.

Unified Model
Documentation Paper No. 3

SOFTWARE STANDARDS for the UNIFIED MODEL: FORTRAN and UNIX

Version 7.2
5/2/98

Phillip Andrews

Model version 4.4

Numerical Weather Prediction
Meteorological Office
London Road
BRACKNELL
Berkshire
RG12 2SZ
United Kingdom

Modification Record		
Document version	**Author**	**Description.....**
7.1	M.Hatton	Subroutine modification histories to contain Unified model version number, not subroutine version number. The previous version 7 does not need to be kept as usual (for the use of external users who may be using old model versions) because subroutine versions were never regarded as correct practice since the issue of version 7. Some locations of internal Met.Office files have been corrected; the old locations do not exist any more, and so were useless information.
7.2	M.Hatton	Removal of system component and task numbers from headers. Removal of instructions about writing external documentataion.

Contents:

1.0 Introduction

This document specifies the software standards to be used when writing new routines for the Unified Model (UM). When making extensive changes to an existing routine a rewrite should be done to ensure that the routine meets these standards. An exception is that there is no requirement to rewrite imported code to these standards before it can be included in the UM. Where imported code has its own software standards modifications to that code should follow those original standards, ensuring that within a single routine only one set of standards is followed. New routines for imported code should follow these standards. The standards set out here are based on previous UM software standards, but are somewhat more detailed and do deviate from them in one or two instances. A modern programming style has been adopted, similar to that which we should use for Fortran 90. This is to help simplify the transition to the new language. Use is made of commonly found extensions to FORTRAN 77 which are also in the fortran 90 standard.

In the UM, code is divided into control and meteorological routines. Comdecks, provided by the nupdate precompiler, are used for global variables. These are all covered by this standards document.

The plug compatibility rules for routines dealing with physical parameterizations detailed in "Rules for Interchange of Physical Parameterizations" Kalnay et al. 1989 , Bull. A.M.S. 70 No. 6 p 620 are included in the standards defined here.

1.1 Why Have Standards?

This document is intended for new computer users as well as experienced programmers, so a few words about why there is a need for software standards at all may be in order. The aim of software standards is to reduce portability and maintainability problems. Remember that software should be written for people and not just for computers! As long as the syntax rules of the programming language (eg FORTRAN 77) are followed the computer does not care how the code is written: you could use archaic language structures; add no comments; leave no spaces etc. However, another programmer trying to use, maintain or alter the code certainly does care. A little extra effort whilst writing the code can greatly simplify the task of this other programmer (which might actually be the original author a year or so after writing the code when details of it are bound to have been forgotten). In addition, following these standards may well help you to write better, more efficient, programs containing fewer bugs.

1.2 Units

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

The standard SI prefixes are:

Prefix	Symbol	Multiplication Factor
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	k	10^{3}
hecto	h	10^{2}
deca	da	10^{1}
deci	d	10^{-1}
centi	c	10^{-2}
milli	m	10^{-3}
micro	µ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}

1.3 Code Development Stages

The preparation of new routines and of changes to existing routines should, in principle, go through all the following stages:

a) Plain language specification of purpose
   i.e. specify what needs to be done.

b) Draft documentation (scientific &/ or technical)
   i.e. decide how to do it.

c) Program design
   i.e. decide how to implement the solution.

d) Code written (review first routine to check adherence to standards)
   i.e. code it.

e) Code working

   i.e. bugs removed.

f) Code tested/ evaluated
   i.e. make sure the change was worth while / beneficial.

f) Code reviewed
   i) Code tested/ evaluated
      i.e. make sure the change was worth while / beneficial, and has no unexpected/ undesirable side effects.

   ii) Software standards check

i.e. make sure software standards are adhered to.

iii) Finalization of documentation
i.e. make sure it is up to date and correct.

i) Code accepted into the UM
- once any changes requested by the reviewer have been made & checked.

In practice not all stages will be relevant for all changes e.g. a bug fix, or tidying up maintenance task may not necessarily require alteration of the documentation.

