Fortran coding standard for FCM

Latest content update: 14 January 2010.

Questions regarding this document or permissions to quote from it should be directed to:

IPR Manager
Met Office
FitzRoy Road
Exeter, Devon
EX1 3PB
United Kingdom

---

1. Introduction

Fortran is the standard programming language at the Met Office for developing scientific and research applications, in particular, for programs running on the supercomputers. This document describes some guidelines that should be followed by developers when writing Fortran code, especially for code in systems hosted by FCM.

2. Programming Fortran for the FCM build system

2.1 General

To get the most out of the FCM build system, you should follow these guidelines when you develop your code:

1. Each source file should contain one and no more than one top level program unit, (such as a `PROGRAM`, a standalone `SUBROUTINE`/`FUNCTION` or a `MODULE`). All top level standalone program units in a source tree should be uniquely named. (*Top level* means a standalone program unit that is compilable independently, i.e. this rule does not restrict the naming and placements of sub-programs in a `CONTAINS` section.) FCM may fail to set up the dependency tree of your source correctly if you do not follow this rule.

   A clash of program unit names happens most often when you have multiple versions of the same program unit in the same source tree. You should design your code to avoid this situation. If it is not possible to do so, you may have to use a pre-processor to ensure that there is only one copy of each program unit in the source tree. Another situation where clashes of program unit names may occur is when you are developing code that is shared between several projects. In this case, you may want to agree with the other projects a naming convention to define a unique namespace for program units in each project. (E.g. some projects at the Met Office have adopted a naming convention such that all shared program units in a project are named with a unique prefix.)

2. All code should be written using the free source form. (At Fortran 95, the free source form can have a maximum of 132 characters per line, and up to 39 continuations in a Fortran statement.) The fixed source form is obsolete, and is not supported by the interface file generators used by FCM.

3. An interface should be provided when calling a SUBROUTINE or a FUNCTION. Not only is this considered good practice, it also allows FCM to determine the dependency relationship of your source files. An interface can be provided in these ways:
   Internal sub-program

   Place sub-programs in the CONTAINS section of a standalone program unit. There are two advantages for this approach. Firstly, the sub-programs will get an automatic interface when the container program unit is compiled. Secondly, it should be easier for the compiler to provide optimisation when the sub-programs are internal to the caller. The disadvantage of this approach is that the sub-programs are local to the caller, and so they cannot be called by other program units. Therefore, this approach is only suitable for small sub-programs local to a particular program unit.

   Note: One way to share a sub-program unit between several top level program units is to make use of the Fortran INCLUDE statement. You can write the sub-program unit in a separate file and place it in the CONTAINS section of different program units using INCLUDE statements. The disadvantage of this approach is that when the program is compiled, a copy of the sub-program unit will be embedded within each of the top level program units. This may lead to an increase in size of the executable, and so this approach is still only suitable for small sub-programs local to a small number of program units.

   For example, if we have a SUBROUTINE in sub_prog.inc:

```
SUBROUTINE sub_prog(some, arg)
! Some declarations ...
! Some executable statements ...
END SUBROUTINE sub_prog
```

   It can be placed in a MODULE in bar.f90:

```
SUBROUTINE bar(more, arg)
! Some declarations ...
! Some executable statements ...
CALL sub_prog(some, arg)
! More executable statements ...
CONTAINS
  INCLUDE 'sub_prog.inc'
END SUBROUTINE bar
```

   Module procedures

   Place sub-programs in the CONTAINS section of a MODULE. Again, the sub-programs will have automatic interfaces when the MODULE is compiled. If you give the sub-programs the PUBLIC attribute (which is the default), you will be able to call them from anywhere using the current MODULE. You will also gain all the advantages offered by a MODULE. (E.g. a MODULE will allow you to design your code in a more object-oriented manner.) However, MODULE dependency can have an impact on the efficiency of incremental compilations. For example, if you modify items that are local to the MODULE, it is very difficult for the build system to detect that your change does not affect program units using the MODULE, so the build system will end up compiling the MODULE and all the program units that use it.

