CCFE-R(15) 34

CCFE[1]

Fortran 95 Programming Style

Wayne Arter[2], Nick Brealey[3], James Eastwood and Guy Morgan[4]

March 25, 2020

[1]UK Atomic Energy Authority, Culham Science Centre, Abingdon, Oxfordshire OX14 3DB
[2]Telephone: +44 1235 466433 E-mail: wayne.arter@ccfe.ac.uk
[3]Cobham Technical Services, Network House, Langford Locks, Kidlington, OX5 1LH
[4]Culham Electromagnetics Ltd, Culham Science Centre, Abingdon, OX14 3DB

Contents

Abstract

This document summarises the Fortran 95 programming style to be used for object-oriented Fortran codes.

Chapter 1

Introduction

The consistent use of a set of programming conventions can make software easier to understand, develop and maintain. This leads to increased reliability of the software.

This work instruction (based on the unpublished report [1]) describes the programming conventions for the use of Fortran 95 in the QPROG project. The programming conventions used have an impact on the following aspects of the software which should be considered when making choices:

1. clarity,
2. modularity,
3. defensive programming,
4. flexibility,
5. portability,
6. performance,
7. robustness and
8. traceability.

By clarity we mean how easy the code is to read and understand. Code layout, naming conventions and comments are low-level factors which are particularly important here but the architecture of the software and external documentation are also important.

Modularity is related to clarity. It means breaking the code up into manageable pieces so that a developer can work on one piece of the code without having a complete understanding of every other part of the code.

By defensive programming we mean how robust the code is with respect to programming errors as distinct from how the code handles errors due to incorrect user input or system errors, which we regard as being related to robustness. We consider the code to be programmed defensively if programming errors will stop the code compiling. Factors which contribute to a defensive programming style include using explicit interfaces to subroutines and assumed shape arrays so that mismatched argument lists are

detected, using keywords in subroutine calls, using `implicit none` and `private` so that mistyped variable names are more likely to be detected, using derived types etc.

By flexibility we mean how easy it is to change the code. Leaving our options open is part of this. Flexibility can be improved by using derived types where we can change the implementation later, using `private` so that we know that changing one part of the code won't have an unexpected effect on another part of the code.

Portability means that the software should not be tied to particular computers or compilers. For QPROG, we aim to achieve portability by writing software which conforms to the Fortran 95 standard [2]. There are however minor portability and standards-compliance issues with obtaining command line arguments. (Features from Fortran 2003 address these issues.)

Performance is concerned with how fast the code runs, how large a problem can be run on a particular system and how accurate the results are. The choice of algorithm and input parameters are the primary factors here but choices such as using pointers or indices, or using derived types or multiple arrays, may have an effect on performance.

Robustness is concerned with how the code behaves in response to user errors or system errors such as not being able to allocate more memory or running out of space on a disk. Checking the status of `allocate`, `read` and `write` statements and avoiding large automatic arrays are relevant here.

Traceability is concerned with knowing which versions of the source files were used to build the software, which compiler was used, what compiler options were used, whether the validation test deck been run with a particular executable etc.

Chapter 2

Style Recommendations

This section describes recommendations for programming style. Section 3 describes recommendations which are more related to architecture, although the distinction may be subjective.

2.1 Free source format

Use the free source form which was introduced in Fortran 90.

2.2 One module per file

Only put one module in each file. We expect the program to consist entirely of Fortran modules and one program unit. Interface modules shall be used to provide interfaces to heritage Fortran 77 program units.

2.3 File names

The file name should be the same as the name of the module or program unit, in lower case, with the extension `.f90`.

If it is necessary to have different versions of a file add a suffix to the root part of the name e.g. if we have a standard `types` module in file `types.f90` and we want an alternative version which uses 32 bit numbers instead of 64 bit numbers we could put the alternative version in the file `types.r4.f90`.

The `.f90` *means the free source format which was introduced in Fortran 90. Some people have suggested that we use* `.f95` *to indicate that the source is Fortran 95. The problem with this is that there are Fortran 95 compilers which do not accept the* `.f95` *extension. We use lower case for file names because they are easier to read than uppercase.*

2.4 Lower-case tokens

Use lower case for all Fortran tokens (keywords, names, constants, operators, edit descriptors etc) except for character literal constants. (Comments should be in mixed case.)

2.5 Naming conventions

We advocate using the module name as the prefix for public variables, public derived types and public routines in a module e.g. public variables and routines in module `group` should have names like `group_number` or `group_first()`.

It should be remembered that dummy argument names can be used as keyword names in Fortran 95 routines with explicit interfaces. Dummy argument names should be simple names which make sense when used as keyword arguments e.g.

```
n_tree = tree_size( root_node=node(1) )
```

Local variables and types should not have the module prefix. They should have names like `j` or `max_iterations` (assuming that there is no `max` module).

The names for private module variables and routines should be chosen by the programmer to aid readability. They should not use the module name as a prefix, so that they are distinguished from their public counterparts.

Derived types should have the suffix `_t` e.g. `tree_node_t`. This allows the use of dummy argument names without the `_t` which is convenient for keyword arguments e.g. `call tree_clear(tree_node=x)`.

2.6 Object-oriented design pattern

QPROG will use an object-oriented design pattern to allow multiple instances of various classes of object.

In this design pattern the data which represent an instance of an object are stored in a variable of a derived type which is created outside the module. The 'member functions' which operate on the object are module routines which take the data derived type as an argument. The derived type is defined in the module.

For this design pattern we need conventions for the module name, the type which contains the data for an object, the constructor for the object, the destructor for the object and the dummy argument name and position for the variable containing the object's data.

The naming convention for QPROG is

1. The name with a `_m` suffix is the name of the module (and the base name of the file containing the module).

2. The name with the suffix `_t` is the derived type containing the data for the instance of an object.

3. `self` is the name of the dummy argument of the module data type. The `self` argument should be the first argument.

4. The constructor for an object should be a subroutine with the module name with the suffix `_init`.

5. The destructor for an object should be a subroutine with the module name with the suffix `_delete`.

6. When a procedure needs to have an object of type `xxx_t` passed to it a dummy argument name of `xxx` should be used if the `self` dummy argument name is inappropriate.

Appendix A contains a template object-oriented design pattern `oo_m.f90` for use in QPROG.

2.7 Avoid using double or trailing underscores

Don't use names like `my__variable` or `my_variable_`. Double underscores are hard to distinguish from single underscores in some fonts.

2.8 Use implicit none and private

Every module should have `implicit none` and `private` statements. Program units should have `implicit none`. The `private` attribute can be used in user-defined types to hide the components of a type.

2.9 One statement per line

Generally, do not put more than one statement per line (don't use the `;` statement separator). However, this practice may be helpful if the statements are simple equalities and involve closely related variables, e.g. assignments to different components of a vector or derived type.

2.10 Indenting

The source code should use indenting. The indenting style is as follows. Statements that logically subdivide the source (such as `module`, `subroutine`, `end`

`function`) begin in column 1. All other statements are indented at least 2 characters, except statements with numeric labels.

The extra indentation used in blocks such as `if`, `do`, should be 2 more characters per level of indentation. There should be no extra indentation applied to continuation lines, which should be denoted by a trailing ampersand `&`. Optionally `&` may appear in column 2. Comments should be indented so that the initial `!` lines up with what they are commenting on.

2.11 Continuation lines

Use an ampersand, `&`, at the end of the line to be continued and in column 2 of the continuation line if desired.