1.4 The Rest of this Document

Details of the standards are given in section 2; section 3 contains modifications for control routines; and section 4 for comdecks. A concise summary is given in appendix A. Section 5 sets out standards for writing unix scripts. Section 6 makes a recommendation for monitoring the application of the standards, whilst section 7 considers QA Fortran, and section 8 makes a recommendation for the removal of nupdate commands. Appendix A contains a concise summary of the Fortran standards. The standard template subroutine header is reproduced (with minor alterations to make it fit an A4 page) in appendix B, and a very simple example Fortran met. subroutine is given in appendix C.

2.0 Standards for General Routines

2.1 Fortran 90 Compatibility Rules

The rules in this section are to make the code look as similar to fortran 90 as possible, though many could have been included on readability grounds alone. For full explanations of the rules specified in this section see books on Fortran 90 such as "Fortran 90 Explained" by Metcalf & Reid; "Programmers Guide to Fortran 90" by Brainerd, Goldberg & Adams; or our own in house document "Introduction to Fortran 90" by Hibling & Wardle.

2.1.1 The only symbol to be used as a continuation line marker is &.

2.1.2 The only symbol to be used for comments is !. Exception: Cray compiler directives in NUPDATE commands will have to remain as they are (or they wont work!).

2.1.3 Avoid using archaic FORTRAN 77 features and features deprecated Fortran 90.

2.1.3.1 Avoid COMMON blocks - use argument lists to pass variables into subroutines.

2.1.3.2 Avoid use of the EQUIVALENCE statement: only use them as a last resort.

2.1.3.3 Never use the PAUSE statement.

2.1.3.4 Never use assigned or computed GO TO statements.

2.1.3.5 Never use arithmetic IF statements.

2.1.4 Avoid using numeric LABELS wherever possible:

2.1.4.1 Loops MUST be terminated with an END DO statement. To improve the clarity of program structure you can optionally add comments after the Do & End do statements e.g.

```
Do i = 1, 100
  Do j = 1, 10
    ...code statements...
```

```
  End do  !j
End do !i
```

Use of comments like these (possibly with more detail) is required for both large DO loops and large IF blocks i.e. those spanning 15 lines or more.

2.1.4.2 Never use a FORMAT statement: they require the use of labels, and obscure the meaning of the I/O statement. The formatting information can be placed explicitly within the READ, WRITE or PRINT statement, or be assigned to a CHARACTER variable in a PARAMETER statement in the header of the routine for later use in IO statements. Never place output text within the format specifier: i.e. only format information may be placed within the FMT= part of an I/O statement, all variables and literals, including any character literals, must be 'arguments' of the I/O routine itself. This improves readability by clearly separating what is to be read/ written from how to read/ write it.

2.1.4.3 Avoid the use of the GO TO statement.

2.1.4.3.1 Never use a GO TO to jump upwards in the code.
2.1.4.3.2 A GO TO may only jump to a CONTINUE statement. Note: this deliberately differs from the previous UM standard as it allows code, for example to report that an error has occurred, to be placed before the RETURN statement.

2.1.4.3.3 All GO TO's must be commented to explain why it is there and what it is doing.

2.1.4.3.4 An acceptable use of GO TO is to jump to the end of a routine after the detection of an error, in which case you must use 9999 as the label (then everyone will understand what GO TO 9999 means).With Fortran 90 this will be the only acceptable use of GO TO and even so this use should be restricted to situations where other coding techniques, such as If tests on ErrorStatus around blocks of code, are too unwieldy to be used.

2.1.5 Code should be restricted to 72 columns for the sake of being able to read the code on any VDU; reading nupdate line numbers etc...

2.1.6 Never access arrays outside of their declared bounds - the results are unpredictable and often undesirable.

2.2 General style rules

The rules in this section are designed to improve the readability of the code and to reduce the differences in the look of code produced by different programmers. Both are intended to reduce the overheads involved in maintaining or altering another programmers code. The 'golden rule' is to be consistent.

