European Standards for Writing and Documenting Exchangeable Fortran 90 Code

Version 1.1

Phillip Andrews (UKMO), Gerard Cats (KNMI/HIRLAM), David Dent (ECMWF), Michael Gertz (DWD), Jean Louis Ricard (Meteo France)

Contents:

Introduction

The aim of this document is to provide a framework for the use of Fortran 90 in European meteorological organizations and thereby to facilitate the exchange of code between these . In order to achieve this goal we set standards for the documentation of code, both internal and external to the software itself, as well as setting standards for writing the code. The latter standards are designed to improve code readability and maintainability as well as to ensure, as far as possible, its portability and the efficient use of computer resources.

Documentation

Documentation may be split into two categories: external documentation, outside the code; and internal documentation, inside the code. These are described in sections 2.1 and 2.2 respectively. In order for the documentation to be useful it needs to be both up to date and readable at centres other than that at which the code was originally produced. Since these other centres may wish or need to modify the imported code we specify that all documentation, both internal and external, must be available in English.

- ### External Documentation

  In most cases this will be provided at the package level, rather than for each individual routine. It must include the following:

  1. Top Level Scientific documentation: this sets out the problem being solved by the package and the scientific rationale for the solution method adopted. This documentation should be independent of (i.e. not refer to) the code itself.

  2. Implementation documentation: this documents a particular implementation of the solution method described in the scientific documentation. All routines (subroutines, functions, modules etc...) in the package should be listed by name together with a brief description of what they do. A calling tree for routines within the package must be included.

  3. A User Guide: this describes in detail all inputs into the package. This includes both subroutine arguments to the package and any switches or 'tuneable' variables within the package. Where appropriate default values; sensible value ranges; etc should be given. Any files or namelists read should be described in detail.

- ### Internal Documentation

  This is to be applied at the individual routine level. There are four types of internal documentation, all of which must be present.

  1. Procedure headers: every subroutine, function, module etc must have a header. The purpose of the header is to describe the function of the routine, probably by referring to external documentation, and to document the variables used within the routine. All variables used within a routine must be declared in the header and commented as to their purpose. It is a requirement of this standard that the headers listed in Appendix A be used and completed fully. Centres are allowed to add extra sections to these headers if they so wish.

  2. Section comments: these divide the code into numbered logical sections and may refer to the external documentation. These comments must be placed on their own lines at the

start of the section they are defining. The recommended format for section comments is:

```
!-------------------------------------------------------------------------------------------

! <Section number> <Section title>

!-------------------------------------------------------------------------------------------
```

where the text in <> is to be replaced appropriately.

3. Other comments: these are aimed at a programmer reading the code and are intended to simplify the task of understanding what is going on. These comments must be placed either immediately before or on the same line as the code they are commenting. The recommended format for these comments is:

   ```
   ! <Comment>
   ```

   where the text in <> is to be replaced appropriately.

4. Meaningful names: code is much more readable if meaningful words are used to construct variable & subprogram names.

   It is recommended that all internal documentation be written in English. However, it is recognized that this may not always be possible, so alternative rules for native language comments with duplicate English comments are provided.

   i) Meaningful names may be written in native language.

   ii) Section comments (**see above**) written in native language must be duplicated in English.

   iii) Other comments (**see above**) written in native language should preferably be duplicated in English.

   iv) Description and method sections of the header written in native language must be duplicated in English.

   v) Comments describing the declared variables in the header section may be written in native language. A duplicate declaration with an English language comment should be placed in a separate file (containing all such duplicate declarations for the package). Since repeated use of the same variable name for different purposes with a given package is forbidden, a simple tool can be provided to replace the native language declarations with English language declarations in the source files of ported code.

Coding Rules For Packages

These rules are loosely based on the "plug compatibility" rules of Kalnay et al. (1989). Rules which appear elsewhere in this document have not been duplicated in this section.

- A package shall refer only to its own modules and subprograms and to those intrinsic routines included in the Fortran 90 standard.

  There may well be a need to extend this to include named (in this document) libraries such as the NAG library. We could even have a joint Met. utilities library which could contain things such as multigrid p.d.e. solvers or routines to calculate saturated vapour pressure...

- A package shall provide separate set up and running procedures, each with a single entry point. All initialization of static data must be done in the set up procedure and this data must not be altered by the running procedure.

- External communication with the package must be via:

    1. The argument lists of the packages entry and set up routines.

    2. A NAMELIST file read.

    3. Environment variables read by a set of standard routines. The environment variables must be documented in and set by a script appropriate to the operating system in use (i.e. a posix script for the unix operating system).