2.12 Labelling

- Numeric labels should not be used except in the case of the number 9999 or 999 and then only for an error/emergency exit case.
- The `goto` statement can be used to break out of a loop but `exit` and `cycle` are preferred where possible.
- Longer `do` loops should be labelled with a meaningful text identifier to aid the understanding of programmers who are not familiar with the code.
- `do` loops with `exit` statements are preferred to the `do while` construct which is constrained to test the logical clause at the beginning of the loop whereas an `exit` statement can occur at any point in a loop.

2.13 Format statements

In order of preference:

1. From a character variable,
2. embedded in the `write` statement, e.g. `write(luout,'(i2,a)')...`,
3. Numbered `format` statements.

2.14 Keywords and spaces

The preferred style is to use spaces in `else if`, `end do`, `end subroutine` etc.

`goto` is preferred to `go to`.

2.15 Declarations

All parts of the declaration of a variable should be in a single statement rather than having multiple statements with the declaration scattered:

```
real(kr2), dimension(:,:), target, allocatable, save, public :: x
```

is preferred over:

```
real(kr2) x
...
dimension x(:,:)
...
allocatable x
...
target    x
...
public    x
...
save      x
```

N.B. the form above is preferred over:

```
real(kr2), target, allocatable, save, public :: x(:,:)
```

because it allows the variable to be initialised in the declaration:

```
real(kr2), public, dimension(3) :: x0 = 0.0_kr2
```

Similarly, the attributes `intent(...)`, `optional`, `parameter`, `pointer` and `private` should be used in type declaration statements in preference to the corresponding separate declarations.

The new form of character declaration should be used:

```
character(4), parameter :: extension = '.dat'
```

or

```
character(len=4), parameter :: extension = '.dat'
```

This is more regular than the older way of specifying the length (which can only be used with characters with default kind):

```
character*4, parameter :: extension = '.dat'
```

2.16 Comments and documentation

In QPROG, documentation is part of the code. We propose a particular commenting style which can be processed by doxygen [3] to produce html and/or pdflatex documentation automatically. doxygen operation is by default controlled by a file named

"doxyfile", supplied with the source. Note that doxygen 1.8.5 or later is needed for best results.

The following features of `doxyfile` are noted, viz. the setting of the tag
`PROJECT_NAME =` QPROG
The html aliases

```
ALIASES = Bold{1}="<b>\1</b>"
ALIASES += Emph{1}="<em>\1</em>"
ALIASES += Tt{1}="<tt>\1</tt>"
```

doxygen uses `!>` to indicate that the comment refers to the following Fortran source, `!<` to the preceding Fortran source and `!!` (double exclamation marks) to extend a comment over a line. The comments applying to a module, subroutine, function or variable should be placed immediately before the item they are describing, ie. use `!>`. Comments can also be placed at the end of a line (using `<!`) but because of space limitations this is usually only suitable for declarations of variables which don't require much description. With the `ALIASES` tag set as above text enclosed by braces following `\Emph`, `\Bold` and `\Tt` is set respectively to the emphasised, strong and monotype fonts. Indented text starting with '-' is set as itemised lists in the output from doxygen. Alternatively, html markup can be included in the comments in the code, but this makes it less readable in the code listing. Mathematical quantities can be set in LaTex by enclosing the LaTeX formulae by the strings `\f$` and `\f$`.

Appendix A shows listings of sample module files:

1. a template file `template_m.f90` showing the standard usage of the doxygen formatted comments.
2. an object-oriented pattern file `oo_m.f90` showing doxygen comments in the OO factory pattern to be used in QPROG and
3. a second, bigger file `bigtobj_m.f90` showing the pattern to be followed for objects with associated file handling.
4. a object definition file `bigobj_h.f90` to go with objects with associated file handling, or which have many different uses.
5. a program control object-oriented methods module `qcontrol_m.f90` for use with program `qprog`. This reads in data from separate namelists which defines files for the program (as distinct from its modules), a set of parameters for the program and a set of output control options for the program.
6. a program control object definition module `qcontrol_h.f90` for use with module `qcontrol_m`.
7. Main program `qprog.f90` for program `qprog`.

Appendix B, actually now a separate document, shows the doxygen output for these sample modules.

Documentation of the Fortran 95 program files shall take the following form:

1. Each file shall have the rubric as specified in Section 2.17 and shown in the examples in Appendix A.

2. Every module should have after the rubric comments giving the summary detailed design documentation associated with it. This falls under the following five headings, the last two of which are optional if doxygen has been used to produce html documentation:

   Type The type should be defined by stating its logical and physical characteristics.

   The *logical characteristics* should be stated by saying which of the following categories the module belongs to (more than one may apply and others may be added):

   - control data input and checking
   - file handling
   - geometrical processing
   - framework and infrastructure
   - magnetic field processing

   The *physical characteristics* should be defined by stating which of the following descriptions apply to the module (more than one may apply and others may be added):

   - data module
   - execution module
   - function
   - main program unit
   - object-oriented module
   - subroutine

   Purpose This section describes the purpose of the module. It traces back to the software requirements which justify the existence of the module either explicitly by citing the software requirement or implicitly by saying that this component is a subordinate of another module.

   Function This section describes the function of the module. It should describe what is does rather than how it does it, which should be described in the processing section.

   Processing (Optional) This section describes the processing done by the module. It should describe how the module does what it does. The use of pseudocode is encouraged.

   Dependencies (Optional) This section describes what needs to have been done before this module and its subordinate modules can be used.

3. Every subroutine or function should have a comment associated with it saying what it does. A single line comment is usually inadequate, but may be sufficient if details of the code are published or described elsewhere and doxygen-produced documentation is available.

4. Every variable and type should have a comment associated with it saying what it is. Public variables and dummy arguments will normally need longer descriptions than local variables. Every component of a derived type should have a comment. Multidimensional arrays will normally need longer descriptions than scalar variables because it is necessary to describe what each dimension represents.

5. Comments for all public quantities should begin with `!>` or `!<` so that doxygen picks them up, with `!!` used for intermediate comment lines. Private quantities can be commented using `!` (or isolated `!!`) depending on whether or not the programmer deems it helpful for the information to be included in the html detailed design documentation generated by doxygen.

The use of horizontal ruled lines to separate subprograms in modules should be used to aid readability of the program listing.

2.17 Protective markings and version control

Every source file should contain the following header at the top of the file. The text between the `!begin rubric` and `!end rubric` can be updated automatically when necessary.

```
!begin rubric
!*****************************************************************************
!*                United Kingdom Atomic Energy Authority                     *
!*                                                                           *
!*                                                                           *
!*                       SMITER 2015 RELEASE 1.0                             *
!*                                                                           *
!*****************************************************************************
!*                                                                           *
!* This is an unpublished work created in 2015. It may be copied,            *
!* distributed and used as set out in the standard terms and conditions for  *
!* use of IMAS software.                                                     *
!*                                                                           *
!* Since these conditions had not been finalised at time of delivery, there  *
!* is the rider that IMAS terms and conditions are expected to exclude the   *
!* use of the software for activities unrelated to the ITER project.         *
!* As an interim measure, it is here explicitly stated that this software    *
!* must not be used for for nuclear reactor design activities unrelated to   *
!* ITER  without the express written consent of UKAEA.                       *
!*                                                                           *
!* Author: Wayne Arter                                                       *
! Copyright (c) 2015, UKAEA and ITER Organisation                           *
!*****************************************************************************
!
!----------------------------------------------------------------------------
!end rubric
```

Version control will be by `git`. Different deliveries to the customer should have an

explicit (different) Release number in the rubric. For development work, the branch of the git repository used by the developer needs to be recorded, then the developer must ensure that the code executed is the latest version on the branch *or* explicitly record the version.