2.2.1 Use meaningful variable names (but do try to keep them reasonably short!).

2.2.2 Write well structured code making use of subroutines to separate specific subtasks. In particular all file I/O should be done through subroutines: this greatly facilitates the portability of the code. I/O to standard input/ output should use * rather than hard wired unit numbers. Subroutines should be kept reasonably short, say up to about 100 lines of executable code, but don't forget there are start up overheads involved in calling an external subroutine so they should do a reasonable amount of work.

2.2.3 Code does `NOT HAVE TO BE WRITTEN IN UPPER CASE ONLY` and in fact it does get a little wearing to be shouted at by uppercase only code. It is better to use case to emphasize the structure of the code (as in the Do loop example of 2.1.4.1). This is very

similar to the use of capitols at the start of sentences in English, and when combined with meaningful variable names can greatly enhance the readability of the code. You can also use case instead of _ to make meaningful variable names more readable e.g. input_file could be written as InputFile.

2.2.4 Indent code within Do, Do while and If blocks by 2 characters (OK 2 is arbitrary but we may as well all use the same number!).

2.2.4.1 Continuation lines should also be indented: either by 2 characters; or to ensure that equations line up in a readable manner, as appropriate.

2.2.4.2 Comments should also be indented to follow the structure of the code, but by one less character to make them more easily identifiable. You cant put ! in column 6 as it will be interpreted as a continuation line marker, so start comments in column 1 for unindented code.

2.2.5 Use blank space sensibly to improve readability. This takes very little effort but can make a big difference to readability. Leave blank characters between variables and operators; try to line up related code into columns.

For example, instead of:

```
! Initialize variables

          x=1
          MeaningfulName=3
          RealNumber=5.0
```

write:

```
! Initialize variables
          x              = 1
          MeaningfulName = 3
          RealNumber     = 5.0
```

Similarly use blank lines to improve readability and to emphasize code structure: for example leave a blank line before a comment line; leave a blank line before an End do, Else, Else if, or End if statement (or before the first of a series of these statements as in the Do example of 2.1.4.1) as this again helps to emphasize the code structure. For example:

```
! Start of example code
     ... code statements...

! Example If block
     If (IntVariable .eq. SomeValue) then

      ! an indented comment
       Do Loop = 1, EndLoop

        ! another indented comment
         ... code statements...

       End do
     Else if (IntVariable .eq. SomeOtherValue) then
      ! another indented comment
       ... code statements ...

     End if
```

2.2.5.1 Standardization of Fortran statements: it greatly simplifies global searches if

programmers are consistent in their use of spaces in FORTRAN keywords. Thus GO TO, ELSE IF, END IF , END DO etc may each be written either as two words separated by one blank character, or as single 'words', but within a particular program unit the usage must be consistent.

2.2.6 Only use block If statements i.e. always use 'then' and 'End if', as this improves the readability of the code.

2.2.7 All subroutines must have only one entry and only one exit point.

2.2.8 Functions should never alter their arguments (i.e. all arguments must be intent in).

2.2.9 Subroutine naming conventions. This is related to the UM submodel project, which aims to split the UM into independently callable submodels. The names of all subroutines within a given submodel (or possibly sub task within a submodel) will be given the same short (2 character) prefix, to be separated from the rest of the name by an underscore. This makes the purpose of routines much more obvious; clearly identifies related routines; and drastically reduces the chance of name clashes within the UM. Deck names should use the same prefix but without the underscore.

2.2.10 Never use STOP statements: if errors are detected make a controlled exit from the routine (see 2.1.4.3).

2.2.11 Always use the generic names of intrinsic functions. These are often more meaningful and it reduces the number of names to learn.