   For example, if we have a MODULE in my_mod.f90:

```
MODULE my_mod
! Some module declarations
CONTAINS
  SUBROUTINE sub_prog(some, arg)
  ! Some declarations ...
  ! Some executable statements ...
  END SUBROUTINE sub_prog
END MODULE my_mod
```

It can be imported to another program as such:

```
SUBROUTINE foo(some, arg)
USE my_mod, ONLY: sub_prog
! Some declarations ...
! Some executable statements ...
CALL sub_prog(some, arg)
! More executable statements ...
END SUBROUTINE foo
```

Interface files

For each source file containing a standalone `SUBROUTINE` or `FUNCTION`, FCM generates a file containing the interface of the `SUBROUTINE` or `FUNCTION`. By default, the generated file is named after the original source file, but with the file extension replaced by `*.interface`. In the specification section of the caller routine, you will then be able to declare the interface using a Fortran `INCLUDE` statement to include the interface file. This type of `INCLUDE` statement is detected automatically by FCM, which will use it to set up the dependency tree.

The advantage of using an interface file is that the caller is now dependent on the interface file, rather than the `SUBROUTINE` or `FUNCTION` itself. If you change the `SUBROUTINE` or `FUNCTION` without modifying its interface, the build system will not re-compile the caller in incremental build, (but it will be intelligent enough to re-link the executable with the updated object).

Note: By default, an interface file is named after the original source file. Bearing this in mind, it is worth noting that file names in a Unix/Linux system are case-sensitive, and so the interface file name declared by your `INCLUDE` statement is also case sensitive. If you use an incorrect case in the `INCLUDE` statement, the dependency tree will be set up incorrectly and the compilation will fail. Another problem is that if you do not name your file after the program unit, the dependency tree will be wrong. To avoid this problem, it is recommended that all source files are named in lower case after the program units they contain. (Alternatively, you can use the `TOOL::INTERFACE` option in the FCM build configuration file to allow you to alter the default behaviour so that the interface file is named after the program unit in lowercase. We may alter FCM in the future so that this will become the default. In the mean time, it is highly recommended that you use this option and design your new code accordingly.)

For example, if we have a `SUBROUTINE` in `sub_prog.f90`:

```
SUBROUTINE sub_prog(some, arg)
! Some declarations ...
! Some executable statements ...
END SUBROUTINE sub_prog
```

It can be imported to another program as such:

```
SUBROUTINE egg(some, arg)
! Some declarations ...
INCLUDE 'sub_prog.interface'
! More declarations ...
! Some executable statements ...
CALL sub_prog(some, arg)
! More executable statements ...
END SUBROUTINE egg
```

Interfaces in a module

There is also a half-way house approach between the second and the third options. You can have a dedicated `MODULE` where a large number of `INCLUDE` interface file statements are placed. Other program units get their interfaces by importing from this `MODULE`. A major disadvantage of this approach is that the sub-programs with their interfaces declared within this `MODULE` will not be able to call any other sub-programs declared within the same `MODULE`, as it will run into a cyclic dependency problem. Another disadvantage is that if an interface changes, the `MODULE` and all program units depending on the `MODULE` will have to be re-compiled, even though the change may be unrelated to some or all of these program units. For these reasons, this approach is only good if you have a bundle of sub-programs that have relatively stable interfaces and are very much independent of one another.

Note: a similar approach can be useful when you have a library of legacy or external code. In this situation, you will simply declare the interfaces for all the library sub-programs in the `MODULE`. Any programs that call sub-programs within the library can then import their interfaces by using the `MODULE`.

For example, if we have a `MODULE my_i_mod`:

```
MODULE my_i_mod
! Some declarations
INCLUDE 'sub_prog.interface'
! More declarations
END MODULE my_i_mod
```

Its `PUBLIC` procedures can be imported as such:

```
SUBROUTINE ham(some, arguments)
USE my_i_mod, ONLY: sub_prog
! Some declarations ...
! Some executable statements ...
CALL sub_prog(some, arguments)
! More executable statements ...
END SUBROUTINE ham
```

FCM also supports the use of a `! DEPENDS ON` directive for users to specify a dependency from within a source file. This feature is documented in the Further dependency features sub-section of the FCM user guide. However, it is worth noting that this method is only included in FCM to support legacy code. It is not a feature recommended for new code, and its use should be gradually phased out from existing code.