2.18 File templates

Template files are shown in Appendix A. The first one `template_m.f90` shows the general layout and documentation for a module, and the second one, `oo_m.f90`, gives the design pattern for an object-oriented module. Developers should copy the appropriate template when creating a new module.

Template files should be included in the git repository. Editors such as Xemacs should be customised so that the template file is available from a menu to encourage developers to make use of the template.

2.19 Extensions

The only extensions to Fortran 95 that should be used are the use of `getarg` and `iargc` for accessing the command line arguments and the use of `flush` for flushing output to files. Fortran 2003 introduces standard routines `command_argument_count` and `get_command_argument` but currently these are not as widely available as `getarg` and `iargc`. `flush` is widely available and is part of Fortran 2003.

It is assumed that the Fortran 95 compiler used is compliant with extensions defined in ISO/IEC TR 15581:2001(E) [4], so that allocatable arrays are allowed as components of derived types. Note that this ISO document is formally a revision of the Fortran 95 standard.

There are many other features of Fortran 2003 which are desirable but none is as vital as the features mentioned above and they should be avoided.

2.20 Specify intent for dummy arguments

Developers should always specify the intent for the dummy arguments of subroutines and functions because doing so can help detect errors. It also makes the code easier to understand and may allow the compiler to produce faster code.

2.21 Use pure or elemental

Use the `pure` or `elemental` prefix for procedures which are pure or elemental. This can make them easier to understand and allows them to be used in ways which may enhance the performance of the code.

2.22 Use only

Use the `only` keyword with `use` statements if only a few names from a module are being used. This makes the code easier to understand and more robust, however is arguably the least important of all the requirements placed on the developer.

2.23 Use keywords for long argument lists

Use keywords for argument lists if there are more than a few arguments. The exact point at which keywords should start to be used depends on how regular the argument list is. Consider using optional arguments if some of the arguments can have default values or are not always needed.

2.24 Large automatic arrays

Do not use large automatic arrays. Allocate storage for large arrays using the `allocate` statement. Automatic arrays are usually put on the stack and stack size is usually limited.

2.25 Target attribute

Use the `target` attribute with the `allocatable` attribute when you are going to allocate storage and associate a pointer with it. This makes the design intention clearer than using a `pointer` attribute and may allow the compiler to produce faster code (although there should be no significant speed difference with a good compiler). The `pointer` attribute should be reserved for when you are using it to define an alias.

It is assumed that the Fortran 95 compiler used is compliant with extensions defined in ISO/IEC TR 15581:2001(E) [4], so that allocatable arrays are allowed as components of derived types. If it is not, then the `pointer` attribute must be used instead to define allocatable components of a derived type.

Do not use

```
integer, pointer :: grpcode(:)
type(node_tree), pointer, public :: treenode(:)
...
allocate(grpcode(ngrp), stat=istat)
...
allocate(treenode(nnode), stat=istat)
```

when the following would be clearer:

```
integer, allocatable, target :: grpcode(:)
```

```
type(node_tree), allocatable, target, public :: treenode(:)
...
allocate(grpcode(ngrp), stat=istat)
...
allocate(treenode(nnode), stat=istat)
```

2.26 Deprecated features

Avoid `equivalence` statements, `common` blocks, the `block data` program unit, and the `entry` statement.

2.27 Shape disagreement

Avoid shape disagreement between dummy arguments and actual arguments unless absolutely necessary. This type of disagreement will usually only be necessary if it is wished to view the array as having different ranks in different parts of the code. `reshape` could be used to avoid the disagreement but this involves two additional copy operations. Code which uses shape disagreement should make the implicit reshaping of the array as explicit as possible:

```
real, dimension(1000) :: x

call comp( y=x(101:200), & ! Effectively y=reshape( x(101:200), (/10,10/)
   &  m=10 , n=10 )
...
subroutine comp(y, m, n)
real, dimension(m,n), intent(inout) :: y
...
```

2.28 Character length disagreement

Avoid character length disagreement between dummy arguments and actual arguments.

2.29 Redundant features

Avoid redundant features. These include the `include` line, the `do while` loop construct, the `dimension` and `parameter` statements and specific names of intrinsic functions e.g. `float`, `dsin` etc.

Use `do` and `end do` instead of using statements with numerical labels.

Use the new forms of relation operators ==, /=, <, <=, >, and >= instead of the old forms. *The new forms are more compact and easier to read than the old forms*.

2.30 Use simple constructs

Use simple constructs. Complicated or obscure constructs may be hard to understand and are more likely to suffer from compiler bugs.

Do not replace simple code with more complicated code to avoid bugs in particular compilers. Get the compilers fixed or if absolutely necessary issue temporary patches for the code for use with the buggy compilers.

The developer should not optimise code prematurely. If it is found necessary to optimise code in a way which makes it harder to understand the developer should consider having a switch which allows the simpler version to be selected.

2.31 Include stubs for future development

Include comments, derived types, and dummy routines for future extensions. These can act as a reminder for when the features need to be implemented. Make sure unused features are marked "not used" or are commented out with a clear comment about what they are for.

2.32 Named constants

Consider using named constants rather than literal values to make changing the types of variables easier:

```
real(rk2), parameter :: one = 1.0_kr2
```

The module `const_numphys_h` provides numerical values for a small range of different named numeric constants, such as π. Inspect the contents of the module to see what is currently available, and consider adding new numeric constants here, observing the convention that all such variables begin `const_`.

Chapter 3

Architectural Recommendations

This section describes recommendations which are architectural in nature although some of the recommendations in Section 2 may be considered as being architectural as well.

3.1 Explicit interfaces

This is the most important architectural recommendation. Explicit interfaces must be used throughout the code. This means using Fortran 95 modules (or possibly interface blocks to interface to legacy code). The use of explicit interfaces allows the compiler to check that actual and dummy arguments match in number and type, which reduces programming errors. Explicit interfaces also allow the use of keywords, optional arguments and generic interfaces which make the code easier to understand and maintain.

3.2 Parameterised types

Do not hardwire the types of real and integer variables. It may be helpful both for code development and distribution on different hardware to be able to change easily between 4, 8 and 16 byte real numbers for different variables, and between 1, 2, 4, and 8 byte integers for different variables (codes routinely overflow 4 byte integers in computing how much memory or disk space is needed). The module `const_kind_m` is provided for this purpose, and should be inspected to ascertain the available types.

3.3 Save attributes

Module variables may become undefined if the module goes out of scope. Either use the save attribute on module variables or ensure that the module remains in scope. It is safer to use the `save` attribute on module variables because it is difficult to make sure that the module variables remain in scope.

3.4 Check status flags

Always check the error status from `open`, `read`, `write` and `allocate`. The module `log_m` is provided for this purpose, and contains subroutines

```
log_open_check ---- checks file opening
log_read_check ---- checks result of read
log_write_check ---- checks result of write
log_alloc_check ---- checks array allocation
```

The errors from `close`, `deallocate` may also be significant, but can largely be avoided for example by the use of

```
if (allocated(array)) deallocate(array)
```

If it is necessary to deal with status flags in line, the style:

```
write(luout,stat=istat)
if ( istat /= 0 ) then
  ...
end if
...
write(luout,stat=istat)
if ( istat /= 0 ) then
  ...
end if
```

is better than

```
write(luout,err=999)
...
write(luout,err=999)
...
999 continue
...
```

because the error handling code is closer to the place where the error occurs.