       We could also allow the use of standard modules, to be defined in this document, containing information on the grid type and resolution etc; error handling...

    4. Internally the package may use Modules - for example the set up and running procedures may communicate in this way.

    5. Interface blocks must be provided for the set up and running procedures (possibly via module(s)). This allows the use of assumed shape arrays, optional arguments, etc as well as allowing the compiler to check that the actual and dummy arguments types match. If variables in these external argument lists are of derived type, a module must be supplied which contains the type definition statements required to make these derived types available to the routine calling the package.

    6. The package shall not terminate program execution. If any error occurs within the package it should gracefully exit, externally reporting the error via an integer variable in its argument list (+ve = fatal error). Packages should also write a diagnostic message describing the problem, using fortran I/O, to an 'error stream' unit number selectable via the package's set up routine.

       Note that if the package starts at the unix script, rather than Fortran, level making a graceful exit includes returning control to the package's script level by using a STOP statement in the Fortran part of the package.

    7. The package should be written so that it is as resolution independent as possible. The resolution must be adjustable via the set up routine for the package.

    8. Precompilers: these are used, for example, to provide a means of selecting (or deselecting) parts of the code for compilation. Clearly to simplify portability of code we need to all use the same precompiler, and this needs to be available to every centre. The C precompiler is probably the best option since it will be found on all machines using the unix operating system.

       Adopting this as the standard precompiler will have some problems as different centres are currently committed to different precompilers. However, it may not be too big a task for each centre to convert from their current precompiler to the C precompiler.

    9. All unix scripts must be written using the posix shell. This is a standardized shell, available on all POSIX compliant unix systems, with many useful features.

    10. Each program unit should be stored in a separate file.

Guidance For The Use Of Dynamic Memory

The use of dynamic memory is highly desirable as, in principle, it allows one set of compiled code to work for any specified resolution (or at least up to hardware memory limits); and allows the efficient reuse of work space memory. Care must be taken, however, as there is potential for inefficient memory usage, particularly in parallelized code. For example heap fragmentation can occur if space is allocated by a lower level routine and then not freed before control is passed back up the calling tree. There are three ways of obtaining dynamic memory in Fortran 90:

1. Automatic arrays: These are arrays initially declared within a subprogram whose extents depend upon variables known at runtime e.g. variables passed into the subprogram via its argument list.
2. Pointer arrays: Array variables declared with the POINTER attribute may be allocated space at run time by using the ALLOCATE command.
3. Allocatable arrays: Array variables declared with the ALLOCATABLE attribute may be allocated space at run time by using the ALLOCATE command. However, unlike pointers, allocatables are not allowed inside derived data types.

- Use automatic arrays in preference to the other forms of dynamic memory allocation.
- Space allocated using b) and c) above must be explicitly freed using the DEALLOCATE statement.
- In a given program unit do not repeatedly ALLOCATE space, DEALLOCATE it and then ALLOCATE a larger block of space. This will almost certainly generate large amounts of unusable memory.
- Always test the success of a dynamic memory allocation and deallocation. The ALLOCATE and DEALLOCATE statements have an optional argument to let you do this.

Coding Rules For Routines

By routines we mean any fortran program unit such as a subroutine, function, module or program. These rules are designed to encourage good structured programming practice, to simplify maintenance tasks, and to ease readability of exchanged code by establishing some basic common style rules.

Banned Fortran Features

Some of the following sections detail features deprecated in or made redundant by Fortran 90. Others ban features whose use is deemed to be bad programming practice as they can degrade the maintainability of code.

1. COMMON blocks - use Modules instead.
2. EQUIVALENCE - use POINTERS or derived data types instead.
3. Assigned and computed GO TOs - use the CASE construct instead.
4. Arithmetic IF statements - use the block IF construct instead.
5. Labels (only one allowed use).
    - Labelled DO constructs - use End DO instead.

- I/O routine's End = and ERR = use IOSTAT instead.
- FORMAT statements: use Character parameters or explicit format specifiers inside the Read or Write statement instead.
- GO TO

  The only recognized use of GO TO, indeed of labels, is to jump to the error handling section at the end of a routine on detection of an error. The jump must be to a CONTINUE statement and the label used must be 9999. Evens so, it is recommended that this practice be avoided.

  Any other use of GO TO can probably be avoided by making use of IF, CASE, DO WHILE, EXIT or CYCLE statements. If a GO TO really has to be used, then clearly comment it to explain what is going on and terminate the jump on a similarly commented CONTINUE statement.

