Unified Model Documentation Paper 003

Software Standards

UM Version	**:**	10.6
Last Updated	**:**	2016-07-22 (for vn10.6)
Owner	**:**	Joe Mancell

Contributors:
G. Greed and J. Mancell

Met Office
FitzRoy Road
Exeter
Devon EX1 3PB
United Kingdom

Contents

1 Introduction

This document specifies the software standards and coding styles to be used when writing new code files for the Met Office Unified Model. When making **extensive** changes to an existing file a **rewrite** of the whole file should be done to ensure that the file meets the UM coding standard and style. **All code modifications within an existing file should follow these standards**.

In the UM, code is divided into control and meteorological files. The C pre-processor is used to make machine specific and scientific choices. These are all covered by this standards and style document. The use of additional pre-processor flags is discouraged as work is underway to retire as many as possible.

The only exception to following these coding standards is that there is no requirement to rewrite 'imported code' to these standards before it is included within the UM. All new code developed within the Met Office should follow these standards.

Imported code; is developed as part of a collaboration project and then proposed to be suitable for use within the UM; for example the original UKCA code developed in academia. Collaborative developed code specifically for the UM should meet these standards.

1.1 Why have standards?

This document is intended for new as well as experienced programmers, so a few words about why there is a need for software standards and styles may be in order.

Coding standards specify a standard working practice for a project with the aim of improving portability, maintainability and the readability of code. This process makes code development and reviewing easier for all developers involved in the project. Remember that software should be written for people and not just for computers! As long as the syntax rules of the programming language (e.g. Fortran IV – 2008) 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 will have trouble working out what the code does and how it does it. A little extra effort whilst writing the code can greatly simplify the task of this other programmer (which might 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.

While code style is very subjective, by standardising the style, UM routine layout will become familiar to all code developers/reviewers even when they are not familiar with the underlying science.

1.2 Units

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

1.3 Working practices

The preparation of new files and of changes to existing files should, meet this UM standard documentation and must be developed following the stages outlined in " Working Practices for UM Development under FCM".

1.4 Examples

This document provides an example programming unit to aid the code developer. This example meets the standards detailed within this paper, with references to the relevant sections.

1.5 Fortran 95/2003

As most major Fortran compilers support the majority of the Fortran 2003 standards, from UM8.5 Fortran 2003 code is acceptable for inclusion in the UM. Albeit as we aim to ensure UM portability, some Fortran features are excluded at this time. For further advice please see B.

2 How to meet the coding standards

The following code example exhibits all that is defined as a good coding standard and how code should be written for inclusion within the UM.

The example is highlighted with references (section links) to the remainder of this document which provide further details on the standard and style used.

example_mod.F90 3.1

```
! *****************************COPYRIGHT*******************************  3.2
! (C) Crown copyright Met Office. All rights reserved.
! For further details please refer to the file COPYRIGHT.txt
! which you should have received as part of this distribution.
! *****************************COPYRIGHT*******************************
!
! An example routine depicting how one should construct new code
! to meet the UMDP3 coding standards.                                    3.2
!
MODULE example_mod                                                       3.3 3.4 3.6

IMPLICIT NONE                                                            3.7

! Description:                                                           3.2
!   A noddy routine that illustrates the way to apply the UMDP3
!   coding standards to new code to help code developers
!   pass code reviews.
!
! Method:
!   In this routine we apply many of the UMDP3 features                  3.2
!   to construct a simple routine. The references on the RHS take the reader
!   to the appropriate section of the UMDP3 guide with further details.
!
! Code Owner: Please refer to the UM file CodeOwners.txt                 3.2
! This file belongs in section: Control
!
! Code description:                                                      3.2
!   Language: Fortran 95.
!   This code is written to UMDP3 standards.

CHARACTER(LEN=*), PARAMETER, PRIVATE :: ModuleName='EXAMPLE_MOD'         3.15

CONTAINS                                                                 3.1

! Subroutine Interface:
SUBROUTINE example (xlen,ylen,input1,l_unscale,input2 &                  3.10
                                    output, l_loud_opt)

! Description:
!   Nothing further to add to module description.                        3.2
USE atmos_constants_mod,    ONLY: r                                      3.6
USE ereport_mod,            ONLY: ereport
USE parkind1,               ONLY: jpim, jprb                             3.15
USE umprintMgr,             ONLY: umprint,ummessage,PrNorm               3.12
USE errormessagelength_mod, ONLY: errormessagelength
USE yomhook,                ONLY: lhook, dr_hook                         3.15
```

```
IMPLICIT NONE                                                                    3.7

! Subroutine arguments
INTEGER, INTENT(IN) :: xlen  !Length of first dimension of the arrays.           3.7
INTEGER, INTENT(IN) :: ylen  !Length of second dimension of the arrays.

REAL, INTENT(IN)    :: input1(xlen, ylen) !First input array                     3.7

LOGICAL, INTENT(IN) :: l_unscale  ! switch scaling off.

REAL, INTENT(INOUT) :: input2(xlen, ylen) !Second input array                    3.7
REAL, INTENT(OUT)   :: output(xlen, ylen) !Contains the result                   3.7

LOGICAL, INTENT(IN), OPTIONAL :: l_loud_opt !optional debug flag                 3.7

! Local variables
INTEGER(KIND=jpim), PARAMETER :: zhook_in  = 0  ! DrHook tracing entry           3.7 3.15
INTEGER(KIND=jpim), PARAMETER :: zhook_out = 1  ! DrHook tracing exit
INTEGER :: i                ! Loop counter
INTEGER :: j                ! Loop counter
INTEGER :: icode            ! error code for EReport
LOGICAL :: l_loud           ! debug flag (default false unless l_loud_opt is used) 3.7

REAL, ALLOCATABLE :: field(:,:)     ! Scaling array to fill.                     3.8
REAL(KIND=jprb)   :: zhook_handle   ! DrHook tracing                             3.15

CHARACTER(LEN=*), PARAMETER        :: RoutineName='EXAMPLE'                      3.19
CHARACTER(LEN=errormessagelength)  :: Cmessage ! used for EReport
CHARACTER(LEN=256)                 :: my_char  ! string for output

! End of header
IF (lhook) CALL dr_hook(ModuleName//':'//RoutineName,zhook_in,zhook_handle)      3.15

! Set debug flag if argument is present
l_loud = .FALSE.
IF (PRESENT(l_loud_opt)) THEN                                                    3.7
  l_loud = l_loud_opt
END IF

my_char                                                 &                        3.10
    =  'This is a very very very very very very very '  &
    // 'loud character assignment'    ! A pointless long character example.
icode=0

! verbosity choice, output some numbers to aid with debugging                    3.5
! protected by printstatus>=PrNorm and pe=0
WRITE(ummessage,'(A,I4)')'xlen=',xlen                                            3.12
CALL umprint(ummessage,level=PrNorm,pe=0,src='example_mod')                      3.13
WRITE(ummessage,'(A,I4)')'ylen=',ylen
CALL umprint(ummessage,level=PrNorm,pe=0,src='example_mod')
IF (l_loud) CALL umprint(my_char,level=PrNormal,src='example_mod')

! Allocate and initialise scaling array                                          3.5
! Noddy code warns user when scalling is not employed.
IF ( l_unscale ) THEN                                                            3.9
  icode = -100  ! set up WARNING message
  ALLOCATE(field( 1,1 ) )                                                        3.8
  cmessage='Scaling is switched off in run!'
```

```
  CALL ereport(RoutineName,icode,cmessage)                                    3.19
ELSE
  ALLOCATE(field( xlen, ylen ) )                                              3.8
  DO j=1,ylen                                                                 3.9
    DO i=1,xlen
      field(i, j)  = (1.0*i) + (2.0*j)                                        3.4
      input2(i, j) = input2(i, j) * field(i, j)
    END DO
  END DO
END IF

! The main calculation of the routine, using OpenMP.                         3.5
!$OMP  PARALLEL DEFAULT(NONE)                                    &            3.16
!$OMP  SHARED(xlen,ylen,input1,input2,field,output)               &
!$OMP  PRIVATE(i, j )                                                        3.16
!$OMP DO SCHEDULE(STATIC)
DO j = 1, ylen
  i_loop: DO i = 1, xlen                                                      3.9
    ! Calculate the Output value:
    output(i, j) = (input1(i, j) * input2(i, j))
  END DO i_loop
END DO ! j loop
!$OMP END DO                                                                 3.16
!$OMP END PARALLEL                                                           3.16

DEALLOCATE (field)                                                            3.8

IF (lhook) CALL dr_hook(ModuleName//':'//RoutineName,zhook_out,zhook_handle)  3.15
RETURN
END SUBROUTINE example                                                        3.4

END MODULE example_mod                                                        3.4
```