2.2.12 Avoid the use of 'magic numbers' that is numeric constants hard wired into the code. These are very hard to maintain and obscure the function of the code. It is much better to assign the 'magic number' to a variable or constant with a meaningful name and then to use this throughout the code. In many cases the variable will be assigned in a top level control routine and passed down usually via the argument list but possibly via a comdeck. This ensures that all subroutines will use the correct value of the numeric constant and that alteration of it in one place will be propagated to all its occurrences. Unless the value needs to be alterable whilst the program is running (e.g. is altered via I/O such as a namelist) the assignment in the top level routine should be made using a PARAMETER statement.
For example, instead of writing:

```
If (ObsType .eq. 3) then
```

specify in the header local parameter section:

```
INTEGER       SurfaceWind   !Obs type No for surface wind
  PARAMETER  (SurfaceWind = 3)
```

and then write:

```
If (ObsType .eq. SurfaceWind) then
```

2.2.13 Guidance for commenting

2.2.13.1 Comment freely.

2.2.13.2 Use comments to say what a particular section of code is doing. These comments should be numbered sequentially: 1.0, 2.0 etc for main sections; use the number after the . to sequentially number subsections of a main section. These comments should refer to particular equations in documentation papers.

2.2.13.3 Also use comments to tell another programmer what is being done e.g. work array now holds geopotential values; this loop does ... etc.

2.2.13.4 Place comments either on the same line as the code they are commenting on, or on the line immediately before it.

2.2.14 When calling subroutines, add comments to show the intent of the subroutine's arguments. For example, a call to the subroutine Example of appendix B might look like:

```
 Call Example (
&  x_len, y_len, constant, work1, work2, ! Intent (In)
&  answer)                                ! Intent (Out)
```

2.2.15 Do not leave tab characters in your code: replace them with the appropriate number of spaces. This will ensure that the code looks as intended when ported and also will avoid potential compilation problems.

2.3 Subroutine & Function Standard Headers

Headers are an immensely important part of any code as they document what it does, and how it does it. You should write as much of the header as possible BEFORE writing the code, as this will focus your mind wonderfully on what you are doing and how you intend to do it! I have provided template subroutine and function headers: it is a requirement of these standards to use them. They are kept under http://fr0800/umdoc/header.templates/ in the form umshed* and umfhed*

2.3.1 Within (but only within) the header of a routine Fortran statements (but not variable names) must be written in upper case.

2.3.2 The standard template for subroutine headers is reproduced in appendix B of this document. Text within the triangular < > brackets are to be replaced by the user appropriately. The header is divided into two parts. The first part (up to the "! Global variables" line) must be completed in its entirety. Unused sections in the second part (the "! Global variables" line and after) may be deleted e.g. if there are no parameters local to the subroutine, then delete the Local parameters heading. Of course the heading must be reinstated if a local parameter is added at some later stage.

2.3.3 All variables must be declared, and commented with a brief description. This increases understandability and reduces errors caused by misspellings of variables.

2.3.3.1 The IMPLICIT NONE statement ensures that all variables must be declared and must be used.

2.3.3.3 Subroutine arguments must be declared in the same order in the header as they appear in the subroutine statement. This order is not random but is determined by intent, variable dimensions and variable type. All input arguments come first, followed by all input/output arguments and finally by all output arguments. Within each intent all scalar arguments must come before all array arguments. Within each of these divisions arguments must be in the same manner as specified in section 2.3.3.4. The comment for input/ output arguments must say what the argument is on both entry to and exit from the routine.

2.3.3.4 Within each section of part 2 of the header, variables of a given type should be grouped together. These groups must be declared in the order Integer, Real, Logical and then Character, with each grouping separated by a blank line. In general variables should be declared one per line, followed by a comment describing the variable and its purpose.

Use a separate type statement for each line as this makes it easier to copy code around (you can always use the editor to repeat a line to save typing the type statement again) and prevents you from running out of continuation lines. For example a section of part 2 of the header might look like the following:

```
...
! Local Parameters:
     INTEGER      InputUnit         !Unit number for input file
       PARAMETER (InputUnit = 23)

! Local scalars:
     INTEGER      IntValue1         !First integer variable
                                    !in example header
     INTEGER      IntValue2         !Second integer variable
                                    !in example header

     REAL         RealValue         !A variable of type real

! Local dynamic arrays:
     REAL         InputData(size)   !Array for data read from
                                    !input file.
...
```

2.3.4 Never use a separate DIMENSION statement. Array sizes should be declared in the type statement (as shown in the example of section 2.3.3.4).
2.3.5 History:
2.3.5.1 Add a new line to the History section when modifying existing code. This should include the date of the modification and a brief comment detailing the changes as well as the name of the person making the changes.