6. PAUSE
7. ENTRY statements: - a subprogram may only have one entry point.
8. Functions with side effects i.e. functions that alter variables in their argument list or in modules used by the function; or one that performs I/O operations. *This is very common in C programming, but can be confusing. Also, efficiencies can be made if the compiler knows that functions have no side effects. High Performance Fortran, a variant of Fortran 90 designed for massively parallel computers, will allow such instructions.*
9. 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: it is therefore unlikely to work on a massively parallel computer. Hence the practice is banned.

Style Rules

The general ethos is to write portable code that is easily readable and maintainable. Code should be written in as general a way as possible to allow for unforseen modifications. In practice this means that coding will take a little longer. This extra effort is well spent, however, as maintenance costs will be reduced over the lifetime of the software.

1. Use free format syntax.
2. The maximum line length permitted is 80 characters. Fortran 90 allows a line length of up to 132 characters, however this could cause problems when viewed on older terminals, or if print outs have to be obtained on A4 paper.
3. Implicit none must be used in all program units. This ensures that all variables must be explicitly declared, and hence documented. It also allows the compiler to detect typographical errors in variable names.
4. Use meaningful variable names, preferably in English. Recognized abbreviations are acceptable as a means of preventing variable names getting too long.

5. Fortran statements must be written in upper case only or with initial letter capitalization and the rest in lower case. Names, of variables, parameters, subroutines etc may be written in mixed, mostly lower, case.

6. To improve readability indent code within DO; DO WHILE; block IF; CASE; Interface; etc constructs by 2 characters.

7. Indent continuation lines to ensure that e.g. parts of a multi line equation line up in a readable manner.

8. Where they occur on separate lines indent type c) internal comments to reflect the structure of the code. If this is done by one character less than the code indentation comments are clearly separated from the code yet do not break up its structure.

9. Use blank space, in the horizontal and vertical, to improve readability. In particular leave blank space between variables and operators, and try to line up related code into columns. For example,

   instead of:

   ```
    ! Initialize Variables
   x=1

   MEANINGFUL_NAME=3.0

   SILLY_NAME=2.0
   ```

   write:

   ```
    ! Initialize variables
   x = 1

   MeaningfulName = 3.0

   SillyName = 2.0
   ```

   Similarly, try to make equations recognizable and readable as equations. Readability is greatly enhanced by starting a continuation line with an operator placed in an appropriate column rather than ending the continued line with an operator.

10. Do not use tab characters in your code: this will ensure that the code looks as intended when ported.

11. Separate the information to be output from the formatting information on how to output it on I/O statements. That is don't put text inside the brackets of the I/O statement.

12. Delete unused header components.

13. There was a strong desire by many of the authors of this document to include a recommended naming convention for variables. It was decided however that Fortran 77 conventions were unsuitable, and that more experience of Fortran 90 was required before an appropriate convention could be specified. It is intended that such a convention be included in a revised version of this document.

Use Of New Fortran Features

It is inevitable that there will be 'silly' ways to use the new features introduced into Fortran 90. Clearly we may want to ban such uses and to recommend certain practices over others. However, there will have to be a certain amount of experimentation with the new language until we gain enough experience to make a complete list of such recommendations. The following rules will have to be amended and expanded in the light of such experience.

1. We recommend the use of Use, ONLY to specify which of the variables, type definitions etc defined in a module are to be made available to the Useing routine.

2. Discussion of the use of Interface Blocks:

   Introduction

   Explicit interface blocks are required between f90 routines if optional or keyword arguments are to be used. They also allow the compiler to check that the type, shape and number of arguments specified in the CALL are the same as those specified in the subprogram itself. In addition some compilers (e.g. the Cray f90 compiler) use the presence of an interface block in order to determine if the subprogram being called is written in f90 (this alters how array information is passed to the subroutine). Thus, in general it is desirable to provide explicit interface blocks between f90 routines.There are several ways to do this, each of which has implications for program design; code management; and even configuration control. The three main options are discussed in the following sections:

   Option I: Explicitly Coded Interface Blocks

   Interface blocks may be explicitly written into the calling routine, essentially by copying the argument list declaration section from the called routine. This direct approach has, however, some disadvantages. Firstly, it creates an undesirable increase in the work required to maintain the calling routine, as if the argument list of the called routine changes the Interface block must be updated as well as the CALL. Further, there is no guarantee that the Interface block in the calling routine is actually up to date and the same as the actual interface to the called routine.