3.5 Choice of data types

Fortran 95 has a much richer choice of data types and design patterns available than Fortran 77. Do not just use the Fortran 77 approach without considering the alternatives.

One example where there are more choices is in how to have an array of variable-sized arrays. There are three main ways to do this:

1. have an array of indices pointing into a single array which actually holds the data,

2. have an array of pointers to arrays which are associated with a section of a single array or

3. have an array of pointers to arrays which are associated with storage using the `allocate` statement.

Fortran 95 does not allow arrays of pointers which are required for options 2 and 3 but it does allow arrays of derived types which contain a single component which is a pointer to an array:

```
type array_ptr_t
  real(dbl), dimension(:), pointer :: ptr !! pointer to array
end type array_ptr_t
...
type(array_ptr_t), dimension(10) :: x !! array of pointers to arrays
...
real(dbl), dimension(100), target : storage
...
x(2)%ptr => storage(5:104)          ! case 2
allocate (x(1)%ptr(100),stat=istat) ! case 3
```

Option 3 is the safest option and option 1 is the least safe option. Option 2 is intermediate but may be useful if developers want to manage storage themselves instead of letting the system manage storage with `allocate` and `deallocate`. An implementation using option 3 can easily be changed to use option 2 by replacing the `allocate` statement with code to manage storage in an array.

Options 2 and 3 use slightly more storage than option 1 because a pointer will usually consist of an address and a small number of integers giving the rank, dimensions and strides of the array. The pointer probably needs between 3 and 6 times as much storage as a simple integer index. This is usually a small price to pay for the extra safety and improved clarity of the code.

With option 3 it would be better to have an array of allocatable arrays. This can be realised by using a derived type with a single component which is an allocatable array, instead of a pointer as described above. Note that this requires the TR15581 [4] extension to Fortran 95. We shall assume that compilers for QPROG will have this extension.

3.6 Record lengths for unformatted direct access files

The specification of record length for unformatted direct access input and output is mandatory but the unit used in the `recl=` specifier in an `open` statement is unspecified.

The `iolength=` specifier should be used with the `inquire` statement to determine the record length which should be used. The `inquire` statement should be used

with all possible input/output lists and the maximum length used. This avoids making assumptions about how data are packed:

```
real(kr2), dimension(10) :: x
integer(ki3), dimension(5) :: i
integer :: length_max, length
...
length_max = 0
inquire(iolength=length) x(1:5),i(1:2)
length_max = max(length, length_max)
inquire(iolength=length) x(1:4),i(1:3)
length_max = max(length, length_max)
...
open(..., recl=length_max, ...)
...
write(...)  x(1:5),i(1:2)
write(...)  x(1:4),i(1:3)
```

It is however reasonable to assume that inserting an extra item in an input/output list will not make the record length shorter so that in the example above it would also be acceptable to have:

```
write(...)  x(1:5),i(1)
write(...)  x(1:4),i(1:2)
write(...)  x(1:4),i(1:3)
...
```

We don't expect to use formatted direct access files.

3.7 Estimating storage sizes

The amount of storage required for a type can be estimated using the `size` and `transfer` functions:

```
program sizeof
implicit none
integer, parameter :: ki1 = selected_int_kind(2)
integer, parameter :: ki1_bits = bit_size(1_ki1)
integer, parameter :: ki1_bytes = ki1_bits/8
type x
  real :: y
  character :: c
  integer :: i
end type x
integer, parameter :: real_bytes = ki1_bytes*size(transfer(1.0,(/1_ki1/)))
integer, parameter :: char_bytes = ki1_bytes*size(transfer(' ',(/1_ki1/)))
integer, parameter :: int_bytes = ki1_bytes*size(transfer(1,(/1_ki1/)))
```

```
integer, parameter :: x_bytes = ki1_bytes*size(transfer(x(1.0,'',1),(/1_k
write(*,*) 'integer kind ',ki1,' contains ',ki1_bits,  ' bits'
write(*,*) 'integer kind ',ki1,' contains ',ki1_bytes,  ' bytes'
write(*,*) 'real contains ',real_bytes,' bytes'
write(*,*) 'character contains ',char_bytes,' bytes'
write(*,*) 'integer contains ',int_bytes,' bytes'
write(*,*) 'type x contains ',x_bytes,' bytes'
end program sizeof
```

3.8 Precision and kinds

The module `const_kind_m` provides a set of commonly used variable precisions, such as are eg. equivalent to single and double precision reals. Inspect the contents of the module to see what is currently available, and consider adding new variable kinds here, observing the convention that all such variables begin `k`.

3.9 Error handler and program termination

Routines should not use the `stop` statement directly. The program should only be terminated using a standard interface, via a routine defined in the `log_m` module, specifically `log_error`. The complete error handling and logging system also requires routines `log_init` and `log_close`, and there is a facility to record information using subroutine `log_value`.

3.10 Code monitoring

The module `clock_m` is provided to measure cpu times between different calls. Inspect the contents of the module to see what is currently available.

The module `date_time_m` provides functions which return the current date in either a long or short format, and its routines should be used whenever dates or times are required.

3.11 Physical and numerical constants

The module `const_numphys_h` provides numerical values for a small range of different physical and numerical constants, including π. Inspect the contents of the module to see what is currently available, and consider adding new physical constants here, observing the convention that all such variables begin `const_`.

3.12 Selection of I/O units

A module should be provided for allocating I/O units. This enhances portability by avoiding assumptions about which unit numbers are used for standard purposes.

Bibliography

[1] J.W. Eastwood and J.G. Morgan. Fortran 95 Conventions for FISPACT. Technical Report CEM/081203/WI/1 Issue 2, Culham Electromagnetics Ltd, 2009.

[2] ISO/IEC. Information technology, programming languages, Fortran. International Standard. Part 1: Base Language. Technical Report 1539-1, first edition 1997-12-15, 1997.

[3] D. van Heesch. doxygen Manual for version 1.7.1. Technical report, www.doxygen.org, 2007.

[4] ISO/IEC. Information technology, programming languages, Fortran, enhanced data type facililities. Technical Report TR 15581(E), second edition, 2001.

[5] W. Arter, J.W. Eastwood, and J.G. Morgan. QPROG Example, Generated by Doxygen 1.8.6. Technical report, UKAEA, September 2015.

APPENDICES

Appendix A

Sample Modules

A.1 A template module

```
module template_m

  use another_m, only: another_var1 !<  Use only another_var1 from this module
  use second_m,  second_var1 => v   !<  Use all module with second_var renamed v
  use precision_m, only krb         !<  Use base real precision

  implicit none
  private

! public subroutines

  public template_usersub

! public types

!> Example type 1
  type,public :: template_xyz_t
     integer :: num = 0                !< Number of things
     integer,pointer :: ptr => null() !< Pointer to first thing
  end type template_xyz_t

! public variables

!> The description of the first public variable in the module.
!! Public variables will usually require at least a full line of
!! description.
!!
  integer,dimension(1:3),public :: template_eg1 !< local variable
!> second public variable
  integer,dimension(:),allocatable,public :: template_eg2 !< local variable

! private variables
```

```
  integer,dimension(1:3) :: eg1 !< example variable 1
!> Example variable 2 has a longer description
!! which is too long to fit at the end of the line.
  integer,dimension(1:3) :: eg2 = 0 !< local variable
  integer,dimension(1:3) :: eg3 = 0            !< example variable 3
  integer,dimension(:),allocatable :: eg4     !< example variable 4
  integer,target                   :: eg5 = 0 !< example variable 5


  contains
!-----------------------------------------------------------------------
!> First user subroutine with at least a line of description
!! saying what it does.
!!
subroutine template_usersub(arg1)
  ! arguments
  real(krb), intent(in) :: arg1 !< argument 1
  ! local variables
  integer :: i !< local variable description
  integer :: j !< local variable description
  integer :: k !< local variable description

  ! user code
  call module_xyz()

end subroutine template_usersub
!-----------------------------------------------------------------------
!> Second user subroutine with at least a line of description
!! saying what it does.
!!
subroutine template_usersub2(arg2)
  ! arguments
  integer(krb), intent(out) :: arg2 !< argument 1
  ! local variables
  integer :: i !< local variable description
  integer :: j !< local variable description
  integer :: k !< local variable description

  ! user code
  arg2=10

end subroutine template_usersub
!-----------------------------------------------------------------------

end module template_m
```