2.3.5.2 Version numbering: during the development phase of a project it may be helpful to use version numbers specific to each routine. When development is over & the code is ready for operational use then this header entry must contain the UM version number at which the change is put into the UM.

2.4 Error reporting

When it is possible that errors may occur, they should be detected and appropriate action taken. Errors may be of 2 types: fatal errors requiring program termination; and non fatal errors, which don't.

2.4.1 ErrorStatus
2.4.1.1 ErrorStatus is an integer variable used to store the current error state. A value of 0 means no (fatal) error, whilst a positive non zero value means a fatal error has occurred. Negative values are used for non fatal errors.

2.4.1.2 ErrorStatus should only be used by a routine if the value of ErrorStatus could change in that routine or in any routine called by it.

2.4.1.3 All routines which use ErrorStatus should check its value on entry (as a double check in case the calling routine failed to check for a bad ErrorStatus) and after any process which could change its value. A controlled exit should be made if a positive value is detected, possibly by following the procedure outlined in section 2.1.4.3.

2.4.1.4 Strictly speaking ErrorStatus is an IN/OUT variable. However, it is probably clearer to separate it from the other variables in the argument list. So, in an exception to the normal ordering rules for subroutine arguments, if present ErrorStatus must be the last variable in an

argument list.

2.4.1.5 On the initial detection of a fatal error the value of ErrorStatus must be set to a non zero positive value (+1).

2.4.1.6 On the initial detection of a non fatal error, ErrorStatus may be set to any non zero negative value. This value may then be used by the calling routine to take appropriate action.
2.4.2 ErrorReport
2.4.2.1 Once the nature of the error has been determined ErrorReport must be called with an appropriate ErrorMessage. ErrorReport will reset negative ErrorStatus values to zero.

2.4.2.2 The arguments of ErrorReport are:

```
 Call ErrorReport (NoOfLines, NameOfRoutine, ErrorMessage,
&  ErrorStatus)


 !   Scalar arguments with intent(in):
 INTEGER       NoOfLines            !Number of lines of output in
                                    !ErrorMessage.
 CHARACTER*40  NameOfRoutine        !Name of the routine calling
                                    !ErrorReport.
 !   Array arguments with intent(in):
 CHARACTER*80  ErrorMessage(NoOfLines)!Text for output.

 !   ErrorStatus
 INTEGER        ErrorStatus
```

ErrorReport writes an error message consisting of the type of error: fatal (ErrorStatus +ve) or warning (ErrorStatus -ve); the name of the calling routine; and ErrorMessage. It also resets ErrorStatus to zero if it was -ve on input. At present ErrorReport merely writes to standard output but this could easily be extended, such as directing output to any user specified file, should the need arise.

2.4.2.3 All routines using ErrorStatus should check for a +ve value before exiting, calling ErrorTrace (usually with a blank message) if true. This will provide a trace back mechanism showing the calling tree to the routine which reported the error.

2.4.3 ErrorTrace is a routine used to provide a calling tree on detection of an error.i

2.4.3.1 All routines using ErrorStatus should check for a +ve value before exiting, calling ErrorTrace (usually with a blank message) if true e.g.

```
If (ErrorStatus .gt. 0) then
  Call ErrorTrace ("NameOfRoutine", " ")

End if
```

2.4.3.2 The arguments of ErrorTrace are:

```
Call ErrorTrace ("NameOfRoutine", "message")

!   Scalar arguments with intent(in):
CHARACTER*40   NameOfRoutine        !Name of the routine calling
                                    !ErrorTrace
CHARACTER*80   message              !A one line message
```

The character variable message can be used to report additional information. For example, if the error occurred in a subroutine called inside a loop, message could be used to output the value of the loop counter.