   Option II: Explicitly Coded Interface Blocks in a Module

   Interface blocks for all routines in a package may be explicitly written into a module, and this module used by all routines in the package. This has the advantage of having one routine to examine to find the interface specification for all routines - which may be easier than individually examining the source code for all called routines. However, an Interface block must still be maintained in addition to the routine itself and CALLs to it, though a program or e.g. a unix script could be written to automatically generate the module containing the interface blocks.

   Option III: Automatic Interface Blocks

   Fortran 90 compilers can automatically provide explicit interface blocks between routines following a Contains statement. The interface blocks are also supplied to any routine Useing the module. Thus, it is possible to design a system where no Interface blocks are actually coded and yet explicit interface blocks are provided between all routines by the compiler. One way to do this would be to 'modularise' the code at the f90 module level, i.e. to place related code together in one module after the Contains statement. Routine a, in module A calling routine b in module B would then only have to Use module B to be automatically provided with an explicit interface to routine b. Obviously if routine b was in module a instead then no Use would be required. One consequence of this approach is that a module and all the routines contained within it make up a single compilation unit. This may be a disadvantage if modules are large or if each module in a package contains routines which Use many other modules within the package (in which case changing one routine in one module would necessitate the recompilation of virtually the entire package). On the other hand the number of compilation units is greatly reduced, simplifying the compilation and configuration control systems.

Conclusion

Options II) and III) both provide workable solutions to the problem of explicit interface blocks. Option III is probably preferable as the compiler does the work of providing the interface blocks, reducing programming overheads, and at the same time guaranteeing that the interface blocks used are correct. Which ever option is chosen will have significant impact on code management and configuration control as well as program design.

3. Array notation should be used whenever possible. This should help optimization and will reduce the number of lines of code required. To improve readability show the array's shape in brackets, e.g.:

   1dArrayA(:) = 1dArrayB(:) + 1dArrayC(:)

   2dArray(:, :) = scalar * Another2dArray(:, :)

4. When accessing subsections of arrays, for example in finite difference equations, do so by using the triplet notation on the full array, e.g.:

```
2dArray(:, 2:len2) = scalar                        &
                   * ( Another2dArray(:, 1:len2 -1) &
                     - Another2dArray(:, 2:len2)    &
                     )
```

5. Always name 'program units' and always use the End program; End subroutine; End interface; End module; etc constructs, again specifying the name of the 'program unit'.

6. Use >, >=, ==, <, <=, /= instead of .gt., .ge., .eq., .lt., .le., .ne. in logical comparisons. The new syntax, being closer to standard mathematical notation, should be clearer.

7. Don't put multiple statements on one line: this will reduce code readability.

8. Variable declarations: it would improve understandability if we all adopt the same conventions for declaring variables as Fortran 90 offers many different syntaxes to achieve the same result.

   - Don't use the DIMENSION statement or attribute: declare the shape and size of arrays inside brackets after the variable name on the declaration statement.

   - Always use the :: notation, even if their are no attributes.

   - Declare the length of a character variable using the (len =) syntax.

9. We recommend against the use of recursive routines on efficiency grounds (they tend to be inefficient in their use of cpu and memory).

10. It is recommended that new operators be defined rather than overload existing ones. This will more clearly document what is going on and should avoid degrading code readability or maintainability.

11. To improve portability between 32 and 64 bit platforms, it is extremely useful to make use of kinds to obtain the required numerical precision and range. A module should be written to define parameters corresponding to each required kind, for example:

```
 Integer, parameter       :: single            & ! single precision kind.
                           = selected_real_kind(6,50)
 Integer, parameter       :: double            & ! double precision kind.
                           = selected_real_kind(12,150)
```

This module can then be Used in every routine allowing all variables declared with an appropriate kind e.g.

```
  Real(single), pointer   :: geopotential(:,:,:) ! Geopotential height
fields
```

Enforcing These Standards

It is obviously important to ensure that these standards are adhered to - particularly that the documentation is kept up to date with the software; and that the software is written in as portable a manner as possible. If these standards are not adhered to exchangeability of the code will suffer. It may be that software tools, such as QA fortran, could be tailored to test for compliancy to these standards. This needs investigation.

One option would be to set up a central database for exchangeable code and its external documentation. The acceptance criteria would be that the standards set out in this document are met. This would of course require funding to provide hardware, software and personnel to maintain the data base...

At the other extreme each centre could adopt it's own enforcement strategy, and distribute lists of it's own exchangeable software to the other centres...