3 UM programming standards; Code Layout, Formatting, Style and Fortran features

This section outlines the programming standards you should adhere to when developing code for inclusion within the Unified Model. The rules set out in this section aim to improve code readability and ensure that UM code is compatible with both the Fortran 2003 standard and FCM.

3.1 Source files should only contain a single program unit

- Modules may be used to group related variables, subroutines and functions. Each separate file within the source tree should be uniquely named.
- The name of the file should reflect the name of the programming unit. Multiple versions of the same file should be named `filename-#ver` where `#ver` is the section/version number (e.g. 1a,2a,2b...). For example:
  - `<filename-#ver>.F90` when writing a `<subroutine>`
  - `<filename_mod-#ver>.F90` with writing a `<module_mod>`
  - `<existing filename>.F90` with `<module_mod>` only if upgrading existing subroutine since Subversion does not handle renaming of files very well and this allows history of the file to be easily retrieved.

  This makes it easier to navigate the UM code source tree for given routines.
- You should avoid naming your **program units** and **variables** with names that match an intrinsic `FUNCTION`, `SUBROUTINE` or `MODULE`. We recommend the use of unique names within a program unit.
- You should also avoid naming your program units and variables with names that match a keyword in a Fortran statement.
- Subroutines should be kept reasonably short, where appropriate, 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.

3.2 Headers

- All programming units require a suitable copyright header. Met Office derived code should use the standard UM copyright header as depicted in the good example code. Collaborative UM developed code may require alternative headers as agreed in the collaborative agreements. e.g. UKCA code. The IPR (intelectual property rights) of UM code is important and needs to be protected appropriately.
- 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 on what you are doing and how you intend to do it!
- The description of the `MODULE` and its contained `SUBROUTINE` may be the same and thus it need not be repeated in the latter. If a `MODULE` contains more than one subroutine then further descriptions are required.
- History comments should not be included in the header or routine code. FCM TRAC provides the history of our codes.
- Code author names should NOT be included explicitly within the code as they quickly become out of date and are sometimes misleading. Instead we reference a single maintainable text file which is included within the UM code repository.

```
! Code Owner: Please refer to the UM file CodeOwners.txt
! This file belongs in section: <section_name_to_be_entered>
```

- Example UM templates are provided with the source of this document; subroutine, function and module templates.

3.3 Free source form

- All code should be written using the free source form.
- Please restrict code to 80 columns, so that your code can be easily viewed on any editor and screen and can be printed easily on A4 paper.
- Never put more than one statement on a line.
- Write your program in UK English, unless you have a very good reason for not doing so. Write your comments in simple UK English and name your program units and variables based on sensible UK English words. Always bear in mind that your code may be read by people who are not proficient English speakers.

3.4 Fortran style

- To improve readability, write your code using the ALL CAPS Fortran keywords approach. The rest of the code may be written in either lower-case with underscores or CamelCase. This approach has the advantage that Fortran keywords stand out.
- To improve readability, you should always use the optional space to separate the following Fortran keywords:

```
ELSE IF      END DO          END FORALL    END FUNCTION
END IF       END INTERFACE   END MODULE    END PROGRAM
END SELECT   END SUBROUTINE  END TYPE      END WHERE
SELECT CASE
```

- The full version of `END` should be used at all times, eg `END SUBROUTINE <name>` and `END FUNCTION <name>`
- New code should be written using Fortran 95/2003 features. Avoid non-portable vendor/compiler extensions.
- When writing a `REAL` literal with an integer value, put a 0 after the decimal point (i.e. 1.0 as opposed to 1.) to improve readability.
- Avoid the use of old FORTRAN depreciated code, make use of the F95/2003 features.
- Never use the `PAUSE` statement.
- Never use the `STOP` statement, see 3.19
- The standard delimiter for namelists is `/`. In particular, note that `&END` is non-standard and should be avoided. For further information on namelists please refer to 4.1
- Only use the generic names of intrinsic functions, avoid the use of 'hardware' specific intrinsic functions. Use the latter if an only if there is an optimisation benefit and then it must be protected by a platform specific CPP flag 3.18.

3.5 Comments and white spacing

- Always comment code!
- Start comments with a single '!'. The indention of whole line comments should match that of the code.
- Use spaces and blank lines where appropriate to format your code to improve readability.
- Never use tabs within UM code as the tab character is not in the Fortran character set. If your editor inserts tabs automatically, you should configure it to switch off the functionality when you are editing Fortran source files.
- Line up your statements, where appropriate, to improve readability.

3.6 The use of modules

MODULEs are strongly encouraged as the mainstay of future UM code program units; making use of the implicit INTERFACE checking and removing the need for the !DEPENDS ON. Argument lists within SUBROUTINE CALLs may also shorten.