4. Arguments and local variables should be declared in different statements. It makes your declaration clearer, and it is friendlier to the interface file generator.
Common practice

```
SUBROUTINE foo(a, b, c)
INTEGER :: a, b, c, i, j, k
! ...
END SUBROUTINE foo
```

Better approach

```
SUBROUTINE foo(a, b, c)
INTEGER :: a, b, c
INTEGER :: i, j, k
! ...
END SUBROUTINE foo
```

5. Use the `ONLY` clause in a `USE <module>` statement to declare all imported symbols (i.e. parameters, variables, functions, subroutines, etc). This makes it easier to locate the source of each symbol, and avoids unintentional access to other `PUBLIC` symbols within the `MODULE`. It is also friendlier to the compiler and the interface file generator, as they will not have to import modules and symbols that are unnecessary.
6. In its default settings, FCM recognises the following file extensions as Fortran free format source files:
   - `*.f90, *.f95`: regular Fortran free format source files
   - `*.F90, *.F95`: Fortran free format source files that require pre-processing
   - `*.inc`: Include files that can be added to a regular Fortran free format source file with a Fortran `INCLUDE` statement

2.2 Use of C pre-processor with Fortran

We do not recommend the use of C pre-processor with Fortran. However, it is acknowledged that there are some situations when it is necessary to pre-process Fortran code. FCM supports pre-processing in two ways. Pre-processing can be left to the compiler or it can be done in a separate early stage of the build process. A separate pre-process stage can be useful if pre-processing changes any of the following in a program unit:

- its name
- its calling interface
- its dependencies

However, using a separate pre-process stage is not the best way of working, as it adds an overhead to the build process. If your code requires pre-processing, you should try to design it to avoid changes in the above.

In practice, the only reasonable use of C pre-processor with Fortran is for code selection. For example, pre-processing is useful for isolating machine specific libraries or instructions, where it may be appropriate to use inline alternatives for small sections of code. Another example is when multiple versions of the same procedure exist in the source tree and you need to use the pre-processor to select the correct version for your build.

Avoid using the C pre-processor for code inclusion, as you should be able to do the same via the Fortran `INCLUDE` statement. You should also avoid embedding pre-processor macros within the continuations of a Fortran statement, as it can make your code very confusing.

3. Programming Fortran in general

The guidelines in this section are recommended practices for programming Fortran in general. These are guidelines you should try to adhere to when you are developing new code. If you are modifying existing code, you should adhere to its existing standard and style where possible. If you want to change its standard and style, you should seek prior agreements with the owner and the usual developers of the code. Where possible, you should try to maintain the same layout and style within a source file, and preferably, within all the source code in a particular project.

When reading these guidelines, it is assumed that you already have a good understanding of modern Fortran terminology. It is understood that these guidelines may not cover every aspect of your work. In such cases, you will be a winner if you use a bit of common sense, and always bearing in mind that some other people may have to maintain the code in the future.

Always test your code before releasing it. Do not ignore compiler warnings, as they may point you to potential problems.

3.1 Layout and formatting

The following is a list of recommended practices for layout and formatting when you write code in Fortran.