The best solution may well be to design a distributed system making use of Mosaic and internet.

References

Kalnay et al. (1989) "Rules for Interchange of Physical Parametrizations" Bull. A.M.S., 70 No. 6, p 620.

Appendix A: Modification History:

23/3/94: Draft Version 0.1 Phillip Andrews (UKMO)
22/4/94: Draft Version 0.2 Phillip Andrews (UKMO)
7/6/94: Draft Version 0.3 Phillip Andrews (UKMO)
31/7/94: Draft Version 0.4 Phillip Andrews (UKMO)
29/9/94: Draft Version 0.5 Phillip Andrews (UKMO)
14/10/94: Draft Version 0.6 Phillip Andrews (UKMO)

Renumbered Version 1.0

20/6/95: Version 1.1 Phillip Andrews (UKMO)

Appendix B: Standard Headers

The standard headers are presented in this appendix. They are written as templates. Text inside < > brackets must be replaced with appropriate text by the user.

Program Header

```
!+ <A one line description of this program>
!
Program <NameOfProgram>

! Description:
!   <Say what this program does>
!
! Method:
!   <Say how it does it: include references to external documentation>
!   <If this routine is divided into sections, be brief here,
!        and put Method comments at the start of each section>
!
! Input files:
!   <Describe these, and say in which routine they are read>

! Output files:
!   <Describe these, and say in which routine they are written>

! Current Code Owner: <Name of person responsible for this code>

! History:
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>

! Code Description:
!   Language:           Fortran 90.
!   Software Standards: "European Standards for Writing and
!     Documenting Exchangeable Fortran 90 Code".

! Declarations:

! Modules used:

Use, Only : &
! Imported Type Definitions:

! Imported Parameters:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Routines:

! <Repeat from Use for each module...>

Implicit None

! Include statements
! Declarations must be of the form:
```

```
! <type>   <VariableName>      ! Description/ purpose of variable

! Local parameters:

! Local scalars:

! Local arrays:

!- End of header ---------------------------------------------------------------
```

Subroutine header

```
!+ <A one line description of this subroutine>
!
Subroutine <SubroutineName> &
!
  (<InputArguments, inoutArguments, OutputArguments>)

! Description:
!   <Say what this routine does>
!
! Method:
!   <Say how it does it: include references to external documentation>
!   <If this routine is divided into sections, be brief here,
!        and put Method comments at the start of each section>
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
!   Language:           Fortran 90.
!   Software Standards: "European Standards for Writing and
!     Documenting Exchangeable Fortran 90 Code".
!
! Declarations:
! Modules used:

Use, Only : &
! Imported Type Definitions:

! Imported Parameters:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Routines:

! <Repeat from Use for each module...>

Implicit None

! Include statements:
! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable
! Subroutine arguments
! Scalar arguments with intent(in):

! Array  arguments with intent(in):

! Scalar arguments with intent(inout):

! Array  arguments with intent(inout):

! Scalar arguments with intent(out):

! Array  arguments with intent(out):

! Local parameters:

! Local scalars:

! Local arrays:

!- End of header ---------------------------------------------------------------
```

Function header

```
!+ <A one line description of this function>
!
Function <FunctionName> &
  (<InputArguments>)    &
Result (<ResultName>) ! The use of result is recommended
                            ! but is not compulsory.

! Description:
!   <Say what this function does>
!
! Method:
!   <Say how it does it: include references to external documentation>
!   <If this routine is divided into sections, be brief here,
!        and put Method comments at the start of each section>
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
!   Language:           Fortran 90.
!   Software Standards: "European Standards for Writing and
!     Documenting Exchangeable Fortran 90 Code".
!
! Declarations:
! Modules used:

Use, Only : &
! Imported Type Definitions:

! Imported Parameters:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):
```

```
! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Routines:

! <Repeat from Use for each module...>

Implicit None

! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable

! Include statements:

! Function arguments
! Scalar arguments with intent(in):

! Array  arguments with intent(in):

! Local parameters:

! Local scalars:

! Local arrays:

!- End of header ------------------------------------------------------------
```