2.4.4 Location of ErrorReport & ErrorTrace:
These will be included in the UM at version 3.4. Until then, they may be used by including the following modset:

um1.prmod304.mods(PA110394)

2.5 **Plug Compatibility Rules**
2.5.1 A package shall refer only to its own subprograms and to the intrinsic functions included in the ANSI FORTRAN 77 standard.

2.5.2 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. The running procedure must not alter these values.

2.5.3 All communication with the package shall be through the argument list at the entry points.

2.5.4 The package must not use unnamed (blank) common.

2.5.5 The horizontal index shall be the innermost of FORTRAN arrays. The range of this index processed on each call shall be specifiable via the argument list.

2.5.6 The number of vertical levels the package uses shall be specifiable via the argument list.

2.5.7 All dimensions of dummy argument arrays must appear in the argument list.

2.5.8 All I/O from the package must be limited to diagnostic output written to FORTRAN units specified in the argument list.

3.0 Control Routine Standards

These are essentially the same as given in section 2 for general routines, but are where most of the features to be avoided will occur such as the use of EQUIVALENCE or of COMMON blocks. It is also at this level that physics packages must meet the plug compatibility rules, in which all model variables are required to be passed by argument.

As a requirement for dynamic allocation of primary data arrays within the model it is necessary to pass lists of arrays by argument down through several levels of control routines before the meteorological routines are accessed. The lists of arrays and their type declarations should be defined by *COMDECKs to ensure that recognizable blocks of information are common throughout the control structure and to facilitate maintenance. Each list of arrays associated with a functional component used extensively throughout the model (such as diagnostic processing) is described by *CALL ARGxxxx with a corresponding type declaration *CALL TYPxxxx. In addition, the number of arguments passed at the top control level should be minimized by the combining arrays into ’super arrays’. See UM Documentation Paper C1: ’Dynamic allocation of primary fields’ for more information.

4.0 Comdeck Standards

These are blocks of declarations of global variables and constants. Comdecks declaring physical constants may be called by any routine, but most other comdecks will be called only by control routines. Comdecks should follow the standards set out in section 2 as much as possible, but have their own standard templates under http://fr0800/umdoc/header.templates/ of the form umcomd* .

5.0 Unix Script Standards

This standard covers UM UNICOS shell scripts which are used in the operational suite. This includes the automatic output processing subsystem and Operational Suite scripts. The requirements that this standard is intended to meet are as follows:

i) The script should be easily understood and used, and should be easy for a programmer other than the original author to modify.

ii) To simplify portability it should conform to the unix standard as much as possible, and exclude obsolescent and implementation-specific features when possible.

iii) It should be written in an efficient way.

iv) The structure of the script should conform to the design agreed in the project plan.

5.1 External Documentation of scripts

Scripts are to be regarded as being control code as far as external documentation is concerned.

5.2 Coding of scripts

Wherever possible the style and philosophy of the Fortran standards should be applied when writing unix /unicos scripts.

5.2.1 Scripts must use the standard script header, at 4.3 this is under http://fr0800/umdoc/header.templates/ of the form umscrpt*

5.2.2 The Bourne shell should be used unless this is impossible without compromising efficiency.

5.2.3 The logical structure of the script is to be determined by for, if, while, until and case constructs.

5.2.4 The first section of each script must initialize the local environment variables.

5.2.5 Error exits must return an explicit error code via "exit £E_COND", where E_COND is an environment variable described in a header comment and set either in the initialisation section of the script or is imported.

5.2.6 Error return codes from lower level scripts or executables are to be checked, with appropriate action taken in the event of an error.

6.0 Code reviews

In order to ensure that these standards are adhered to and are achieving the desired effect code reviews must be held. Reviews can also be useful in disseminating computing skills. To this end two types of code review are proposed: the first, to be called simply a code review, is a preacceptance check that the code accomplishes the specified task, and adheres to the software standards set in this document. See section 6.1 for a full description of the reviewer’s duties.