- You are expected to USE <module>, ONLY : <variables> and variables should be imported from the module in which they were originally declared thus enabling a code audit trail of variables around the UM code.
- For code portability, be careful not to USE <module> twice in a routine for the same MODULE, especially where using ONLY. This can lead to compiler Warning and Error messages.
- Where possible, module variables and procedures should be declared PRIVATE. This avoids unnecessary export of symbols, promotes data hiding and may also help the compiler to optimise the code.
- The use of derived types is encouraged, to group related variables and their use within Modules.
- Review your use of arguments within subroutine calls, could some be simplified by using Modules?
- Do not duplicate include files within a Module, either replace the include file with a Module or #include it within a module until one is able to migrate properly to a MODULE retiring the include file.
- Before writing your Module, check the UM source that no one has already created a Module to do what you want. For example do not declare a new variable/parameter without checking if it is already available in a suitable UM module.
- Global type constants (e.g. g and π) should be maintained at a high level within the UM code and not duplicated within modules at the code section level; USE <insert global consts module name here> instead. Only section specific constants should be maintained at the section level.
- When calling another Subroutine or an External Function the use of "! DEPENDS ON" directive is required within the Unified Model prior to the CALL unless the Subroutine or Function is wrapped within a Module; thus USE it,

```
! DEPENDS ON: gather_field_gcom
CALL gather_field_gcom(local_field,    global_field,       &
                       local_row_len,  local_rows,         &
                       global_row_len, global_rows,        &
                       grid_type,      halo_type,          &
                       gather_pe,      proc_group,         &
                       icode,          cmessage)
```

- Avoid the introduction of additional COMMON blocks. Developers should now be using MODULEs.

3.7 Argument and variable declaration

- Use IMPLICIT NONE in all program units. This forces you to declare all your variables explicitly. This helps to reduce bugs in your program that will otherwise be difficult to track.
- Use meaningful variable names to aid code comprehension.
- All variables must be declared, and commented with a brief description. This increases understandability and reduces errors caused by misspellings of variables.
- Use INTENT in declaring arguments as this allows for checks to be done at compile time.
- Arguments should be declared separately from local variables.
- Subroutine arguments should 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 then all output arguments. The exception being any OPTIONAL arguments which should be appended to the end of the argument list. If more than one OPTIONAL argument is used then one should also use keywords so that the OPTIONAL arguments are not tied to a specific 'position' near the end of the argument list.

- As OPTIONAL arguments are possible when using MODULEs (an interface is required) there is no requirement in future for DUMMY arguments and glue routines.
- It is recommended that one uses local variables in routines which are set to the values of optional arguments in the code if present, otherwise a default value is used. This removes the requirement to always use PRESENT when using the optional argument.
- Within each section 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. 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.
- If an array is dimensioned by another variable, ensure that the variable is declared first.
- The EXTERNAL statement should not be used for subroutines although is allowed for functions, again for code portability.
- Avoid the DIMENSION attribute or statement. Declare the dimension with the declared variables which improves readability.

  Common practice

  ```
  INTEGER, DIMENSION(10,20) :: a, b, c
  ```

  Better approach

  ```
  INTEGER :: a(10, 20), b(10, 20), c(10, 20)
  ```

- Where using includes, use include *.h as a file extension for #include files. Please migrate includes to Modules and do not create new include files! For some more information on control routine includes please refer to 4.4
- Initialisation in the declaration of a variable should only be done after considering whether it is to be only initialised on the first encounter of the variable or not. Fortran automatically adds SAVE to the declaration attribute to this type of intialisation. This is especially important in OpenMP and when you expect the variable to be reset everytime the routine is entered. POINTERs are also affected so please be aware of the effects.
- Character strings must be declared with a length when stored in an array.

3.8 Allocatables

- When Allocating and deallocating, use a separate ALLOCATE and DEALLOCATE statement for each array.
- When using the ALLOCATE statement, ensure that any arrays passed to subroutines have been allocated, even if it's anticipated that they won't be used.

```
IF (L_mcr_qrain) THEN
  ALLOCATE ( mix_rain_phys2(1-offx:row_length+offx,         &
                            1-offy:rows+offy, wet_levels)
ELSE
  ALLOCATE ( mix_rain_phys2(1,1,1) )
END IF

! DEPENDS ON: q_to_mix
CALL do_something(row_length, rows, wet_levels,           &
                  offx,offy, mix_rain_phys2   )
```

- To prevent memory fragmentation ensure that allocates and deallocates match in reverse order.

```
ALLOCATE ( A(row_length,rows,levels) )
ALLOCATE ( B(row_length,rows,levels) )
ALLOCATE ( C(row_length,rows,levels) )
....
```

```
DEALLOCATE ( C )
DEALLOCATE ( B )
DEALLOCATE ( A )
```

- Where possible, an ALLOCATE statement for an ALLOCATABLE array (or a POINTER used as a dynamic array) should be coupled with a DEALLOCATE within the same scope. If an ALLOCATABLE array is a PUBLIC MODULE variable, it is highly desirable for its memory allocation and deallocation to be only performed in procedures within the MODULE in which it is declared. You may consider writing specific SUBROUTINEs within the MODULE to handle these memory managements.
- Always define a POINTER before using it. You can define a POINTER in its declaration by pointing it to the intrinsic function NULL() (also see advice in 3.7). Alternatively, you can make sure that your POINTER is defined or nullified early on in the program unit. Similarly, NULLIFY a POINTER when it is no longer in use, either by using the NULLIFY statement or by pointing your POINTER to NULL().
- New operators can be defined within an `INTERFACE` block.
- `ASSOCIATED` should only be done on initialised pointers. Uninitialised pointers are undefined and `ASSOCIATED` can have different effects on different platforms.

3.9 Code IF blocks and DO LOOPs

- The use of comments is required for both large `DO` loops and large `IF` blocks; those spanning 15 lines or more, see 3.5
- Indent blocks of code by 2 characters.
- Use the following syntax for LOGICAL comparisons, i.e.:

```
== instead of .EQ.
/= instead of .NE.
>  intead of .GT.
<  instead of .LT.
>= instead of .GE. (do not use =>)
<= instead of .LE. (do not use =<)
```

- Positive logic is usually easier to understand. When using an IF-ELSE-END IF construct you should use positive logic in the IF test, provided that the positive and the negative blocks are about the same length.

  Common practice

```
IF (my_var != some_value) THEN
  CALL do_this()
ELSE
  CALL do_that()
END IF
```

  Better approach

```
IF (my_var == some_value) THEN
  CALL do_that()
ELSE
  CALL do_this()
END IF
```

- Where appropriate, simplify your LOGICAL assignments, for example:

Common practice

```
IF (my_var == some_value) THEN
  something      = .TRUE.
  something_else = .FALSE.
ELSE
  something      = .FALSE.
  something_else = .TRUE.
END IF
! ...
IF (something .EQV. .TRUE.) THEN
  CALL do_something()
  ! ...
END IF
```

Better approach

```
something      = (my_var == some_value)
something_else = (my_var /= some_value)
! ...
IF (something) THEN
  CALL do_something()
  ! ...
END IF
```

- 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 via a include file or module. 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 should be made using a `PARAMETER` statement.

  Poor Practice

  ```
  IF (ObsType == 3) THEN
  ```

  Better Approach

  ```
  ...specify in the header local constant section....

  INTEGER, PARAMETER :: SurfaceWind = 3 !No. for surface wind

  ...and then use in the logical code...

  IF (ObsType == SurfaceWind) THEN
  ```

- **Be careful** when comparing real numbers using ==. To avoid problems related to machine precision, a threshold on the difference between the two numbers is often preferable, e.g.

