format convention, for most (possibly all?) compound keywords (e.g. `end do` and `go to`) the interven-

ing space is optional. For `ModelE` the convention is to require the intervening space for all such constructs except for `goto`:

- `goto`
- `end do`
- `end if`
- `end select`
- `end subroutine`
- `end function`
- `end subroutine`

- ~~`go to`~~
- ~~`enddo`~~
- ~~`endif`~~
- ~~`endselect`~~
- ~~`endsubroutine`~~
- ~~`endfunction`~~
- ~~`endsubroutine`~~

Mandatory
Use a space between compound keywords except for the `goto` statement.

4.3.2 Operators

Encouraged
To improve legibility, expressions should attempt to use the space character in a judicious manner.

The rules here are not absolute, but guidelines that should be followed unless other legibility issues are more important. In order of decreasing priority one should:

- Use at least one space should be left on each side of the assignment ("`=`") operator.
- Use at least one space on each side of "+" and "-" operators to both emphasize grouping as well as order of precedence among operators.
- *Not* use space around "*" and "**" operators.
- Use one space after "," in arguments to procedures and functions.
- *Not* use space between array indices.

4.4 Capitalization

Although Fortran is case insensitive, capitalization can be useful to convey additional information to readers. Because modern editors can generally highlight language keywords, capitalization is generally only to be applied to user-defined entities. As mentioned above, capitalization should be used to separate words within multi-word names, as well as for derived type and module names.

Encouraged
Use lower case for Fortran keywords.

Encouraged
Use mixed case for multiword names.

Encouraged
Start names with lower case except for derived types and modules.

5 Documentation

`ModelE` uses scripts to dynamically assemble certain documentation from source code in an automated manner based upon special identification tags.

5.1 Documentation of Fortran modules

Each module *must* have a top-level summary indicated with the comment tag: "`!@sum`". This summary should explain the nature of the modules contents and the role of the module within the context of the overall model.

All global (i.e. `public` module entities *must* be documented with the comment tag: "`!@var`". This documentation should emphasize the purpose of the entity, and for physical quantities the documentation should specify the physical units (e.g. "`m/s`").

Where appropriate each module should specify the primary author(s) or point(s)-of-contact with the comment tag: "`!@auth`". For more complex situations, the repository is a better mechanism for determining which developers are responsible for any bit of code.

5.2 Documentation of Fortran procedures

Mandatory
Each public procedure (subroutine or function) *must* have a top-level summary indicated with the comment tag: "`!@sum`".

This summary should explain the nature of the modules contents and the role of the module within the context of the overall model.

Mandatory
Each procedure dummy variable *must* be documented with the comment tag: "`!@var`".

This documentation should emphasize the purpose of the entity, and for physical quantities the documentation should specify the physical units (e.g. "`m/s`").

Encouraged
Important/nontrivial local variables should be also be documented with the "`!@var`" tag.

Encouraged
Where appropriate and/or different than for the surrounding module, each procedure should specify the primary author or point-of-contact with the comment tag: "`!@auth`".

For more complex situations, the repository is a better mechanism for determining which developers are responsible for any bit of code.

5.3 Documentation of rundeck parameters

Rundeck parameters are among the most important quantities from the point-of-view of other users of the software, and strong documentation for those parameters is a very high priority.

Mandatory
All rundeck parameters *must* be documented using the comment tag "`!@dbparam`".

6 Miscellaneous

6.1 Free format templates

Some users may find it convenient to begin new modules and/or procedures with a skeleton implementation that indicates such things as proper indentation and other conventions. Figure 1 provides a template for Fortran modules that conforms to the conventions established in this document. Figure 2 provides an analogous template for Fortran subroutines, and figure 3 provides a template for Fortran functions.

```
module <module-name>Mod
!@sum <summary>
!@auth <principle author>
  use <use-module>, only: <item>
  ...
  implicit none
  private

  ! list public entities
  public :: <item>
  ...

  ! declare any public derived types
  type <name>_type
    private
    <declare components of derived type>
  end type <name>_type
  ...

  ! declare public variables
  !@var <var1> <description> <units>
  real*8, allocatable :: <var1>(:,:,:)
  ...

contains

  <procedure 1>

  <procedure 2>

end module <module-name>Mod
```

Figure 1: Template for Fortran module in free-format.

```
subroutine <routine-name>(<arg1>[, <arg2>, ...])
!@sum <summary>
!@auth <principle author>
  use <use-module>, only: <item>
  ...
  implicit none ! not required for module subroutine

  ! declare dummy arguments
  !@var <arg1> <description> <units>
  real*8, allocatable, intent(...) :: <arg1>(:,:)

  ! declare local variables
  real*8, allocatable :: <var1>(:,:)


  <executable statement>
  ...

end subroutine <routine-name>
```

Figure 2: Template for Fortran subroutine in free-format.

```
function <routine-name>(<arg1>[, <arg2>, ...])
!@sum <summary>
!@auth <principle author>
  use <use-module>, only: <item>
  ...
  implicit none ! not required for module subroutine

  ! declare dummy arguments
  !@var <arg1> <description> <units>
  real*8, allocatable, intent(...) :: <arg1>(:,:)

  ! declare return type
  real*8 :: <routine-name>

  ! declare local variables
  real*8, allocatable :: <var1>(:,:)


  <executable statement>
  ...

end function <routine-name>
```

Figure 3: Template for Fortran function in free-format.

6.2 Emacs settings

The Emacs editor has a number of useful features for editing free-format Fortran files. However, the default settings (e.g. indentation) do not correspond to the conventions established in this document. The elisp code in figure 4, when inserted into a users `.emacs` file, will cause Emacs to automatically recognize files ending in "`.F90`" or "`.f90`" as free-format and set the default indentation to be 2 characters.

```
; Ensure that F90 is the default mode for F90 files
(setq auto-mode-alist (append auto-mode-alist
                        (list '("\\.f90$" . f90-mode)
                              '("\\.F90$" . f90-mode))))
; ModelE F90 indentation rules
(setq  f90-directive-comment-re "!@")
(setq  f90-do-indent 2)
(setq  f90-if-indent 2)
(setq  f90-program-indent 2)
(setq  f90-type-indent 2))
(setq  fortran-do-indent 2)
(setq  fortran-if-indent 2)
(setq  fortran-structure-indent 2)
```

Figure 4: Elisp code to customize Emacs environment for `ModelE` conventions.