The second type of review is a group code review. These group code reviews should be attended by several programmers working in the same area (e.g. on the same project) and act as a mini workshop on software design and implementation allowing us to learn good programming techniques from each other. Clearly not too much time should be spent holding group code reviews so the following schedule is proposed. The first routine written under these standards by each programmer should be group reviewed. This should pick up any problems, misinterpretation of the rules etc. quickly, and therefore simplify their correction. It will also ensure that people new to the office are made aware of these standards from the start. Subsequent reviews should be held at regular, say 6 monthly, intervals with the programmer chosen on a rota basis from the code review group.

6.1 Instructions for Reviewers

Essentially reviewers must complete stage f) of section 1.3.

6.1.1 Reviewers must understand the routine or section under review and work through the logic to check that it fulfils the aims set out for the modification.

6.1.2 Reviewers must ensure that the updated code is written to the standards specified in this document.

6.1.3 Reviewers must ensure that the inline documentation describes the updated code.

6.1.4 Reviewers must ensure that external documentation is updated as necessary.

6.1.5 Reviewers must check that the modified code has been tested and any significant increase in memory usage or cpu time noted, and that this is communicated to the project management before acceptance of the code.

6.1.6 Disagreements: the reviewer can request evidence of tests and listings as necessary, and can demand action on non-observance of the above rules. The reviewer can make other comments as desired and in the case of disagreements these should be referred to project management for a decision on their implementation.

7.0 QA Fortran

QA Fortran is a potentially useful tool which you should run your Fortran routines through even if you (currently) decide to ignore some (possibly a large number) of its comments. Central computing are currently investigating ways of tailoring QA Fortran to test for compliance with the standards specified in this document. This would greatly enhance the value of QA Fortran. An examination of the QA Fortran report on your code should be a part of the code review (of either type).

8.0 Nupdate

Nupdate is an unsupported Cray product for managing code. It is currently extensively used throughout the UM. Clearly this causes portability problems and poses long term maintenance problems for the UM. Nupdate is used for two main purposes: as a preprocessor and as a code change management system. The preprocessor provides two main calls: *CALL which is very similar to the FORTRAN statement INCLUDE and so doesn't pose too many problems; and *IF, which provides a means of selecting (or deselecting) parts of the code for compilation. Use of *IF should be avoided as much as possible. In particular it should not be used when a FORTRAN IF statement could be used instead. To help with this a comdeck, C_GLOBAL, containing the INTEGER variable ModelType and the INTEGER parameters GlobalModel and LimitedAreaModel to test ModelType against, has been provided. Use of *IF to select different declarations or argument lists for global and limited area runs is strongly discouraged.

Appendix A: Concise Rules Summary

The rules discussed in the main text are reproduced here in summary form. Numbers in brackets after the rule refer to the appropriate section of the main text. The letter x has been used to represent any integer in one or two of these referrals.

1.0 Continuation line marker must be & (2.1).
2.0 Comments
2.1 The comment symbol must be ! (2.1.2).
2.2 Comments should be indented with the code and a blank line should be left before (but
not after) the comment line (2.2.5).
2.3 Comments should be used freely, and for 2 main purposes (2.2.13).
2.3.1 Numbered comments to say what a particular section, or subsection, of code is doing.
2.3.2 Comments to explain what is going on to another programmer reading the code.
3.0 Avoid archaic FORTRAN 77 features (2.1.3).
3.1 Never use PAUSE; assigned GO TO; arithmetic IF.
3.2 Avoid use of COMMON; EQUIVALENCE; DO WHILE.
4.0 Avoid labels (2.1.4.x).
4.1 Terminate loops with END DO
4.2 Never use FORMAT
4.3 Avoid use of GO TO (2.1.4.3.x).
4.3.1 Only jump to a CONTINUE statement.
4.3.2 Error trapping must use the label 9999.
5.0 For now, restrict line lengths to 80 columns (2.1.5).
6.0 Use meaningful variable names (2.2.1, 2.2.3).
7.0 Use case sensibly (2.2.3).
8.0 Use blank space sensibly (2.2.5).
9.0 Indent code within DO or IF blocks by 2 characters (2.2.4).
9.1 Only use block IF statements (2.2.6).
10.0 Avoid using "magic numbers" (2.2.12).
11.0 Only use the generic names of intrinsic functions (2.2.11).
12.0 Subprograms
12.1 Can have only one entry & only one exit point (2.2.7).
12.2 Functions must not alter variables passed to them via their argument list (2.2.8).
12.3 Use the naming convention (2.2.9).
12.4 Use standard subroutine headers (2.3.x).
12.4.1 Uppercase only fortran keywords within the header.
12.4.2 Use implicit none.
12.4.3 Use the correct order for variables in argument list.
12.4.4 Declare arguments in same order as in argument list.
12.4.5 Declare arguments & variables in the correct manner.
12.4.7 Fill in other header items.
12.5 Use comments to show the intent of subroutine arguments in a CALL to a subroutine (2.2.14).
13.0 Error reporting (2.3).
13.1 Set ErrorStatus to non zero value after fatal error & exit gracefully.