- Indent blocks with space characters. The use of 2 spaces per indentation level is wide spread at the Met Office. The use of 4 spaces per indentation level is most popular in the open source community because it is easier for the human eyes. If you are modifying existing scripts, you should stick to their existing styles. Otherwise, the use of 2 and 4 spaces per indentation level are both acceptable, as long as it is consistent within a source file. Where possible, comments should be indented with the code within a block.
- Use space and blank lines where appropriate to format your code to improve readability. (Use genuine spaces but avoid using tabs, as the **`tab`** character is not in the Fortran character set.) In the following example, the code on the right hand side is preferred:
  Common practice

```
DO i=1,n
  a(i)%c=10*i/n
  b(i)%d=20+i
ENDDO
IF(this==that)THEN
  distance=0
  time=0
ENDIF
```

  Better approach

```
DO i = 1, n
  a(i)%c = 10 * i / n
  b(i)%d = 20 + i
END DO

IF (this == that) THEN
  distance = 0
  time     = 0
END IF
```

- Try to confine your line width to 80 characters, so that your code can be printed easily on A4 paper.
- Line up your statements, where appropriate, to improve readability. For example:
  Common practice

```
REAL, INTENT(OUT) :: my_out(:)
REAL, INTENT(INOUT) :: my_inout(:)
REAL, INTENT(IN) :: my_in(:)
! ...
CHARACTER(LEN=256) :: my_char
! ...
my_char = 'This is a very very very very very very very very very very ' // &
   'very very very very very very very very very very very very ' // &
   'long character assignment'
```

Better approach

```
REAL, INTENT(  OUT) ::   my_out(:)
REAL, INTENT(INOUT) :: my_inout(:)
REAL, INTENT(IN   ) ::    my_in(:)
! ...
CHARACTER(LEN=256) :: my_char
! ...
my_char                                                                    &
    =  'This is a very very very very very very very very very very '  &
    // 'very very very very very very very very very very very very '  &
    // 'long character assignment'
```

- 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.
- Avoid putting multiple statements on the same line. It is not good for readability.

3.2 Style

The following is a list of recommended styles when you write code in Fortran.

- New code should be written using Fortran 95 syntax. Avoid unportable vendor/compiler extensions. Avoid Fortran 2003 features for the moment, as they will not become widely available in the near future. (Having said that, there is no harm in designing your code with the future in mind. For example, if there is a feature that is not in Fortran 95 and you know that it is in Fortran 2003, you may want to write your Fortran 95 code to make it easier for the future upgrade.)
- 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, bearing in mind that your code may be read by people who are not proficient English speakers.
- When naming your variables and program units, always bear in mind that Fortran is a case-insensitive language. (E.g. *EditOrExit* is the same as *EditorExit*.)
- Use only characters in the Fortran character set. In particular, accent characters and tabs are not allowed in code, although they are usually OK in comments. If your editor inserts tabs automatically, you should configure it to switch off the functionality when you are editing Fortran source files.
- Although Fortran has no reserved keywords, you should avoid naming your program units and variables with names that match an intrinsic `FUNCTION` or `SUBROUTINE`. Similarly, you should avoid naming your program units and variables with names that match a *keyword* in a Fortran statement.
- Be generous with comments, but avoid repeating the Fortran logic in words. State the reason for doing something instead.
- To improve readability, write your program in mainly lower case characters. Writing a program in mainly lower case also means that you will not have to use the **`Shift/Caps Lock`** keys often, hence, improving your code's accessibility. There is a lot of debate on using upper/lower cases in a case insensitive language such as Fortran. There is no right or wrong, but people have adopted the different approaches over time, each has its own merit. If you are starting a new project, you should choose a suitable option and stick to it. Otherwise, you should stick with the style in the existing code. Some options are listed here:
    - The ALL CAPS Fortran keywords approach, like most of the examples in this document, where all Fortran keywords and intrinsic procedures are written in ALL CAPS. This approach has the advantage that Fortran keywords stand out, but it does increase how often the Shift/Caps Lock key is used. Programmers who are used to some other programming languages may also find it difficult to read a program with a lot of upper case characters.

- The Title Case Fortran keywords approach, where all Fortran keywords are written with an initial capital case letter.
- The sentence case approach, where only the initial character in a Fortran statements is written in capital case letter, like a normal sentence.
- The all lower case approach, where all Fortran keywords are written in lower case letters.
- Some people have also proposed a variable naming convention where local variables start with an initial lower case letter, private module level variables with an initial capital case letter and public module variables written in all caps. However, this approach has been seen by many as too restrictive, and so its use has not been widely spread.