  Common practice

  ```
  IF ( real1 == real2 ) THEN
    ...
  END IF
  ```

  Better approach

  ```
  IF ( ABS(real1 - real2) < small_number ) THEN
    ...
  END IF
  ```

  where small_number is some suitably small number. In most cases, a suitable value for small_number can be obtained using the Fortran intrinsic functions `EPSILON` or `TINY`.

The UM perturbation sensitivity project is currently in the process of identifying coding issues that lead to excessive perturbation growth in the model. Currently, all problems are emerging at IF tests that contain comparisons between real numbers. Typical, real case UM examples of what can go wrong are detailed in appendix C of this document.

- Loops *must* terminate with an `END DO` statement. To improve the clarity of program structure you are encouraged to add labels or comments to the `DO` and `END DO` statements. This is especially helpful when using `EXIT`

```
DO i = 1, 100
  j_loop: DO j = 1, 10
    DO k = 1, 10
      ...code statements...
    END DO ! k
  END DO j_loop
END DO ! outer loop i
```

- Avoid the use of the `GO TO` statement.
  - The only 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).
  - UM Error reporting guidance is detailed in 3.19
- Avoid assigned `GO TO`, computed `GO TO`, arithmetic `IF`, etc. Use the appropriate modern constructs such as `IF`, `WHERE`, `SELECT CASE`, etc..
- Where possible, consider using `CYCLE`, `EXIT` or a `WHERE` construct to simplify complicated `DO` loops.
- Be aware that logic in `IF` conditions can be performed in any order. So checking that array is greater than lower bound and using that index is not safe.

  Common approach

```
DO j = 1, rows
  DO i = 1, row_length
    IF (cloud_level(i,j) > 0 .AND. cloud(i,j,cloud_level(i,j)) == 0.0) THEN
      cloud(i,j,cloud_level(i,j)) = 1.0
    END IF
  END DO
END DO
```

  Better approach

```
DO j = 1, rows
  DO i = 1, row_length
    IF (cloud_level(i,j) > 0) THEN
      IF (cloud(i,j,cloud_level(i,j)) == 0.0) THEN
        cloud(i,j,cloud_level(i,j)) = 1.0
      END IF
    END IF
  END DO
END DO
```

3.10 Line continuation

- The only symbol to be used as a continuation line marker is '`&`' at the end of a line. It is suggested that you align these continuation markers to aid readability. Do not add a second '`&`' to the beginning of the next line. This advice also applies to blocks of Fortran code protected by the OpenMP sentinel '`!$`'. The only currently allowed exception is to continuation lines used with OpenMP directives, i.e. '`!$OMP`', where the '`&`' marker may optionally be used. Please see section 3.16 for more advice on OpenMP.
- Short and simple Fortran statements are easier to read and understand than long and complex ones. Where possible, avoid using continuation lines in a statement.

- Try to avoid string continuations and spread the string across multiple lines using catenations (//) instead.

3.11 Fortran I/O

- When calling OPEN, ensure that the ACTION argument is specified. In particular, ACTION='READ' shall be used for files that are opened only for reading as this reduces file locking costs.
- Don't check for the existence of a file by using INQUIRE if the only action you'll take if the file doesn't exist is to report an error. Rather use OPEN(... , IOSTAT=icode, IOMSG=iomessage) and include the iomessage in an error message if icode is non-zero. This will capture a wider range of errors with fewer filesystem metadata accesses.

3.12 Formatting and output of text

Writing output to the "stdout" stream, commonly unit 6 in fortran must use the provided API, which is accesible by including USE umPrintMgr in the calling code.

- Single string output should be written as,

  ```
  CALL umprint('Hello',src='this_file')
  ```

  where this_file is the base part of the filename, this_file.F90

- Multi-component output must first be written to an internal file via WRITE statement. The umPrintMgr module provides a convenient string for this purpose; umMessage, though you may use your own.

  ```
  WRITE (ummessage,'(A,I2,A)') 'I am ', age, ' years old'
  CALL umprint(ummessage,src='this_file')
  ```

- Avoid the use of WRITE (ummessage,'*')
- Always add formatting information to your write statements. It is important to ensure that the output message fits within the string being written to. Numerical output without formatting information is heavily padded with leading blanks on Linux machines. This can cause output to exceed the length of the string being written to. This is a particular problem for error report messages written to internal files, which can cause the job to abort before ereport can output any useful information
- The character variable newline (from the umPrintmgr module), is recognised as a newline if embedded in the string passed to umPrint.
- Having used a format descriptor specifying field widths, the total line length should then be calculable and not exceed 80 characters.
- The format descriptor should not include vertical space, blank lines should be explicit;

  ```
  CALL umprint('',src='this_file')
  CALL umprint('This is important, so I want it to stand out.',src='this_file')
  CALL umprint('',src='this_file')
  ```

- Calls to umPrint should be protected by a suitable setting of the PrintStatus variable, see 3.13 either with conditional logic or an additional level argument,

  ```
  CALL umprint(ummessage,src='this_file',level=PrOper)
  ```

- If your output is not required from each processor protect the umPrint either with logic, or an additional pe argument, for example,

  ```
  ! We'll only output at diagnostic level on pe0
  CALL umprint(ummessage,src='this_file',level=PrDiag,pe=0)
  ```

- 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 I/O 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.

Common practice

```
 WRITE(Cmessage,                                                  &
&    '("Cannot run with decomposition ",I3," x ",I3,             &
&      " (",I3,") processors. ",                                 &
&      "Maxproc is ",I3," processors.")')                        &
&       nproc_EW,nproc_NS,nproc_EW*nproc_NS,Maxproc
```

Better approach

```
WRITE(cmessage,'(a,i3,a,i3,a,i3,a,i3,a)')                    &
   'Cannot run with decomposition ',nproc_ew,'x',nproc_ns,    &
   '(',nproc_ew*nproc_ns,') processors. Maxproc is ',maxproc, &
   ' processors.'
```

- In order to flush output buffers, the routine `umprintflush` should be used for "stdout" written via `umprint` and `UM_FORT_FLUSH` for data writtent to any other fortran unit. These routines abstract flush operations providing a portable interface. These are the only method of flushing that should be used.

3.13 PrintStatus

There are four different settings of PrintStatus used in the UM, each of which is assigned a numeric value. There is a shorter form available for each one. These are defined as `PARAMETER`s and so can be tested using constructs similar to:

```
IF (PrintStatus >= PrStatus_Normal) THEN
```

For "stdout", they can also be provided as an argument to `umprint`. The current value of PrintStatus is stored in the variable `PrintStatus` in the aforementioned module, and set using the gui and/or input namelist. Note that the utility executables operate at a fixed value of `PrintStatus` and that output choices in code shared with these utilities will impact their behaviour.

