ESP-r Coding Guide
Version of 9 February 2006

This coding guide has been developed to promote consistent coding style for ESP-r developments, with the objective of ensuring readable code that is accessible to the ESP-r development community. What follows is a set of general principles and prescriptive requirements that should be followed by all developers. It may be expected that this document will evolve over time in response to the evolution of FORTRAN an C coding conventions and trends within the ESP-r development community.

General Principles

- Clear and concise documentation has the same importance as bug-free code.
- The source code should be well documented through the use of appropriately chosen variable and subroutine names and a transparent code structure.
- Details about the operation of code sections should be provided within related code annotations and not in external documentation.
- Invest the necessary time to ensure that your code is understandable to your colleagues (and to yourself in a few years time!).
- Use proper English sentences to document code—this includes capitalisation and punctuation. Strive for clarity as incoherent language leads to confusion and ambiguity.
- When adding new functionality maintain consistency in style with existing source code. For example, coefficient generator subroutines for different plant component types will usually differ only in the equations used to calculate matrix equation coefficients. To install a new plant component it is usually possible to adopt the pattern of an existing component.
- Be consistent in the use of variable names throughout the code. For example, all plant component coefficient generators produce coefficients for the plant matrix solver. These are local variables to each subroutine and are passed in the calling statement. These local variables are named 'COUT' in each plant coefficient generator. Therefore, for consistency's sake it is preferred to use this same variable name in a new plant coefficient generator.

Documentation

The source code should be well commented and these comments should precede the code fragments to which they relate. Such comments should be used judiciously and grouped in a manner that illuminates rather than obscures the code. Spacing should also be used judiciously to logically group blocks of code. Comments may be indented when this clarifies the interpretation of the code. Two examples that illustrate acceptable commenting styles follow.

```
      <preceding code fragment>

C Loop through each of the selected zones and scan the Operations file
C if it exists. If not, insert default crack connection.
      do 38 izt=1,izn
        iz=ivals(izt)
        <following lines of code>

C Increment pointer to the current zone.
        nodeforcurrent=nodeforcurrent+1
```

```
        <preceeding code fragment>

C-------------------------------------------------------------------------------
C Calculate the molar flow rate of each gas constituent.
C-------------------------------------------------------------------------------
C-------N2 in air and fuel flowing into FCPM does not react.
        Ndot_FCPMexh_N2  = chi_air_N2*Ndot_FCPM_air
     &                   + chi_fuel_N2*Ndot_FCPM_fuel
C-------Ar in air flowing into FCPM does not react.
        Ndot_FCPMexh_Ar  = chi_air_Ar*Ndot_FCPM_air
C-------O2 in exhaust comes from fuel and excess air.
        Ndot_FCPMexh_O2  = chi_fuel_O2*Ndot_FCPM_fuel
     &                   + lambda_FCPM*Ndot_FCPM_O2_stoich

        <next code fragment>
```

Subroutine descriptions

Each subroutine should start with a high-level explanation of its purpose and how this is achieved. An example follows.

```
C Subroutine XYZ calculates the four heat loss factors for the foundation
C in the zone under consideration. Correlation coefficients for
C the 'corner-correction method' are used. The factors are placed into
C Common Block BSHLF for use in later heat loss calculations.
```

Citing papers and references

Citations to algorithms and data sources should be given at the beginning of a subroutine. These citations should be complete as a colleague may need to locate the paper or report in the future. Refer to proprietary reports (internal reports, drafts reports, private sources) only when the information is not available in the public domain (conference proceedings, journals, theses). An example follows.

```
C Smith A and Jones B (1999), 'Heat Transfer Coefficient Correlations for
C Building Energy Modelling', Int. J. Heat and Mass Transfer, 38(8), pp856-884.
```

Then, further down in the source code immediately preceding the algorithm cite the appropriate reference from those specified at the beginning of the subroutine and include the specific page numbers, table numbers, equation numbers *etc*, as applicable. This can be helpful to your colleagues (and to yourself) in tracing bugs. An example follows.

```
      elseif( icor .EQ. 5 )then

C Coefficient correlation for a wall with a radiator located under
C a window (Smith and Jones publication, Table 2, Equation 6).
        hc = 2.30*(dt**0.24)
```

Describing assumptions

Whenever an assumption is made, add a comment specifying where the assumption came from. If the assumption can be referenced to a paper, report or book it should be. If it came from a discussion, explain why the assumption was made.