A.2 Object-oriented factory pattern module

```
module oo_m

! use statements
```

```
! default to explicit typing and private
  implicit none
  private

! public subroutines

  public oo_init
  public oo_delete
  public oo_method1

! public types

!> Type to contain non-static components of objects.
!! All components should have default initial values if possible.
  type, public :: oo_t
     integer :: comp1 = 0  !< component 1
     integer :: comp2 = 0  !< component 2
  end type oo_t

! public variables

! private types

! private variables

  contains
!---------------------------------------------------------------------------
!> Constructor for object.
subroutine oo_init(self)

  ! arguments
  !> Variable to contain nonstatic data for self object must be first argument and be ca
  type(oo_t), intent(out) :: self !< local variable
end subroutine oo_init
!---------------------------------------------------------------------------
!> First method for module.
subroutine oo_method1(self,other_oo)

  ! arguments
  type(oo_t), intent(inout) :: self !< Data for self object must be first argument and b
  type(oo_t), intent(in) :: other_oo !< Other object of same type as self

end subroutine oo_method1
!---------------------------------------------------------------------------
!> Destructor for object.
!!
!! The destructor deallocates all storage which the object is
!! responsible for. It is good programming style to make sure that the
!! object cannot be used after it has been deleted by invalidating
!! all the data i.e. nullifying pointers and setting non-pointer
!! variables to values which will generate an error if used.
subroutine oo_delete(self)
```

```
  ! arguments
  type(oo_t), intent(inout) :: self !< data for self object
end subroutine oo_delete
!-----------------------------------------------------------------------

end module oo_m
```