Module header

```
!+ <A one line description of this module>
!
Module <ModuleName>

!
! Description:
!   <Say what this module is for>
!
! Current Code Owner: <Name of person responsible for this code>
!
! History:
!
! Version   Date     Comment
! -------   ----     -------
! <version> <date>   Original code. <Your name>
!
! Code Description:
!   Language:           Fortran 90.
!   Software Standards: "European Standards for Writing and
!     Documenting Exchangeable Fortran 90 Code".
!
! Modules used:
!
Use, only : &
! Imported Type Definitions:

! Imported Parameters:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):
```

```
! Imported Array Variables with intent (out):

! Imported Routines:

! <Repeat from Use for each module...>

! Declarations must be of the form:
! <type>   <VariableName>      ! Description/ purpose of variable

 Imlicit none
! Global (i.e. public) Declarations:
! Global Type Definitions:

! Global Parameters:

! Global Scalars:

! Global Arrays:

! Local (i.e. private) Declarations:
! Local Type Definitions:

! Local Parameters:

! Local Scalars:

! Local Arrays:

! Operator definitions:
!   Define new operators or overload existing ones.

Contains
! Define procedures contained in this module.

End module <ModuleName>

!- End of module header
```

Appendix C: Examples

A VAR subroutine

This example subroutine has been taken from the UKMO's variational assimilation code. Some additional style rules have been applied such as placing one argument per line and commenting the argument with its intent.

```
!+ Allocates and calculates the LS field Vtheta
Subroutine Var_LSVtheta &
 ( LocalFS,             & ! in
   LS )                   ! inout

! Description:
!   Allocates and calculates the linearization state derived field Vtheta:
!   i.e. the virtual potential temperature.
!
! Method:
!   See UMDP 101 section 3.1.3.
!
! Owner: Phil Andrews
!
! History:
```

```
! Version   Date     Comment
! -------   ----     -------
! 0.1     01/09/94   Original code. Phil Andrews
! 0.2     26/10/94   Changed references from theta to thetaL and corrected
! 0.2                the calculation of Vtheta. Mike Thurlow.
! 0.3     07/02/95   Test of FieldStatus added.  Phil Andrews.
! 1.4     11/05/95   Tracing added. JB
!
! Code Description:
!   Language:          Fortran 90.
!   Software Standards: "European Standards for Writing and
!     Documenting Exchangeable Fortran 90 Code".
!
! Parent module: VarMod_LS
!
! Declarations:

! Modules used:
Use VarMod_PFInfo,       only :         &
! Imported routines:
 Var_DeallocateModel,                   &

! Imported Type Definitions:
 ModelDump_type,                        & ! Stores LS PF or Adj states.
 ModelDumpHeader_type,                  & ! Structures use within
 Header_type,                           & ! ModelDump_type
 XYgrid_type,                           & ! ditto
 Zgrid_type,                            & ! ditto

! Imported Parameters:
 FieldStatus_absent                       ! FieldStatus code for absent

Use VarMod_Constants, only :            &
! Imported Parameters:
 InvGasRatioMinusOne

Use VarMod_Trace, only :                &
! Imported routines:
 Var_TraceEntry,                        &
 Var_Trace,                             &
 Var_TraceExit,                         &

! Imported scalars:
 UseTrace,                              &
 TraceNameLen

Implicit none

!* Subroutine arguments
! Scalar arguments with intent(in):
 Integer, intent(in)  :: LocalFS          ! value to use for FieldStatus

! Scalar arguments with intent(inout):
 Type (ModelDump_type), intent(inout)   & ! linearization state
                      :: LS

!* End of Subroutine arguments

! Local parameters:
 Character (len=TraceNameLen), parameter &
                      :: RoutineName = "Var_LSVtheta"

! Local scalars:
 Integer              :: qlevels          ! Number of q levels
 Integer              :: Tlevels          ! Number of temperature levels
```

```
!- End of header --------------------------------------------------------------

 If (LS % header % Vtheta % FieldStatus == FieldStatus_absent) then


  !-----------------------------------------------------------------------------
  ![1.0] Initialize: allocate space, calculate required fields etc...
  !-----------------------------------------------------------------------------
   If (UseTrace) call Var_TraceEntry(RoutineName)

   qlevels = LS % header % qT % z % TopLev
   Tlevels = LS % header % thetaL % z % TopLev

  ! Allocate LS % Vtheta:
   Allocate (LS % Vtheta                           &
              (1:LS % header % thetaL % x % len,   &
               1:LS % header % thetaL % y % len,   &
               1:LS % header % thetaL % z % TopLev &
            ) )

  ! Set header values for LS % Vtheta:
   LS % header % Vtheta               = LS % header % thetaL
   LS % header % Vtheta % FieldStatus = LocalFS

  ! Obtain derived LS fields as necessary:
   Call Var_LSqc         &
    ( LocalFS +1,        & ! in
      LS )                 ! inout

   Call Var_LStheta      &
    ( LocalFS +1,        & ! in
      LS )                 ! inout


  !-----------------------------------------------------------------------------
  ![2.0] Calculate LS % Vtheta:
  !-----------------------------------------------------------------------------
  ! lowest to top wet level:
   LS % Vtheta(:,:,1:qlevels) = LS % theta(:,:,1:qlevels)                     &
                              * ( 1.0                                         &
                                + ( InvGasRatioMinusOne                       &
                                  * ( LS % qT(:,:,:) - LS % qc(:,:,:) )       &
                                ) )

  ! Top wet level +1 to top dry level:
   LS % Vtheta(:,:,qlevels+1:Tlevels) = LS % thetaL(:,:,qlevels +1:Tlevels)

  ! Tidy up:
   Call Var_DeallocateModel &
    ( LocalFS +1,           & ! in
      LS )                    ! inout

  ! That's it!
   If (UseTrace) call Var_TraceExit(RoutineName)

 End if
End subroutine Var_LSVtheta
```