Annotating blocks of code

Use comments to mark ends of blocks. This is useful in identifying blocks when IF, DO or WHILE constructs extend over many lines of code or where there are multiple embedded loops or conditional code blocks. An example follows.

```
      do jj=1,mpcdat

C Has iteration for this plant additional output been requested?
        if ( iPlt_Output_Iter_Flag(ii,jj) .ne. 1 ) then
          < line of code >
          < line of code >
          < line of code >
        endif   ! <- matches  if ( iPlt_Output_Iter_Flag(ii,jj)...
      enddo   ! <- matches do jj = 1, ...
```

Grouping lines of code

Code annotations should be succinct but sufficiently detailed so that the purpose of every line of the code is obvious. If in doubt, say more. Annotate code by grouping lines logically. An example follows.

```
        <preceding code fragment>

C Override the calculated values if user has specified a convection file
C with fixed coefficients (ie. 'type 1' control over convection calculations).
C-------Does a convection file exist?
        IF( IHC(ICOMP).EQ.1 ) THEN
C---------Has the user specified fixed coefficients in the convection file?
          Ltype1 = 0
          DO 22 k=1,NHCFP(ICOMP)
            if( iCTLTP(ICOMP,k).eq.1 ) Ltype1=1
   22     CONTINUE

        <following code fragment>
```

In-line comments

Comments can be added to the end of any code line using the '!' character. This makes commenting blocks of variable definitions and short code lines easier to read. However, in-line comments must not be embedded within statements spanning multiple lines as some compilers will experience problems when parsing such comments.

Coding Style and use of FORTRAN/C

Source code files and subroutines

Long subroutines can be cumbersome to read, understand and test. It is good practice to keep the size of subroutines small and use calls to other subroutines when a distinct set of calculations need to be performed. If there is no distinct grouping of calculations, it is preferable to keep all calculations together. Further, if a set of equations is used in more than one area, the set should be relocated into its own subroutine.

When adding significant new functionality, related subroutines should be grouped into a file and this file located within an appropriate directory. There is no practical limit to file size, however files should contain subroutines that perform related functions or relate to a common theme or purpose. For example, the static template, coefficient generator and related subroutines for a new plant component should each be grouped into a single file and this file added to the *plt* directory.

Saved variables, global variables and common blocks

Any code that depends on static variables should explicitly declare these variables with a SAVE statement.

When multiple routines reference a large set of global variables held in a COMMON statement, consider placing the COMMON declarations in a header file. This reduces the risk of mismatches between the type and dimension of declared global variables, which invariably lead to segmentation faults.

Equations

Equations should be evaluated in the same manner that they would be written in a scientific publication. Avoid combining terms (and hence distorting clearness) in an attempt to produce computationally efficient code.

Working with existing files

When working with existing source code files maintain compatibility with the style.

Formatting source code

- Code line length should be confined to 72 characters.
- Use the '&' character for line continuations (6th column).
- Do not use tab characters within the source code, including comment lines. If using a file editor which allows tab characters, be certain to configure it such that tab characters are converted to spaces upon saving the file.
- All loop statements (IF-THEN, DO, WHILE) should be indented by 2 characters.

GOTO statements

GOTO statements should be avoided. There are plenty of alternatives in modern FORTRAN (e.g. the WHILE statement). A possible exception is when working with an existing subroutine that makes extensive use of GOTO statements in which case it may be clearer to maintain the style.

Floating point comparisons

All possibilities for division by zero must be trapped at the highest possible level and floating point comparisons avoided because variation in compilers and machine architectures can produce unexpected behaviour when the differences in values approach machine precision. For instance, the following example may produce inconsistent results:

```
      real x, y, z
      if ( y .eq. 0.0 ) then
        stop   "Divide by zero error in Subroutine ABC: y is zero!"
      else
        z = x / y
      endif
```

By comparing the difference between the floating numbers with a tolerance (presumably one several orders of magnitude less than the variation between the variables), these inconsistencies can be eliminated:

```
      real x, y, z
      parameter ( small = 1.0E-06 )
      if ( abs(x-y) .lt. small ) then
        stop "Divide by zero error: y is zero!"
      else
        z = x / y
      endif
```

ESP-r contains a subroutine named ECLOSE that can be used to facilitate such comparisons:

```
C Check if airflow rate is zero
            call eclose(convar(icon1,2),0.0,0.001,CloseD)

C If air flow rate is not zero, perform dew point temperature check;
C otherwise do not.
            if (.not. CloseD) then
```

Variable and subroutine names

Variable and subroutine names should be descriptive and clearly delineated from other names. Most modern FORTRAN compilers can support long names.