A.3 Big object-oriented methods module

```
module bigobj_m

  use bigobj_h
  use log_m
  use const_numphys_h
  use const_kind_m

  implicit none
  private

! public subroutines
  public :: &
  bigobj_initfile,  & !< open file
  bigobj_readcon,  & !< read data from file
  bigobj_solve,  & !< generic subroutine
  bigobj_userdefined,  & !< user-defined function
  bigobj_fn, &  !< general external function call
  bigobj_dia, &  !< object diagnostics to log file
  bigobj_initwrite, & !< open new file, making up name
  bigobj_write, &  !< write out object
  bigobj_writeg, &  !< write out object as gnuplot
  bigobj_writev, &  !< write out object as vtk
  bigobj_delete, & !< delete object
  bigobj_close, & !< close file
  bigobj_closewrite !< close write file

! private variables
  character(*), parameter :: m_name='bigobj_m' !< module name
  integer(ki4)  :: status   !< error status
  integer(ki4), save  :: ninbo=5     !< control file unit number
  integer(ki4), save  :: noutbo=6      !< output file unit number
  character(len=80), save :: controlfile !< control file name
  character(len=80), save :: outputfile !< output file name
  integer(ki4) :: i !< loop counter
  integer(ki4) :: j !< loop counter
  integer(ki4) :: k !< loop counter
  integer(ki4) :: l !< loop counter
  integer(ki4) :: ij !< loop counter
  integer(ki4)  :: ilog      !< for namelist dump after error

  contains
!-----------------------------------------------------------------------
!> open file
subroutine bigobj_initfile(file,channel)

  !! arguments
  character(*), intent(in) :: file !< file name
  integer(ki4), intent(out),optional :: channel   !< input channel for object data struc
  !! local
  character(*), parameter :: s_name='bigobj_initfile' !< subroutine name
```

```
  logical :: unitused !< flag to test unit is available

  if (trim(file)=='null') then
     call log_error(m_name,s_name,1,log_info,'null filename ignored')
     return
  end if

  !! get file unit
  do i=99,1,-1
     inquire(i,opened=unitused)
     if(.not.unitused)then
        ninbo=i
        if (present(channel)) channel=i
        exit
     end if
  end do

  !! open file
  controlfile=trim(file)
  call log_value("Control data file",trim(controlfile))
  open(unit=ninbo,file=controlfile,status='OLD',iostat=status)
  if(status/=0)then
     !! error opening file
     print '("Fatal error: Unable to open control file, ",a)',controlfile
     call log_error(m_name,s_name,2,error_fatal,'Cannot open control data file')
     stop
  end if

end subroutine bigobj_initfile
!---------------------------------------------------------------------
!> read data from file
subroutine bigobj_readcon(selfn,channel)

  !! arguments
  type(bonumerics_t), intent(out) :: selfn !< type which data will be assigned to
  integer(ki4), intent(in),optional :: channel   !< input channel for object data struct

  !! local
  character(*), parameter :: s_name='bigobj_readcon' !< subroutine name
  character(len=80) :: bigobj_formula !< formula to be used
  integer(ki4), parameter :: MAX_NUMBER_OF_PARAMETERS=10 !< maximum number of parameters
  real(kr8) :: power_split !< variable with meaningful name

  real(kr8), dimension(MAX_NUMBER_OF_PARAMETERS) :: general_real_parameters  !< local va
  integer(ki4), dimension(MAX_NUMBER_OF_PARAMETERS) :: general_integer_parameters  !< lo
  integer(ki4) :: number_of_real_parameters  !< local variable
  integer(ki4) :: number_of_integer_parameters  !< local variable

  !! bigobj parameters
  namelist /bigobjparameters/ &
 &power_split, bigobj_formula, &
 &general_real_parameters, number_of_real_parameters, &
 &general_integer_parameters, number_of_integer_parameters
```

```
 !! set default bigobj parameters
 power_split=0.5_kr8

 bigobj_formula='unset'
 general_real_parameters=0
 general_integer_parameters=0
 number_of_real_parameters=0
 number_of_integer_parameters=0

 if(present(channel).AND.channel/=0) then
    !! assume unit already open and reading infile
    ninbo=channel
 end if

 !!read bigobj parameters
 read(ninbo,nml=bigobjparameters,iostat=status)
 if(status/=0) then
    !!dump namelist contents to logfile to assist error location
    print '("Fatal error reading bigobj parameters")'
    call log_getunit(ilog)
    write(ilog,nml=bigobjparameters)
    call log_error(m_name,s_name,1,error_fatal,'Error reading bigobj parameters')
 end if

 call lowor(bigobj_formula,1,len_trim(bigobj_formula))
 !! check for valid data

 formula_chosen: select case (bigobj_formula)
 case('unset','exp')
    if(power_split<0.OR.power_split>1) &
&   call log_error(m_name,s_name,11,error_fatal,'power_split must be >=0 and <=1')

 case('expdouble')
    if(power_split<0.OR.power_split>1) &
&   call log_error(m_name,s_name,21,error_fatal,'power_split must be >=0 and <=1')

 case('userdefined')
    if(number_of_real_parameters<0) &
&   call log_error(m_name,s_name,44,error_fatal,'number of real parameters must be >=0'
    if(number_of_real_parameters>MAX_NUMBER_OF_PARAMETERS) then
       call log_value("max number of real parameters",MAX_NUMBER_OF_PARAMETERS)
       call log_error(m_name,s_name,45,error_fatal,'too many parameters: increase MAX_N
    end if
    if(number_of_integer_parameters<0) &
&   call log_error(m_name,s_name,46,error_fatal,'number of integer parameters must be >
    if(number_of_integer_parameters>MAX_NUMBER_OF_PARAMETERS) then
       call log_value("max number of integer parameters",MAX_NUMBER_OF_PARAMETERS)
       call log_error(m_name,s_name,47,error_fatal,'too many parameters: increase MAX_N
    end if
    if(number_of_integer_parameters==0.AND.number_of_real_parameters==0) &
&   call log_error(m_name,s_name,48,error_fatal,'no parameters set')
```

```
  end select formula_chosen

  !! store values
  selfn%formula=bigobj_formula

  selfn%f=power_split

  !! allocate arrays and assign

  selfn%nrpams=number_of_real_parameters
  selfn%nipams=number_of_integer_parameters

  formula_allocate: select case (bigobj_formula)

  case('userdefined')
     if (number_of_real_parameters>0) then
        allocate(selfn%rpar(number_of_real_parameters), stat=status)
        call log_alloc_check(m_name,s_name,65,status)
        selfn%rpar=general_real_parameters(:number_of_real_parameters)
     end if
     if (number_of_integer_parameters>0) then
        allocate(selfn%npar(number_of_integer_parameters), stat=status)
        call log_alloc_check(m_name,s_name,66,status)
        selfn%npar=general_integer_parameters(:number_of_integer_parameters)
     end if
  case default
  end select formula_allocate

end  subroutine bigobj_readcon
!---------------------------------------------------------------------
!> generic subroutine
subroutine bigobj_solve(self)

  !! arguments
  type(bigobj_t), intent(inout) :: self !< module object
  !! local
  character(*), parameter :: s_name='bigobj_solve' !< subroutine name

  self%pow=bigobj_fn(self,0._kr8)

end subroutine bigobj_solve
!---------------------------------------------------------------------
!> output to log file
subroutine bigobj_dia(self)

  !! arguments
  type(bigobj_t), intent(inout) :: self !< module object
  !! local
  character(*), parameter :: s_name='bigobj_dia' !< subroutine name

  call log_value("power ",self%pow)

end subroutine bigobj_dia
```

```
!-----------------------------------------------------------------------
!> userdefined function
function bigobj_userdefined(self,psi)

  !! arguments
  type(bigobj_t), intent(in) :: self !< module object
  real(kr8) :: bigobj_userdefined !< local variable
  real(kr8), intent(in) :: psi !< position in \f$ \psi \f$

  !! local variables
  character(*), parameter :: s_name='bigobj_userdefined' !< subroutine name
  real(kr8) :: pow !< local variable
  real(kr8) :: zpos !< position
  integer(ki4) :: ilocal !< local integer variable

  zpos=psi
  pow=0._kr8
  !> user defines \Tt{pow} here
  !! .....
  !! return bigobj
  bigobj_userdefined=pow

end function bigobj_userdefined
!-----------------------------------------------------------------------
!> general external function call
function bigobj_fn(self,psi)

  !! arguments
  type(bigobj_t), intent(in) :: self !< module object
  real(kr8) :: bigobj_fn !< local variable
  real(kr8), intent(in) :: psi !< position in \f$ \psi \f$

  !! local variables
  character(*), parameter :: s_name='bigobj_fn' !< subroutine name
  real(kr8) :: pow !< local variable

  pow=0._kr8
  !! select bigobj
  formula_chosen: select case (self%n%formula)
  case('userdefined')
     pow=bigobj_userdefined(self,psi)
  end select formula_chosen

  !! return bigobj
  bigobj_fn=pow

end function bigobj_fn
!-----------------------------------------------------------------------
!> open new file, making up name
subroutine bigobj_initwrite(fileroot,channel)

  !! arguments
  character(*), intent(in) :: fileroot !< file root
```

```
  integer(ki4), intent(out),optional :: channel   !< output channel for object data stru
  !! local
  character(*), parameter :: s_name='bigobj_initwrite' !< subroutine name
  logical :: unitused !< flag to test unit is available
  character(len=80) :: outputfile !< output file name

  !! get file unit
  do i=99,1,-1
     inquire(i,opened=unitused)
     if(.not.unitused)then
        if (present(channel)) channel=i
        exit
     end if
  end do
  noutbo=i

  !! open file
  outputfile=trim(fileroot)//"_bigobj.out"
  call log_value("Control data file",trim(outputfile))
  open(unit=noutbo,file=outputfile,status='NEW',iostat=status)
  if(status/=0)then
     open(unit=noutbo,file=outputfile,status='REPLACE',iostat=status)
  end if
  if(status/=0)then
     !! error opening file
     print '("Fatal error: Unable to open output file, ",a)',outputfile
     call log_error(m_name,s_name,1,error_fatal,'Cannot open output data file')
     stop
  end if

end subroutine bigobj_initwrite
!-----------------------------------------------------------------------
!> write bigobj data
subroutine bigobj_write(self,channel)

  !! arguments
  type(bigobj_t), intent(in) :: self   !< bigobj data structure
  integer(ki4), intent(in), optional :: channel   !< output channel for bigobj data stru

  !! local
  character(*), parameter :: s_name='bigobj_write' !< subroutine name
  integer(ki4) :: iout   !< output channel for bigobj data structure

  !! sort out unit
  if(present(channel)) then
     iout=channel
  else
     iout=noutbo
  end if

  write(iout,*,iostat=status) 'bigobj_formula'
  call log_write_check(m_name,s_name,18,status)
  write(iout,*,iostat=status) self%n%formula
```

```
  call log_write_check(m_name,s_name,19,status)
  write(iout,*,iostat=status) 'f'
  call log_write_check(m_name,s_name,20,status)
  write(iout,*,iostat=status) self%n%f
  call log_write_check(m_name,s_name,21,status)
  write(iout,*,iostat=status) 'nrpams'
  call log_write_check(m_name,s_name,46,status)
  write(iout,*,iostat=status) self%n%nrpams
  call log_write_check(m_name,s_name,47,status)
  if (self%n%nrpams>0) then
     write(iout,*,iostat=status) 'real_parameters'
     call log_write_check(m_name,s_name,48,status)
     write(iout,*,iostat=status) self%n%rpar
     call log_write_check(m_name,s_name,49,status)
  end if
  write(iout,*,iostat=status) 'nipams'
  call log_write_check(m_name,s_name,50,status)
  write(iout,*,iostat=status) self%n%nipams
  call log_write_check(m_name,s_name,51,status)
  if (self%n%nipams>0) then
     write(iout,*,iostat=status) 'integer_parameters'
     call log_write_check(m_name,s_name,52,status)
     write(iout,*,iostat=status) self%n%npar
     call log_write_check(m_name,s_name,53,status)
  end if

end subroutine bigobj_write
!-----------------------------------------------------------------------
!> write object data as gnuplot
subroutine bigobj_writeg(self,select,channel)

  !! arguments
  type(bigobj_t), intent(in) :: self   !< object data structure
  character(*), intent(in) :: select  !< case
  integer(ki4), intent(in), optional :: channel   !< output channel for bigobj data stru

  !! local
  character(*), parameter :: s_name='bigobj_writeg' !< subroutine name
  integer(ki4) :: iout   !< output channel for bigobj data structure

  call log_error(m_name,s_name,1,log_info,'gnuplot file produced')

  plot_type: select case(select)
  case('cartesian')

  case default

  end select plot_type

end subroutine bigobj_writeg
!-----------------------------------------------------------------------
!> write object data as vtk
subroutine bigobj_writev(self,select,channel)
```

```
  !! arguments
  type(bigobj_t), intent(in) :: self   !< object data structure
  character(*), intent(in) :: select  !< case
  integer(ki4), intent(in), optional :: channel   !< output channel for bigobj data stru

  !! local
  character(*), parameter :: s_name='bigobj_writev' !< subroutine name
  integer(ki4) :: iout   !< output channel for bigobj data structure

  call log_error(m_name,s_name,1,log_info,'vtk file produced')

  plot_type: select case(select)
  case('cartesian')

  case default

  end select plot_type

end subroutine bigobj_writev
!-----------------------------------------------------------------------
!> close write file
subroutine bigobj_closewrite

  !! local
  character(*), parameter :: s_name='bigobj_closewrite' !< subroutine name

  !! close file
  close(unit=noutbo,iostat=status)
  if(status/=0)then
     !! error closing file
     print '("Fatal error: Unable to close output file, ",a)',outputfile
     call log_error(m_name,s_name,1,error_fatal,'Cannot close output data file')
     stop
  end if

end subroutine bigobj_closewrite
!-----------------------------------------------------------------------
!> delete object
subroutine bigobj_delete(self)

  !! arguments
  type(bigobj_t), intent(inout) :: self !< module object
  !! local
  character(*), parameter :: s_name='bigobj_delete' !< subroutine name

  formula_deallocate: select case (self%n%formula)
  case('userdefined')
     if (self%n%nrpams>0) deallocate(self%n%rpar)
     if (self%n%nipams>0) deallocate(self%n%npar)
  case default
  end select formula_deallocate
```

```
end subroutine bigobj_delete
!---------------------------------------------------------------------
!> close file
subroutine bigobj_close

  !! local
  character(*), parameter :: s_name='bigobj_close' !< subroutine name

  !! close file
  close(unit=ninbo,iostat=status)
  if(status/=0)then
     !! error closing file
     print '("Fatal error: Unable to close control file, ",a)',controlfile
     call log_error(m_name,s_name,1,error_fatal,'Cannot close control data file')
     stop
  end if

end subroutine bigobj_close

end module bigobj_m
```