The different settings are:

- `PrStatus_Min` or `PrMin` - This setting is intended to produce minimal output and should hence be only used for output which is required in every run. Users running with this setting should expect to have to rerun with a more verbose setting to diagnose any problems. Fatal error messages should fall into this category, but otherwise it should not generally be used by developers.
- `PrStatus_Normal` or `PrNorm` - The "standard" setting of PrintStatus. Messages with this setting should be important for all users in every run. Information output using this setting should summarise the situation - more detailed information should be protected by `PrStatus_Diag` instead.
- `PrStatus_Oper` or `PrOper`- Slightly more detailed than `PrStatus_Normal`, this is intended for messages which are not required for research users but are needed when running operationally.
- `PrStatus_Diag` or or `PrDiag` - The most verbose option, all messages which do not fall into one of the above categories should use this setting. Non-essential, detailed information about values of variables, status messages, etc should be included in this category. If a developer adds code to assist debugging problems, it should also be protected by `PrStatus_Diag`.

3.14 Include Files

These are blocks of declarations of global variables and constants. Files declaring physical constants may be called by any routine, but most other files will be called only by control routines. For some more information on control routine inlcudes please refer to 4.4

These are now to be phased out. Do not add another item to an include file, rather consider migrating the includes to modules or at the very least create module for your new item.

Do not create any new include files.

3.15 DrHook

DrHook is a library written by ECMWF which can produce run-time information such as:

- Per-routine profiling information based on walltime, CPU-time and MFlops.
- Tracebacks in the event of code failure. A developer can force a traceback at any point in the code with an appropriate call to the DrHook library.
- Memory usage information.

For DrHook to be effective, calls to the library are needed in each individual subroutine. DrHook must be called:

1. At the start of each routine, before any other executable code.
2. At each exit point from the routine; not only at the end, but just before any other `RETURN` statements.

When adding DrHook to a routine, the following rules should be followed:

- Routines contained in modules should include the name of the module in the call to DrHook, colon-separated. E.g. `'MODULE_NAME:ROUTINE_NAME'`.
- All names should be in capitals.

The necessary instrumentation code and the recommended method of implementing it is shown below.

```
CHARACTER(LEN=*), PARAMETER, PRIVATE :: ModuleName = 'MODULE_NAME'

CONTAINS
...

USE parkind1, ONLY: jpim, jprb
USE yomhook,  ONLY: lhook, dr_hook

...
CHARACTER(LEN=*), PARAMETER :: RoutineName = 'ROUTINE_NAME'

INTEGER(KIND=jpim), PARAMETER :: zhook_in  = 0
INTEGER(KIND=jpim), PARAMETER :: zhook_out = 1
REAL(KIND=jprb)               :: zhook_handle

IF (lhook) CALL dr_hook(ModuleName//':'//RoutineName,zhook_in,zhook_handle)

...

IF (lhook) CALL dr_hook(ModuleName//':'//RoutineName,zhook_out,zhook_handle)
```

The example subroutine shown in 2 demonstrates DrHook instrumentation.

Calls to DrHook add a very small overhead to the code, and so should normally only be added to routines that do a non-trivial amount of work. Adding DrHook calls to very small routines may represent a large increase in the workload of those routines, and furthermore if those routines are called many thousands of times during a single run of the UM then this will generate large amounts of duplicate data. The developer and reviewer may decide it is unnecessary to include DrHook calls in such routines.

Note that there is no benefit to adding DrHook calls to a module that consists only of Fortran declarations and lacks any executable code.

3.16 OpenMP

OpenMP is a very powerful technology for introducing shared memory parallelism to a code, but it does have some potential for confusion. To help minimise this, the following should be adhered to,

- Only use the OpenMP 3.1 standard. Support for OpenMP 4.0 is not yet widespread, and implementations are somewhat immature.
- Only use the `!$OMP` version of the directive and start at beginning of the line (see previous general guidance on sentinels).
- Never rely on the default behaviour for `SHARED` or `PRIVATE` variables. The use of `DEFAULT(NONE)` is preferred, with the type of all variables explicitly specified. A different `DEFAULT` may be allowed if the number of variables is very large (i.e. dozens).
- Parameters by default are shared. To make this obvious it is helpful to list parameters used in the OMP block as a Fortran comment just before the `PARALLEL` region.
- Always use explicit `!$OMP END DO` - don't rely on implicit rules.
- Unlike `SINGLE` regions, `MASTER` regions do not carry an implicit barrier at the end. Please add an `!$OMP BARRIER` directive immediately after `!$OMP END MASTER` directives. Barriers may be omitted for performance reasons if it is safe to do so.
- Calls to OpenMP functions and module use should be protected by the OpenMP sentinel. That is, the line should start with `!$` and a space. No other comment line should start with this combination.
- Always specify the scheduler to be used for DO loops, since the default is implementation specific. A common default is STATIC. This is normally fine but can cause problems within certain cases.
- Any use of a sentinel (including OpenMP) should start at the beginning of the line, e.g.

  The following correctly uses the `!$OMP` sentinel at the beginning of the line.

```
    IF (do_loop) THEN
!$OMP PARALLEL DO PRIVATE(i)
      DO i = 1, 100
      ...
      END DO
!$OMP PARALLEL DO
    END IF
```

  Whilst the following can lead to compilers not using the lines starting with `!$OMP` sentinel.

```
    IF (do_loop) THEN
      !$OMP PARALLEL DO PRIVATE(i)
      DO i = 1, 100
      ...
      END DO
      !$OMP PARALLEL DO
    END IF
```

- Careful use of the OpenMP reduction clauses is required as we want to try and preserve bit-comparison across different threads. This is not guaranteed with some `REDUCTION` clauses.

3.17 MPI

The Unified Model depends on the GCOM library for communications. GCOM has only modest functionality however so the use of MPI is permitted providing the following principles are adhered to:

- Only use MPI via GCOM's MPL interface layer. MPI libraries can be found that support only 32-bit argument or only 64-bit arguments. MPL is designed to abstract this issue away.
- Only use functionality from versions of MPI up to 3.1. These have widespread support.

3.18 Preprocessing

In the past the Cray tool `nupdate` was used for source code management and preprocessing. As it was non-portable it was replaced with a combination of `PUMSCM` for source code management and cpp for preprocessing. Since UM6.3 this has completely moved over to FCM which uses the compiler or a separate build process for preprocessing. For historical reasons large numbers of preprocessing directives (`#if`, `#include`, `#endif`) remain in the code, although their use often complicates matters. Use of preprocessor directives should only be used when its inclusion can be justified, e.g. machine dependent options. Do not use these for selecting science code section versions. As of UM8.6 all AXX_xx cpp flags have been retired. Secondly one "must" use `#if defined` rather than `#if`. If the CPP flag does not exist the the pre-processor evaluates the test to true.

In particular:

- Use run-time rather than compile time switches
- Do not replicate run-time switches with compile-time ones, so avoid

```
#if defined(OCEAN)
  IF (submodel == ocean) THEN
#endif
...
#if defined(OCEAN)
  END IF
#endif
```

- Do not add optional arguments to subroutines protected by directives, instead migrate to FORTRAN 95/2003 code and make use of OPTIONAL argument functionality.
- Put `#if` lines inside included files rather than around the `#include` itself.
- Use directive names that clearly indicate their purpose.
- When removing scientific sections, remove variables that were only needed for that section.
- Do not wrap a routine within CPP flags. Let FCM work out when it is required.
- Please refrain from using consecutive question marks (`??`) in the source code as some preprocessors can interpret them as C trigraphs.