Generic variable names (e.g. INTEGER i, j, k; LOGICAL Done) for loop controls are discouraged. Instead, use descriptive names (e.g. INTEGER iZone, iSurface, iLayer; LOGICAL Loop_Unconverged).

When working with numerical constants or integer flags, define meaningful symbolic parameters to represent the constant or flag.

Documenting local and common block variables

COMMON block variables should be documented in the subroutine where they are first introduced. When variables are defined in the code, the comment should include the units of the variable (e.g., *J*, *kW*, °C, *K etc*).

Local variables should be documented in the subroutines in which they are used.

Two examples that reveal acceptable styles for documenting variables follow.

```
C Maximum infiltration ('finfmax') and ventilation ('fvntmax') flow
C rates (m^3/sec) for each zone. Variable 'icompforinf' is the component
C number associated with unique infiltration flow paths while 'isrczforvent'
C is the source zone associated with the largest ventilation rate.
      dimension finfmax(mcom), fvntmax(mcom), icompforinf(mcom)
      dimension icompforvent(mcom),isrczforvent(mcom)
      integer icompforinf,icompforvent,isrczforvent
      real finfmax,fvntmax
```

```
      real fCyl_Volume       ! Cylinder gas volume (m3)
      real fCyl_solid_mass   ! Mass of cylinder wall (kg)
      real fCyl_solid_Cp     ! Specific heat of cylinder wall (J/kg oC)
      real fCyl_UA_ambient   ! Heat transfer coeff. between cylinder
                             !   & ambient (W/oC)
```

FORTRAN version

All code must be compilable by the GNU f77 FORTRAN compiler.

Explicit declarations

Implicit type casting must be avoided. All new subroutines must include an 'IMPLICIT NONE' statement. This requires that all variables be defined before use and forces programmers to give proper consideration to their name, use, type and documentation. Explicit declarations also reduce debugging effort and strengthen confidence in the program's validity, as mismatches in variable names will produce errors at compile time.

Many existing ESP-r subroutines use the IMPLICIT rule to automatically define all variables that start with the letters 'I' through 'N' as INTEGER and all other variables as double precision REAL. This naming convention should be followed when altering existing subroutines that use the IMPLICIT rule, but any new variables added to the routine should be explicitly declared.

Type casting

Data must often be converted between real and integer formats during execution. Rather than rely on the compiler's native type casting behaviour, such conversions must always be done explicitly:

```
            real x, y
            integer i

            x = 1                    ! <- Implicit casts
            i = x                    !    (bad practice)
            y = i

            x = 1.0                  ! <- Explicit casts
            i = int(x)               !    (good practice)
            y = float(i)
```

FORTRAN/C parameter passing conventions

ESP-r technical modules are primarily written in FORTRAN with graphics implemented in C (e.g. X11 or GTK library calls). An intermediate layer of C code exists to mediates between the low level graphics calls and the primary FORTRAN code. Typically FORTRAN calls C although there are cases where the reverse is true.

There are a number of established patterns for passing integers, reals, characters and arrays between the two languages, which work with a number of compilers and across platforms. Examples can be found in esru_lib.F and esru_x.c both located in the 'lib' directory. (For example, for every string array passed to C, the C parameter list includes an additional 'int' that holds the array length.)

As a rule, FORTRAN never calls directly to the low level graphic functions although C will occasionally call FORTRAN to request information.

A limited number of C++ source files are associated with ESP-r. Passing conventions are less established for this.

In some cases '#ifdef' statements are used to signal differences between the X11 and GTK implementations or to differentiate between F90 and F77 code. Ifdefs are not needed for Windows/Linux/Unix differences as an *isunix* function is provided.