Appendix B: Standard Fortran Header for Met. Subroutines

```
!+ <A one line description of this subroutine>
! Subroutine Interface:
     SUBROUTINE <name(InputArguments, InOutArguments,
    &  OutputArguments)>
     IMPLICIT NONE
! Description:
!   <Say what this routine does>
!
! Method:
!   <Say how it does it: refer to external documentation>
!   <If this routine is very complex, then include a
!    "pseudo code" description of it to make its structure
!    and method clear>
!
! Current Code Owner: <Name of person owning this code>
!
! History:
! Version  Date     Comment
! =======  ====     =======
!<Number>  <date>   Original code. (<Your name>)
!
! Code description:
!   FORTRAN 77 + common extensions also in fortran 90.
!   This code is written to UM programming standards version 6
!
! Declarations: these are of the form:-
!     INTEGER      ExampleVariable  !Description of variable
!
! Global variables (*CALLed common blocks etc.)

! 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):

!   ErrorStatus:
!     INTEGER      ErrorStatus      !+ve = fatal error
! Local parameters:
! Local scalars:
! Local dynamic arrays:
! Function & Subroutine calls:
     External
!- End of header
```

Appendix C: Very Simple Example Fortran Met. Subroutine

```
!+ Multiplies two 2D arrays together and adds a constant.
!
! Subroutine Interface:
     SUBROUTINE Example (x_len, y_len, constant, Input1,
    &  Input2, Output)
     IMPLICIT NONE
!
! Description:
!   Noddy routine to multiply two 2 dimensional arrays
!   together, and add a constant to the result.
!
! Method:
!   The arrays are multiplied element by element, and a
!   constant is added to the result.
!
! Current Code Owner: Phil Andrews
!
! History:
! Version  Date     Comment
! =======  ====     =======
! 3.4      15/10/93 Original code. (Phil Andrews)
!
! Code description:
!   FORTRAN 77 + common extensions also in fortran 90.
!   This code is written to UM programming standards version 6
!
! Declarations: these are of the form:-
!     INTEGER      ExampleVariable  !Description of variable

! Subroutine arguments
!   Scalar arguments with intent(in):
      INTEGER      x_len            !Length of first dimension
                                    !of the arrays.
      INTEGER      y_len            !Length of second dim.
                                    !of the arrays.

      REAL         constant         !Value to be added

!   Array arguments with intent(in):
      REAL         Input1(x_len, y_len) !First input array
      REAL         Input2(x_len, y_len) !Second input array

!   Array arguments with intent(out):
      REAL         Output(x_len, y_len) !Contains the result

! Local scalars:
      INTEGER      x                !Loop counter over x dim.
      INTEGER      y                !Loop counter over y dim.

! End of header


! 1.0 Start of subroutine code: perform the calculation.
      Do y = 1, y_len
        Do x = 1, x_len
         ! Calculate the Output value:
          Output(x, y) = (Input1(x, y) * Input2(x, y))
     &                   + constant

        End do !x
      End do !y

! End of subroutine code.
      Return
      End
```