3.19 Error reporting

Most important rule in error reporting is *never* to `CALL abort` or to use `STOP`, in a parallel environment it can cause problems. 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 warnings, which don't. Both types are passed to a reporting routine `EREPORT`. Fatal errors will cause the program to abort in `EREPORT`, replying on a traceback to report the calling tree. Warnings are reported, and control returned to the controlling routine which can then pass the error upwards for another routine to decide whether to fail or continue.

To pass errors around use the `icode` variable which is unique to each routine, where a value of 0 means no error, whilst a positive non zero value means a fatal error has occurred. Negative values are used for warnings.

The variable `icode` should be set to 0 before calling routines that return it. A routine providing a warning `icode` is likely to have already reported a warning internally, so the calling routine should just expand on the context of the problem before deciding whether to fail, ignore the warning or pass it up the chain.

When using `READ` or `OPEN` or other Fortran intrinsics which deal with IO, please use both the error status variable `IOSTAT` and the error message `IOMSG` arguments, followed by code printing the latter if the former is non-zero. Using the `check_iostat` subroutine (located in `src/control/misc`) is a convenient way to do this.

- Once the nature of the error has been determined `EREPORT` must be called with an appropriate message which should be informative to the user. `EREPORT` will reset negative `ErrorStatus` values to zero. If an IO call has failed (by returning a non-zero IOSTAT variable which you can check for), please append the message returned in the IOMSG variable to your error message before calling ereport.

- The arguments of EREPORT are:

```
CALL ereport (RoutineName, ErrorStatus,Message)

! Name of the routine
CHARACTER(LEN=*), INTENT(IN)                    :: RoutineName
! Error code
INTEGER,          INTENT(IN)                    :: ErrorStatus
! Text for output.
CHARACTER(LEN=errormessagelength), INTENT(INOUT) :: Message
```

- EREPORT writes an error message consisting of the type of error: fatal (ErrorStatus +ve) or warning (ErrorStatus -ve); the name of the calling routine; and Message. It calls UM_ABORT to abort the program for errors and returns for warnings. At present EREPORT merely writes to standard output but this could easily be extended, such as directing output to any user specified file, or standard error should the need arise.

  Please note one needs to include the line `USE errormessagelength_mod, ONLY: errormessagelength` to pickup the centrally set size of message

4 Specific standards

4.1 Runtime namelist variables, defaults, future development

The UM reads in a number of run time 'control' namelists; within READLSTA.F90. Examples are the RUN_<physics> type namelists. When new science options are required to be added to the UM the developer is expected to add the new variable/parameter to the relevant RUN_<physics> namelist and declaration in the corresponding module, updating READLSTA.F90 as required.

The use of cruntimc.h is to be avoided as this approach is being phased out in favour of suitable modules.

- Code development should use MODULES to define namelist LOGICALS, PARAMETERS and VARIABLES (and their defaults) alongwith the NAMELIST.
- It is essential that defaults are set; items within namelists are expected to fall into 3 camps:
  - variable never actually changes; it is a default for all users
    - this should be set in the code and removed from any input namelist.
  - variable rarely changes;
    - set identified default within UM code, with comment explaining choice.
    - We advise that these are not included in the namelist. A code change will be required to alter it.
  - regularly changes or is a new item and thus no default is yet suitable
    - LOGICALs usually to FALSE
    - variables set to RMDI or IMDI
    - CHARACTER strings should be set to a default string. For example,

```
aero_data_dir       = 'aero data dir is unset'
```

An example of preferred practice see RUN_Stochastic. The namelist variables are all defined within a MODULE, stochastic_physics_run_mod.F90, including default values.

4.2 Defensive input programming

When real or integer values are read into the code by a namelist, the Rose metadata should either use a values list or a range so that the Rose GUI can warn the user of invalid values. These values should also be tested in the code to ensure that the values read in are valid. As it is possible to edit Rose namelists, or ignore Rose GUI warnings, the GUI should not be relied on for checking that input values of reals and integers are valid. It may also be appropriate to check logical values if a specific combination of logicals will cause an error for example.

The routine, chk_var, is available for developers to more easily check their inputs. Checks made by chk_var should match any checks made by Rose, however checks by chk_var are made by the code and will by default, abort the run. Developers should refer to the um-training for more information on chk_var.

4.3 Optimised namelist reading procedures

As of UM9.1 the procedure to read UM namelists has been enhanced but this has implications for the code developer, requiring extra code changes when adding/removing a UM input namelist item. Tied with each namelist read is now the requirement for a 'read_nml_routine' usually found in the containing module of the namelist.

If a coder wishes to add a new variable to a namelist (xxxxxx) then the new read_nml_xxxxxx subroutine will need changing. The changes required are:

- increment the relevant type parameter by the variable size (for a real scalar increase n_real by 1)

- add a new line to the list in the my_namelist type declaration in the relevant variable type.
- add a new line to the my_nml population section in the relevant variable type
- add a new line to the namelist population section in the relevant variable type.

See the UM code for examples.

4.4 Control routine standards

Please follow section 3 with particular attention drawn to use of `EQUIVALENCE` and `COMMON` blocks.

As a requirement for dynamic allocation of primary data arrays within the model it is often 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 are currently defined by including files with `#include` 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 `#include 'ARGxxxx'` with a corresponding type declaration `#include 'TYPxxxx'`.

In addition, the number of arguments passed at the top control level should be minimized by the combining arrays into 'super arrays'.

- Please consider migrating such code to Modules.
- Please consider developing new data arrays of this type in Modules.

4.5 Unix script standards

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

- The script should be easily understood and used, and should be easy for a programmer other than the original author to modify.
- To simplify portability it should conform to the unix standard as much as possible, and exclude obsolescent and implementation-specific features when possible.
- It should be written in an efficient way.
- The structure of the script should conform to the design agreed in the project plan.

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

4.6 Python standards

Python code used in or with the UM should obey the standard Python style guide PEP 8. This means that our Python code will follow the same guidelines commonly adhered to in other Python projects, including Rose.

4.7 C standards

C code used in or with the UM should conform to the C99 standard (ISO/IEC 9899:1999: Programming languages – C (1999) by JTC 1/SC 22/WG 14).

4.7.1 Code Layout

Rules regarding whitespace, 80 column line widths, prohibition on tab use, and the use of UK English apply to C code as they would Fortran code. Comments should use the traditional `/* */` style; C++ style comments (`//`) should be avoided.

4.7.2 Copyright and Code Owner Comments

Copyright and code owner comments follow the same rules as in Fortran, except with slight modification for the differing comment delimiters in the two languages - using `/* */` instead of `!`. An example of a compliant comment header detailing copyright and code owner comments is given below.

```
/*************************************COPYRIGHT************************************/
/*            (C) Crown copyright Met Office. All rights reserved.              */
/*         For further details please refer to the file COPYRIGHT.txt           */
/*        which you should have received as part of this distribution.          */
/*************************************COPYRIGHT************************************/

/* Code Owner: Please refer to the UM file CodeOwners.txt                        */
/* This file belongs in section: C Code                                          */
```

4.7.3 Deprecated identifiers

In addition to the identifiers deprecated by the C99 standard, the following table lists identifiers which should be considered deprecated within UM code - and where appropriate, what to replace them with.