Operators

The following program demonstrates how new operators can be defined in Fortran 90. It also shows how to extend the existing operators and the assignment operation.

```
!+ Extend assignment and + operators, & define two new ops
!
Module Operators
!
! Description:
!
! Module to extend the assignment operation so that it will
! convert characters to an integer, extend the + operator to
! add logical values, and define two new operators.
!
! Current Code Owner: A N Other
!
! History:
! Version   Date   Comment
! -------  ------  -------
!   1.0    5/5/95  Original code. (A N Other)
!
! Code Description:
! Language: Fortran 90.
!
! The procedure named must be a subroutine with one
! intent(OUT) argument and one intent(in) argument.

  Implicit none

! Override operators:
  Interface assignment(=)
    Module procedure Chars_to_Integer

  End interface

! Similarly the next interface will arrange for Add_Logicals
! to be called if two logical values are added together. The
! procedure named must be a function with two intent(in)
! arguments. The result of the addition is the result of the
! function.

  Interface operator(+)
    Module procedure Add_Logicals

  End interface

! Define new operators:

! This creates a new operator .Half. It is a unary operator,
! because Divide_By_Two only takes one argument.

  Interface operator(.Half.)
    Module procedure Divide_By_Two

  End interface



! This creates a new operator .JPlusKSq. It is a binary
! operator, because J_Plus_K_Squared takes two arguments.

  Interface operator(.JPlusKSq.)
```

```
    Module procedure J_Plus_K_Squared

  End interface

Contains

! Define procedures contained in this module

  subroutine Chars_to_Integer(int, int_as_chars)

  ! Subroutine to convert a character string containing
  ! digits to an integer.

    Character(len=5), intent(in)  :: int_as_chars
    Integer, intent(OUT)          :: int

    Read (int_as_chars, FMT = '(I5)') int

  End subroutine Chars_to_Integer

  Function Add_Logicals(a, b) result(c)

  ! Description:
  ! Function to implement addition of logical values as an
  ! OR operation.

    Logical, intent(in)  :: a
    Logical, intent(in)  :: b
    Logical              :: c

    c = a .OR. b

  End function Add_Logicals

  Function Divide_By_Two(a) result(b)
    Integer, intent(in)  :: a
    Integer              :: b

    b = a / 2

  End function Divide_By_Two

  Function J_Plus_K_Squared(j, k) result(l)
    Integer, intent(in)  :: j
    Integer, intent(in)  :: k
    Integer              :: l

    l = (j + k) * (j + k)

  End function J_Plus_K_Squared
End module Operators

!- End of module header
!+ Test program for operator module
!
Program Try_Operators

! Description:
! Program to test the operators defined in the module
! Operators
!
! Current Code Owner: A N Other
! History:
! Version   Date   Comment
! -------  ------  -------
!   1.0    5/5/95  Original code. (A N Other)
!
! Code Description:
! Language: Fortran 90.

! Modules used:

Use Operators

Implicit none

! Local scalars:
Character(len=5)  :: string5 = '-1234'   ! Char. to integer

Integer           :: i_value = 99999     ! and half operator
Integer           :: half_value          ! test variables.
Integer           :: n                   ! Test variables
Integer           :: l = 4               ! for add & square
Integer           :: m = 8               ! operation.

Logical           :: x = .false.         ! Test variables
Logical           :: y = .true.          ! for logical
Logical           :: z = .false.         ! addition ops.

!- End of header

  i_value = string5     ! Convert from a string to an integer.
  z       = x + y       ! Add together some logicals.

  Print *, string5, i_value, x, y, z

  half_value = .Half. i_value   ! Use user def unary operator
  n          = l .JPlusKSq. m   ! Use user def binary operator

  Print *, i_value, half_value, l, m, n

End program Try_Operators
```