A.4 Big object definition module

```
module bigobj_h

  use const_kind_m

! public types

! Note these could be merged if the object is instantiated by its parameters  only
!> parameters describing how to construct object
  type, public :: bonumerics_t
     character(len=80) :: formula !< bigobj formula
     real(kr8) :: f !< power split (math variable name allowed)
     integer(ki4) :: nrpams !< number of real parameters
     integer(ki4) :: nipams !< number of integer parameters
     real(kr8), dimension(:), allocatable   :: rpar !< general real parameters
     integer(ki4), dimension(:), allocatable   :: npar !< general integer parameters
  end type bonumerics_t

! type which defines/instantiates the object
  type, public :: bigobj_t
     real(kr8) :: pow !< power
     type(bonumerics_t) :: n !< control  parameters
  end type bigobj_t

end module bigobj_h
```

A.5 Program control object-oriented methods module

```
module qcontrol_m

  use const_kind_m
  use log_m
  use qcontrol_h
  use bigobj_h
  use bigobj_m

  implicit none
  private

! public subroutines
  public :: &
 &qcontrol_init, & !< open input control data file
 &qcontrol_close, & !< close input control data file
 &qcontrol_read !< read data for this run

! private variables
  character(*), parameter :: m_name='qcontrol_m' !< module name
  integer(ki4)  :: status   !< error status
  integer(ki4), save  :: nin      !< control file unit number
  integer(ki4) :: i !< loop counter
  integer(ki4) :: j !< loop counter
  integer(ki4) :: k !< loop counter
  integer(ki4) :: l !< loop counter
  character(len=80) :: root !< file root

  contains
!-----------------------------------------------------------------------
!> open input control data file
subroutine qcontrol_init(fileroot)

  !! arguments
  character(*), intent(in) :: fileroot !< file root
  !! local
  character(*), parameter :: s_name='qcontrol_init' !< subroutine name
  logical :: unitused !< flag to test unit is available
  character(len=80) :: controlfile !< control file name


  !! get file unit
  do i=99,1,-1
     inquire(i,opened=unitused)
     if(.not.unitused)then
        nin=i
        exit
     end if
  end do

  !! open file
```

```
  controlfile=trim(fileroot)//".ctl"
  root=fileroot
  call log_value("Control data file",trim(controlfile))
  open(unit=nin,file=controlfile,status='OLD',iostat=status)
  if(status/=0)then
     !! error opening file
     print '("Fatal error: Unable to open control file, ",a)',controlfile
     call log_error(m_name,s_name,1,error_fatal,'Cannot open control data file')
     stop
  end if


end  subroutine qcontrol_init
!-----------------------------------------------------------------------
!> close input control data file
subroutine qcontrol_close

  !! local
  character(*), parameter :: s_name='qcontrol_close' !< subroutine name

  !! close file unit
  close(unit=nin)

end  subroutine qcontrol_close
!-----------------------------------------------------------------------
!> read data for this run
subroutine qcontrol_read(file,param,bonumerics,plot)

  !! arguments
  type(qfiles_t), intent(out) :: file !< file names
  type(qparams_t), intent(out) :: param !< control parameters
  type(bonumerics_t), intent(out) :: bonumerics !< controls for bigobj
  type(qplots_t), intent(out) :: plot !< plot selectors

  !!local
  character(*), parameter :: s_name='qcontrol_read' !< subroutine name
  logical :: filefound !< true if file exists

  character(len=80) :: prog_input_file  !< qprog data input file
  character(len=80) :: prog_output_file  !< qprog data output file

  character(len=80) :: prog_control  !< option control parameter
  real(kr8) :: prog_realpar !< real control parameter
  integer(ki4) :: prog_intpar !< integer control parameter
  logical :: prog_logicpar !< logical control parameter

  logical :: plot_bigobjout !< bigobj output data selector
  logical :: plot_vtk !< vtk plot selector
  logical :: plot_gnu !< gnuplot plot selector

  !! file names
  namelist /progfiles/ &
 &prog_input_file, &
```

```
&prog_output_file

 !! misc parameters
 namelist /miscparameters/ &
&prog_control,&
&prog_realpar, prog_intpar, &
&prog_logicpar

 !! plot selection parameters
 namelist /plotselections/ &
&plot_bigobjout, &
&plot_vtk, &
&plot_gnu

 !---------------------------------------------------------------------
 !! read input file names
 prog_input_file='null'
 prog_output_file='null'
 read(nin,nml=progfiles,iostat=status)
 if(status/=0) then
    call log_error(m_name,s_name,1,error_fatal,'Error reading input filenames')
    print '("Fatal error reading input filenames")'
 end if

 file%bigobjdata = prog_input_file
 file%out = prog_output_file

 !!check file exists

 call log_value("qprog data file, prog_input_file",trim(file%bigobjdata))
 if(file%bigobjdata.NE.'null') then
    inquire(file=prog_input_file,exist=filefound)
    if(.not.filefound) then
       !! error opening file
       print '("Fatal error: Unable to find qprog data file, ",a)',prog_input_file
       call log_error(m_name,s_name,2,error_fatal,'qprog data file not found')
    end if
 end if

 !! create output file names from root

 !!bigobj file root
 file%bigobjout = trim(root)//"_bigobjout"
 !!vtk file
 file%vtk = trim(root)//"_vtk"
 !!gnu file roots
 file%gnu = trim(root)//"_gnu"

 !---------------------------------------------------------------------
 !! set default misc parameters
 prog_control = 'standard'
 prog_realpar = .0001_kr8
 prog_intpar = 1
```

```
  prog_logicpar = .false.

  !!read misc parameters
  read(nin,nml=miscparameters,iostat=status)
  if(status/=0) then
     call log_error(m_name,s_name,10,error_fatal,'Error reading misc parameters')
     print '("Fatal error reading misc parameters")'
  end if

  !! check for valid data
  if(prog_control/='standard') then
     call log_error(m_name,s_name,11,error_fatal,'only standard control allowed')
  end if
  ! positive real parameter
  if(prog_realpar<0._kr8) then
     call log_error(m_name,s_name,20,error_fatal,'realpar must be non-negative')
  end if
  ! positive integer parameter
  if(prog_intpar<=0) then
     call log_error(m_name,s_name,30,error_fatal,'intpar must be positive')
  end if
  if(prog_logicpar) &
 &call log_error(m_name,s_name,40,error_warning,'logicpar is true')

  param%control=prog_control
  param%realpar=prog_realpar
  param%intpar=prog_intpar
  param%logicpar=prog_logicpar

  !---------------------------------------------------------------------
  !! set default plot selections
  plot_bigobjout = .false.
  plot_vtk = .false.
  plot_gnu = .false.

  !!read plot selections
  read(nin,nml=plotselections,iostat=status)
  if(status/=0) then
     call log_error(m_name,s_name,50,error_fatal,'Error reading plot selections')
     print '("Fatal error reading plot selections")'
  end if

  !! store values
  plot%bigobjout = plot_bigobjout
  plot%vtk = plot_vtk
  plot%gnu = plot_gnu

end  subroutine qcontrol_read

end module qcontrol_m
```