Deprecated indentifier	Replace with
`sprintf()`	`snprintf()`[1]
`strcpy()`	`strncpy()`[1]

[1] These functions take different arguments from the original deprecated functions they replace.

5 Code reviews

In order to ensure that these standards are adhered to and are having 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 performed in the order below:

1. A science/technical review is performed first to ensure that the code performs as it is intended, it complies with the standards and is well documented. Guidance for reviewers is found in the Science/Technical Review Guidance page on the UM homepage.
2. A code/system review is performed to analyse the change for its impact, ensure that it meets this coding standard and to ensure that all concerned parties are made aware of changes that are required. Guidance for reviewers is outlined in Code/System Review Guidance page on the UM homepage.

A UM Software standard summary

The rules discussed in the main text are reproduced here in summary form with pdf links to the sections.

Standard	Section
Use the naming convention for program units.	3.1
Use your header and supply the appropriately complete code header	3.2
History comments are NOT required and should be removed from routines.	3.2
Fortan code should be written in free source form	3.3
Code must occur in columns 1-80.	3.3
Never put more than one statement per line.	3.3
Use English in your code.	3.3
All Fortran keywords should be ALL CAPS while everything else is lowercase or CamelCase.	3.4
Avoid archaic Fortran features	3.4
Only use the generic names of intrinsic functions	3.4
Comments start with a single `!` at beginning of line.	3.5
Single line comments can be indented within the code, after the statement.	3.5
Do not leave a blank line after a comment line.	3.5
Do NOT use TABS within UM code.	3.5
The use of MODULEs is greatly encouraged.	3.6
Avoid additional INCLUDEs or items within INCLUDEs; migrate to MODULEs.	3.6 3.14
Use meaningful variable names	3.7
Use and declare variables and arguments in the UMDP-003 order	3.7
Use `INTENT` in declaring arguments	3.7
Use `IMPLICIT NONE`.	3.7
Use `REAL, EXTERNAL :: func1` for functions	3.7
Do not use `EXTERNAL` statements for subroutines	3.7
The use of ALLOCATABLE arrays can optmize memory use.	3.8
Indent code within `DO` or `IF` blocks by 2 characters	3.9
Terminate loops with `END DO`	3.9
Avoid comparing two reals `IF ( real1 == real2 ) THEN`	3.9
Avoid using 'magic numbers'	3.9
Avoid use of `GO TO`	3.9
Avoid numeric labels	3.9 3.12
Exception is for error trapping, jump to the label `9999 CONTINUE` statement.	3.9
Continuation line marker must be `&` at the end of the line.	3.10
Always use an `ACTION` when you `OPEN` a file.	3.11
Check for file existence with `OPEN` rather than `INQUIRE`	3.11
Always format information explcitly within WRITE, READs etc.	3.12
Ensure that output messages do not use `WRITE(6,...)`,`WRITE(*,...)`, or `PRINT*`.	3.12
Ensure that output messages are protected by an appropriate setting of `PrintStatus`.	3.13
Ensure your subroutines are instrumented for DrHook.	3.15
Only use OpenMP sentinels at the beginning of lines `!$OMP`	3.16
Be very careful when altering calculations within a OpenMP block.	3.16
If possible implement runtime logicals rather than compile time logicals.	3.18
Do not replicate (duplicate) runtime logic with cpp logic.	3.18
Do not protect optional arguments with cpp flags, use OPTIONAL args instead.	3.18
Do not use CPP flags for selecting science code, use runtime logicals	3.18
Never use `STOP` and `CALL abort`	3.19
New namelist items should begin life as category c items.	4.1

B Fortran 2003

The following table provides guidance on which Fortran 2003 features are welcome for inclusion in the UM.

This has been compiled upon review of major Fortran compilers feature support.

Feature	Acceptable	Comment
ISO TR 15581 Allocatable Enhancements	Yes	
Interoperability with C	Yes	
Access to the computing environment	Yes	
Flush	Yes	
IOMSG	Yes	
Assignment to an allocatable array	No	Includes auto-reallocation
Intrinsic Modules	Yes	eg ISO_C_BINDING
Allocatable Scalars	Yes	
Allocatable Character lengths	Yes	gnu offers partial support.
VOLATILE attribute	Yes	
Parametrized derived data types	No	Lack of compiler support
O-O coding: type extension, polymorphic entities, type bound procedures	No	Not for the current UM, but considered for the UM replacement, LFRIC-GUNGHO and MakeBC replacement CreateBC
Derived type input output	No	Lack of compiler support
Kind type parameters of integer specifiers	No	Lack of compiler support
Recursive input/output	No	
Transferring an allocation	No	Prefer to see DEALLOCATEs used for code readability.
Support for international character sets	No	

C Dealing with rounding issues.

C.1 Background

The UM perturbation sensitivity project identified coding issues that lead to excessive perturbation growth in the model. Problems identified included `IF` tests that contained comparisons between real numbers; for example `IF (qCL(i) > 0.0 )` In this test, `qCL(i)` is being used to represent one of two states;

- ”no liquid cloud”
- ”some liquid cloud”

This is fine, but it is then important to ensure that rounding issues do not lead to unintended changes of state prior to the test, such as slightly non-zero `qCL(i)` values when there is supposed to be no liquid cloud. If such problems occur at discontinuous branches in the code, the result is spurious perturbation growth.

This appendix collects together some typical examples of what can go wrong, and how to deal with them. First, though, it is worth making a quick note of some of the characteristics of floating-point arithmetic.

C.2 Floating-point identities and non-identities

In floating-point arithmetic many of the identities that hold in normal arithmetic no longer hold, basically because of the limited precision available to represent real numbers. Thus, it is often important that coders know which algebraic identities pass through to floating-point arithmetic and which don’t, and how results can be affected by the way the calculations are implemented by the compiler. For chapter and verse on floating-point arithmetic, a good reference is “David Goldberg’s article ”

The following floating-point identity always holds:

```
0.0 * x = 0.0
```

The following also hold, but only if the numbers that go into the calculations have the same precision:

```
  0.0 + x = x
  1.0 * x = x
  x / x = 1.0
  x - x = 0.0
  x - y = x + (-y)
  x + y = y + x
  x * y = y * x
2.0 * x = x + x
0.5 * x = x / 2.0
```

For example, optimisation may lead to some variables being held in cache and others in main memory, and these will generally store numbers with different levels of precision. Thus, coding based on these identities will probably work as intended in most circumstances, but may be vulnerable to higher levels of optimisation.