Generic Functions

This program shows how to define a generic function (one which is defined for arguments of more than one type). It also illustrates how explicit interfaces may be required when using subroutines or functions, and how these are provided automatically if the subroutines or functions are placed in a module.

```
!+ Generic function to return a cube of a number
!
Module Generic_Cube
!
! Description:
! Module to define a generic function called Cube which
! returns the cube of a number. The function is defined for
! both integer and real arguments so, for example, Cube(2)
! and Cube(4.263) will both work.
!
! Current Code Owner: A N Other
!
! History:
! Version   Date   Comment
! -------  ------  -------
!   1.0    5/5/95  Original code. (A N Other)
!
! Code Description:
! Language: Fortran 90.

  Implicit none

! The interface says that when the generic function Cube is
! used the compiler should either call R_Cube or I_Cube
! depending on the type of the argument.

  Interface Cube
    Module procedure R_Cube, I_Cube

  End interface

Contains

  Function R_Cube(value) result(res)! Compiler knows to call
    Real  :: value                  ! this with real args
    Real  :: res                    ! because 'value' is

    a = value * value * value       ! declared as Real.

  End function R_Cube

  Function I_Cube(value) result(res)! Compiler knows to call
    Integer  :: value               ! this with integer args
    Integer  :: res                 ! because 'value' is

    a = value * value * value       ! declared as Integer.

  End function I_Cube
End module Generic_Cube

!- End of module header




!+ Print the cube roots numbers.
!
Program Cubes

! Description:
! Program to print the cubes and cube roots of some numbers.
!
! Current Code Owner: A N Other
!
! History:
! Version   Date   Comment
! -------  ------  -------
!   1.0    5/5/95  Original code. (A N Other)
!
! Code Description:
! Language: Fortran 90.


Use Generic_Cube               ! Make the generic function Cube
                               ! available in this program
Implicit none

! Local scalars:
  Integer  :: i = 8            ! Integer test value
```

```fortran
  Real     :: a = 8.01        ! Real test value

! Interface to define a generic function Cube_Root which
! returns the cube root of an integer or real number.
! Because the functions R_Cube_Root and I_Cube_Root are not
! in a module, the compiler doesn't know the types of their
! arguments when it compiles the main program. This means we
! have to give them explicitly in the interface, which is
! laborious and error prone. The moral of the tale is - use
! a module!

Interface Cube_Root
  Function R_Cube_Root(value) result(res)
    Real  :: value
    Real  :: res

  End function R_Cube_Root

  Function I_Cube_Root(value) result(res)
    Integer  :: value
    Integer  :: res

  End function I_Cube_Root
End interface

 ! Print the cubes and cube roots of some numbers using the
 ! Cube and Cube_Root generic functions.

  Write (*, "(3(A, I9.3))") "i: ", i, " Cube: ", Cube(i), &
                            " Cube root: ", Cube_Root(i)
  Write (*, "(3(A, F9.4))") "a: ", a, " Cube: ", Cube(a), &
                            " Cube root: ", Cube_Root(a)

End program Cubes


!+ Return the cube root of a real number
!
Function R_Cube_Root(value) result(res)

! Description:
! Function returning the cube root of a real number.
!
! Current Code Owner: A N Other
!
! History:
! Version   Date   Comment
! -------  ------  -------
!   1.0    5/5/95  Original code. (A N Other)
!
! Code Description:
! Language: Fortran 90.

  Implicit none

! Scalar arguments with intent(in)
  Real  :: value             ! Input value

! Local scalars
  Real  :: res               ! Result

!- End of header

  res = value ** (1.0 / 3.0)
```

```
End function R_Cube_Root


!+ Return the cube root of an integer
!
Function I_Cube_Root(value) result(res)

! Description.
! Function returning the integer part of the cube root of
! its integer argument.
!
! Current Code Owner: A N Other
!
! History:
! Version   Date   Comment
! -------  ------  -------
!   1.0    5/5/95  Original code. (A N Other)
!
! Code Description:
! Language: Fortran 90.

  Implicit none

! Scalar arguments with intent(in)
  Integer  :: value          ! Input value

! Local scalars
  Integer  :: res            ! Result

!- End of header

  res = int(Real(value) ** (1.0 / 3.0))

End function I_Cube_Root
```