- Use the new and clearer syntax for `LOGICAL` comparisons, i.e.:
  - `==` instead of `.EQ.`
  - `/=` instead of `.NE.`
  - `>` instead of `.GT.`
  - `<` instead of `.LT.`
  - `>=` instead of `.GE.`
  - `<=` instead of `.LE.`
- 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
```

- Positive logic is usually easier to understand. When you have 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 size. (However, it may be more appropriate to use negative logic if the negative block is significantly bigger than the positive block.) For example:
  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
```

- 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
```

- If you have a large or complex code block embedding other code blocks, you may consider naming some or all of them to improve readability.
- If you have a large or complex interface block or if you have one or more sub-program units in the `CONTAINS` section, you can improve readability by using the full version of the `END` statement (i.e. `END SUBROUTINE` <name> or `END FUNCTION` <name> instead of just `END`) at the end of each sub-program unit. For readability in general, the full version of the `END` statement is recommended over the simple `END`.
- Where possible, consider using `CYCLE`, `EXIT` or a `WHERE`-construct to simplify complicated `DO`-loops.
- 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.
- Where reasonable and sensible to do so, you should try to match the names of dummy and actual arguments to a `SUBROUTINE`/`FUNCTION`.
- In an array assignment, it is recommended that you use array notations to improve readability. E.g.:
  Common practice

```
INTEGER :: array1(10, 20), array2(10, 20)
INTEGER :: scalar
! ...
array1 = 1
array2 = array1 * scalar
```

  Better approach

```
INTEGER :: array1(10, 20), array2(10, 20)
INTEGER :: scalar
! ...
array1(:, :) = 1
array2(:, :) = array1(:, :) * scalar
```

- Where appropriate, use parentheses to improve readability. E.g.: `a = (b * i) + (c / n)` is easier to read than `a = b * i + c / n`.

3.3 Fortran features

The following is a list of Fortran features that you should use or avoid.

- Use `IMPLICIT NONE` in all program units. It means that you have declare all your variables explicitly. This helps to reduce bugs in your program that will otherwise be difficult to track.
- Design your derived data types carefully and use them to group related variables. Appropriate use of derived data types will allow you to design modules and procedures with simpler and cleaner interfaces.
- 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.
- When you are passing an array argument to a `SUBROUTINE`/`FUNCTION`, and the `SUBROUTINE`/`FUNCTION` does not change the `SIZE`/`DIMENSION` of the array, you should pass it as an assumed shape array. Memory management of such an array is automatically handled by the `SUBROUTINE`/`FUNCTION`, and you do not have to worry about having to

ALLOCATE or DEALLOCATE your array. It also helps the compiler to optimise the code.

- Use an array POINTER when you are passing an array argument to a SUBROUTINE, and the SUBROUTINE has to alter the SIZE/DIMENSION of the array. You should also use an array POINTER when you need a dynamic array in a component of a derived data type. (Note: Fortran 2003 allows passing ALLOCATABLE arrays as arguments as well as using ALLOCATABLE arrays as components of a derived data type. Therefore, the need for using an array POINTER should be reduced once Fortran 2003 becomes more widely accepted.)
- 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 if its memory allocation and deallocation are 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(). 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().
- Avoid the DIMENSION attribute or statement. Declare the DIMENSION with the declared variables. E.g.:
  Common practice

```
INTEGER, DIMENSION(10) :: array1
INTEGER                :: array2
DIMENSION              :: array2(20)
```

  Better approach