The following are non-identities:

```
x + (y + z) != (x + y) + z
x * (y * z) != (x * y) * z
x * (y / z) != (x * y) / z
```

These say that, unlike in normal arithmetic, the order of the calculations matters. Failure to recognise this can cause problems, as in example 1 below. (Note that putting brackets around calculations to try and impose the ”correct” order of calculation will not necessarily work; the compiler will decide for itself!)

C.3 Example 1: Non-distributive arithmetic

At UM vn7.4, the routine `LSP_DEPOSITION` contains the following calculation:

```
            ! Deposition removes some liquid water content
            ! First estimate of the liquid water removed is explicit
            dqil(i) = max (min ( dqi_dep(i)*area_mix(i)                &
     &                /(area_mix(i)+area_ice_1(i)),                    &
     &                qcl(i)*area_mix(i)/cfliq(i)) ,0.0)
...
          If (l_seq) Then
            qcl(i) = qcl(i) - dqil(i)  ! Bergeron Findeisen acts first
```

Here, `dqil` is a change to cloud liquid water `qcl`, which is limited in the calculation to `qcl*area_mix/cfliq`, where `area_mix` is the fraction of the gridbox with both liquid and ice cloud, and `cfliq` is the fraction with liquid cloud. Basically, the change to cloud liquid water is being limited by the amount of liquid cloud which overlaps with ice cloud it can deposit onto.

In the special case that all the liquid cloud coincides with ice cloud, we have `area_mix = cfliq`, implying `area_mix/cfliq = 1.0`. In this case, the limit for `dqil` should be exactly `qcl`, but is coded as `qcl*area_mix/cfliq`. In tests on the IBM, it seems that the compiler decides that the multiplication should precede the division, so the outcome of the calculation is not necessarily `qcl`. Thus, the update to `qcl` on the last line does not necessarily lead to `qcl = 0.0` when the limit is hit.

One solution to this problem is to supply `area_mix/cfliq` directly as a ratio:

```
        If (cfliq(i) /= 0.0) Then
          areamix_over_cfliq(i)=area_mix(i)/cfliq(i)
        End if
...
            ! Deposition removes some liquid water content
            ! First estimate of the liquid water removed is explicit
            dqil(i) = max (min ( dqi_dep(i)*area_mix(i)                &
     &                /(area_mix(i)+area_ice_1(i)),                    &
     &                qcl(i)*areamix_over_cfliq(i)) ,0.0)
```

This is the solution we have adopted in the large-scale precipitation code.

C.4 Example 2: Changing units when applying limits

At UM vn7.4, the routine `LSP_TIDY` contains the following calculation:

```
          ! Calculate transfer rate
          dpr(i) = temp7(i) / lfrcp ! Rate based on Tw excess

          ! Limit to the amount of snow available
          dpr(i) = min(dpr(i) , snow_agg(i)                            &
     &                       * dhi(i)*iterations*rhor(i) )
...
          ! Update values of snow and rain
          If (l_seq) Then
            snow_agg(i) = snow_agg(i) - dpr(i)*rho(i)*dhilsiterr(i)
            qrain(i)    = qrain(i)    + dpr(i)
```

where

```
dhilsiterr(i) = 1.0/(dhi(i)*iterations)
rhor(i)       = 1.0/rho(i)
```

Here, `dpr` is a conversion rate from snow into rain, and the second statement limits this rate to that required to melt all of the snow within the timestep. Thus, the intention is that if this limit is hit the final snow amount will come out to exactly 0.0. However, the outcome in this case is effectively as follows:

```
  dpr(i)      = snow_agg(i) * dhi(i)*iterations*rhor(i)
  snow_agg(i) = snow_agg(i) - dpr(i)*rho(i)*dhilsiterr(i)
            ( = snow_agg(i) &
```

```
              - snow_agg(i) &
              * dhi(i)*iterations*rhor(i)*rho(i)*1.0/(dhi(i)*iterations) )
```

In normal arithmetic, the multiplier on the final line comes out to exactly one, but this is not necessarily the case in floating-point arithmetic. Whether the expression comes out to exactly 1.0 or not will be highly sensitive to the values going into the calculation. If the result is slightly different to 1.0, the outcome is likely to be a tiny but non-zero snow amount.

The basic problem here is that the limit comes from a particular quantity, but is being applied indirectly via its rate of change. Thus when the limiting quantity is updated a change of units is required. The solution here is to apply the limit to the quantity itself, shifting the change of units to calculations involving rates:

```
          ! Calculate transfer
          dp(i) = rho(i)*dhilsiterr(i)*temp7(i) / lfrcp

          ! Limit to the amount of snow available
          dp(i) = min(dp(i), snow_agg(i))
...
          ! Update values of snow and rain
          If (l_seq) Then
            snow_agg(i) = snow_agg(i) - dp(i)
            qrain(i)    = qrain(i)    + dp(i)*dhi(i)*iterations*rhor(i)
```

C.5 Example 3: Dealing with special cases

At UM vn7.4, the routine `LS_CLD` contains the following calculation to update the total cloud fraction `CF` given the liquid and frozen cloud fractions `CFL` and `CFF`:

```
            TEMP0=OVERLAP_RANDOM
            TEMP1=0.5*(OVERLAP_MAX-OVERLAP_MIN)
            TEMP2=0.5*(OVERLAP_MAX+OVERLAP_MIN)-OVERLAP_RANDOM
            CF(I,J,K)=CFL(I,J,K)+CFF(I,J,K)                              &
     &              -(TEMP0+TEMP1*OVERLAP_ICE_LIQUID                     &
     &              +TEMP2*OVERLAP_ICE_LIQUID*OVERLAP_ICE_LIQUID)
! Check that the overlap wasnt negative
            CF(I,J,K)=MIN(CF(I,J,K),CFL(I,J,K)+CFF(I,J,K))
```

During testing, it was observed that `CF` was often coming out to `0.9999999999999...`; i.e., almost but not quite 1.0, and that whether this occured was highly sensitive to the input data. This sensitivity was then being passed down to branches testing on, for example, whether `CF` was equal to `CFF`.

If the above calculations are followed through algebraically, it can be shown that if `CFL+CFF >= 1`, then `CF` must be exactly one. In the floating-point case, however, this no longer follows, so we often get cases where there is a slight deviation from unity. The simplest solution in this example is to deal with the special case separately:

```
            TEMP0=OVERLAP_RANDOM
            TEMP1=0.5*(OVERLAP_MAX-OVERLAP_MIN)
            TEMP2=0.5*(OVERLAP_MAX+OVERLAP_MIN)-OVERLAP_RANDOM
! CFF + CFL >= 1 implies CF = 1
            IF (CFL(I,J,K)+CFF(I,J,K) >= 1.0) THEN
              CF(I,J,K) = 1.0
            ELSE
              CF(I,J,K)=CFL(I,J,K)+CFF(I,J,K)                            &
     &                -(TEMP0+TEMP1*OVERLAP_ICE_LIQUID                     &
     &                +TEMP2*OVERLAP_ICE_LIQUID*OVERLAP_ICE_LIQUID)
! Check that the overlap wasnt negative
              CF(I,J,K)=MIN(CF(I,J,K),CFL(I,J,K)+CFF(I,J,K))
            END IF
```