A.6 Program control object definition module

```
module qcontrol_h

  use const_kind_m

!! public types

!> run control parameters
  type, public :: qparams_t
     character(len=80) :: control !< option control parameter
     real(kr8) :: realpar !< real control parameter
     integer(ki4) :: intpar !< integer control parameter
     logical :: logicpar !< logical control parameter
  end type qparams_t

!> file names
  type, public :: qfiles_t
     character(len=80)  :: out       !< output data
     character(len=80)  :: log           !< log file
     character(len=80)  :: bigobjdata         !< bigobj input data file
     character(len=80)  :: bigobjout         !< bigobj output data file
     character(len=80)  :: vtk   !< vtk file
     character(len=80)  :: gnu !< gnuplot file
  end type qfiles_t

!> plot output selectors
  type, public :: qplots_t
     logical  :: bigobjout !< bigobj output data selector
     logical  :: vtk   !< vtk plot selector
     logical  :: gnu !< gnuplot plot selector
  end type qplots_t

end module qcontrol_h
```

A.7 Main program

```
program p_qprog

  use const_kind_m
  use const_numphys_h
  use date_time_m
  use log_m
  use clock_m
  use gfile_m
  use vfile_m
  use qcontrol_h
  use qcontrol_m
  use bigobj_h
  use bigobj_m

  implicit none

! Local variables
  character(*), parameter :: m_name='qprog' !< module name
  type(qparams_t)     :: param      !< run control parameters
  type(qfiles_t)     :: file      !< names of files
  type(qplots_t)     :: plot      !< diagnostic plot selectors
  type(bigobj_t)  :: bigobj      !< bigobj object
  type(date_time_t) :: timestamp !< timestamp of run
  character(len=80) :: fileroot !< reference name for all files output by run

  integer(ki4):: nin=0 !< unit for other data
  integer(ki4):: nplot=0 !< unit for vtk files
  integer(ki4):: nprint=0 !< unit for gnuplot files
  integer(ki4):: i !< loop variable
  integer(ki4):: j !< loop variable

  character(len=80) :: ibuf !< character workspace
!-------------------------------------------------------------------------
!! initialise timing

  call date_time_init(timestamp)
  call clock_init(40)
  call clock_start(1,'run time')
!-------------------------------------------------------------------------
!! print header

  print *, '-----------------------------------------------------'
  print *, 'qprog: one-line program description '
  print *, '-----------------------------------------------------'
  print '(a)', timestamp%long
!-------------------------------------------------------------------------
!! file root

!! get file root from arg
  if(command_argument_count()<1) then
```

```
!! no file root specified
     print *, 'Fatal error: no file root name specified.'
     print *, 'To run qprog type at the command line:'
     print *, '   qprog fileroot'
     stop
  else
!!get fileroot
     call get_command_argument(1,value=fileroot)
  end if

!! start log
  call log_init(fileroot,timestamp)
!-------------------------------------------------------------------------
!! read control file

  call clock_start(2,'control initialisation time')
  call qcontrol_init(fileroot)
  call qcontrol_read(file,param,bigobj%n,plot)
  call clock_stop(2)
!-------------------------------------------------------------------------
!! other data for bigobj and/or read bigobj controls directly

  call clock_start(15,'initialisation time')
  call bigobj_initfile(file%bigobjdata,nin)
  !call bigobj_readcon(bigobj%n)
  call clock_stop(15)
!-------------------------------------------------------------------------
!! do the main work

  call clock_start(21,'evaluation time')

  select case(param%control)
  case default
     call bigobj_solve(bigobj)
  end select

  call clock_stop(21)
!-------------------------------------------------------------------------
!! data diagnostics to log file - optional

  call clock_start(31,'diagnostics time')

  select case(param%control)
  case default
     call bigobj_dia(bigobj)
  end select

  call clock_stop(31)
!-------------------------------------------------------------------------
!! vtk plot file(s) - optional

  call clock_start(32,'vtk plot time')
```

```
  select case(param%control)
  case default
     if(plot%vtk) then
        call vfile_init(file%vtk,'file header',nplot)
        call bigobj_writev(bigobj,'selector',nplot)
        call vfile_close
     end if
  end select

  call clock_stop(32)
!-------------------------------------------------------------------------
!! gnuplot file(s) - optional

  call clock_start(33,'gnuplot time')

  select case(param%control)
  case default
     if(plot%gnu) then
        call gfile_init(trim(file%gnu),'file header',nprint)
        call bigobj_writeg(bigobj,'selector',nprint)
        call gfile_close
     end if
  end select

  call clock_stop(33)
!-------------------------------------------------------------------------
!! output file

  call clock_start(34,'outfile_init time')
  call bigobj_initwrite(fileroot)
  call bigobj_write(bigobj)
  call bigobj_closewrite()
  call clock_stop(34)
!-------------------------------------------------------------------------
!! cleanup and closedown

  call bigobj_delete(bigobj)

  call clock_stop(1)
  call clock_summary

  call log_close
  call clock_delete
!-------------------------------------------------------------------------

end program p_qprog
```

Appendix B

doxygen Documentation for Sample Program

This is now a separate document, ref [5], available on request, together with the source code.

Acknowledgement

This work has been part funded by the RCUK Energy Programme under grant number EP/I501045. To obtain further information on the data and models underlying this paper please contact PublicationsManager@ccfe.ac.uk. The source code for the QPROG program and its doxygen documentation are also available on request.