```
INTEGER :: array1(10), array2(20)
```

- Avoid COMMON blocks and BLOCK DATA program units. Use PUBLIC MODULE variables.
- Avoid the EQUIVALENCE statament. Use a POINTER or a derived data type, and the TRANSFER intrinsic function to convert between types.
- Avoid the PAUSE statement, as your program will hang in a batch environment. If you need to halt your program for interactive use, consider using a READ* statement instead.
- Avoid the ENTRY statement. Use a MODULE or internal SUBROUTINE.
- Avoid the GOTO statement. The only commonly acceptable usage of GOTO is for error trapping. In such case, the jump should be to a commented 9999 CONTINUE statement near the end of the program unit. Typically, you will only find error handlers beyond the 9999 CONTINUE statement.
- Avoid assigned GOTO, computed GOTO, arithmetic IF, etc. Use the appropriate modern constructs such as IF, WHERE, SELECT CASE, etc..
- Avoid numbered statement labels. DO ... label CONTINUE constructs should be replaced by DO ... END DO constructs. FORMAT statements should be replaced by format strings. (Tip: a format string can be a CHARACTER variable.)
- Avoid the FORALL statement/construct. Despite what it is supposed to do, FORALL is often difficult for compilers to optimise. (See, for example, Implementing the Standards including Fortran 2003 by NAG.) Stick to the equivalent DO construct, WHERE statement/construct or array assignments unless there are actual performance benefits using FORALL in your code.
- A FUNCTION should be PURE, i.e. it should have no side effects (e.g. altering an argument or module variable, or performing I/O). If you need to perform a task with side effects, you should use a SUBROUTINE instead.
- Avoid using a statement FUNCTION. Use an internal FUNCTION instead.
- Avoid RECURSIVE procedures if possible. RECURSIVE procedures are usually difficult to understand, and are always difficult to optimise in a supercomputer environment.

- Avoid using the specific names of intrinsic procedures. Use the generic names of intrinsic procedures where possible.

4. Program templates

The following is a basic template for a `FUNCTION`:

```
[<type>] FUNCTION <name>(<args, ...>) [RESULT(<result>)]

!-------------------------------------------------------------------------------
! (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.
!-------------------------------------------------------------------------------
! DESCRIPTION
!   <Explain what the function does.>
!
!-------------------------------------------------------------------------------
USE <module>, ONLY : <symbols, ...>

IMPLICIT NONE

! Function arguments:
<arguments with INTENT(IN) ...>

! Function result:
<declare the type returned by the function>

! Local declarations:
<parameters, derived data types, variables, ...>

! INTERFACE blocks
<INCLUDE interface files...>
<other interface blocks...>

<other specification statements ...>
!-------------------------------------------------------------------------------
<executable statements ...>
!-------------------------------------------------------------------------------
CONTAINS
  <sub-programs>
END FUNCTION <name>
```

The basic templates for other types of program units are similar to that of a `FUNCTION`, with the following exceptions:

- A `PROGRAM` does not have arguments, so the arguments list in the header and the *Arguments* section in the declaration section should be removed. All declarations are local to a `PROGRAM`, so the *Local Declarations* section should be replaced by a simple *Declarations* section.
- A `SUBROUTINE` may have `INTENT(OUT)` and `INTENT(INOUT)` arguments.
- A `MODULE` does not have arguments, so the arguments list in the header and the *Arguments* section in the declaration section should be removed. Where appropriate, the *Local Declarations* section should be replaced by a *PUBLIC declarations* section and a *PRIVATE declarations* section.

When you are distributing your code, you should include a `COPYRIGHT.txt` file at a top level directory in your source tree. The file should contain the detailed copyright information:

- the copyright year, ranging from the year the code is first distributed to the year the code is last distributed
- the copyright statement

- the owner of the code and his/her address

For example:

```
!-------------------------------------------------------------------------------!
!                                                                               !
! (C) Crown copyright 2005-6 Met Office. All rights reserved.                   !
!                                                                               !
! Use, duplication or disclosure of this code is subject to the restrictions   !
! as set forth in the contract. If no contract has been raised with this copy  !
! of the code, the use, duplication or disclosure of it is strictly            !
! prohibited. Permission to do so must first be obtained in writing from the   !
! Head of Numerical Modelling at the following address:                        !
!                                                                               !
! Met Office, FitzRoy Road, Exeter, Devon, EX1 3PB, United Kingdom             !
!                                                                               !
!-------------------------------------------------------------------------------!
```