

FORTRAN Coding Standard in OPA System

This document defines a set of FORTRAN specifications, rules, and recommendations for the coding of OPA system. It is based on *PRISM system specification handbook*¹ since OPA is and should stay one of the PRISM components. The purpose is to provide a framework that enables users to easily understand or modify code, or to port it to new computational environments. In addition, it is hoped that adherence to these guidelines will facilitate the exchange and incorporation of new packages and parameterizations between PRISM components. Other work that influenced the development of this standard are *Community Climate System Model, Software Developer's Guide*², and *Report on Column Physics Standards*³, *European Standards For Writing and Documenting Exchangeable FORTRAN 90 Code*⁴.

Language

- **FORTRAN 90/95 standard:** The OPA system software adheres to the FORTRAN 95 language standard and not rely on any specific language or vendor extension, although some restraint has been exercised in using the full power of FORTRAN 90. The purpose is to enhance portability, and to allow use of the many desirable new features of the language.
- **English language:** Owing to the international user community, all naming of variables, modules, functions, and subroutines as well as all comments are to be written in English.
- **Physical units:** physical units are MKS. The temperature is expressed in degree Celsius, except in bulk formulae and part of LIM sea-ice model. (do we have to change that?) → action?)
- **Free form source:** Free-form source is used. The F90/95 standard allows up to 132 characters. PRISM recommendation is to use a self-imposed limit of about 80 in order to enhance readability and make life easier for those with bad eyesight, who wish to make overheads of source code, or print source files with two columns per page. We do not recommend a strictly application of this rule: if the readability is improved with more than 80 characters, do not worry about the printing of source files. Nevertheless, multi-line comments that extend to column 100 would be unacceptable.
- **Bounds checking:** As all PRISM components, OPA must be able to run when a compile-time and/or run-time array bounds checking option is enabled. Thus, constructs of the following form are disallowed:
`REAL(wp) :: arr(1)`
where "arr" is an input argument into which the user wishes to index beyond 1. Use of the (*) construct in array dimensioning to circumvent this problem is forbidden because it effectively disables array bounds checking. (Still a few in lib_mpp → action!)
- **Inter-procedural code analysis:** Use of a tool to diagnose problems such as array size mismatches, type mismatches, variables which are defined but not used, etc. is strongly encouraged. Flint is one such tool that has proved valuable in this regard. It is not a strict rule that all PRISM codes and packages must be "flint-free", but the developer must be able to provide adequate explanation for why a given coding construct should be retained even

¹ Ref web to be found !!! ??? and URL

² http://www.cesm.ucar.edu/csm/working_groups/Software/dev_guide/dev_guide/

³ <http://nsipp.gsfc.nasa.gov/infra/>

⁴ <http://nsipp.gsfc.nasa.gov/infra/eurorules.html>

though it elicits a complaint from flint. If too many complaints are issued, the diagnostic value of the tool diminishes toward zero. (Never tested with the code → action!)

• **Memory management:** The use of dynamic memory allocation is not discouraged because we realize that there are some situations in which run-time array sizing is desirable. However, this type of memory allocation can cause performance problems on some machines, and some debuggers get confused when trying to diagnose the contents of such variables. Therefore, dynamic memory allocation is allowed only "when necessary". The ability to run a code at a different spatial resolution without recompiling is not considered to be an adequate reason to use dynamically allocated arrays. As consequence, *there is not dynamic memory allocation in OPA9* and you should have very good reason to add some....

The preferable mechanism for dynamic memory allocation is automatic arrays, as opposed to ALLOCATABLE or POINTER arrays for which memory must be explicitly allocated and de-allocated. An example of an automatic array is:

```
SUBROUTINE sub(n)
  REAL :: a(n)
  ...
END SUBROUTINE sub
```

The same routine using an allocatable array would look like:

```
SUBROUTINE sub(n)
  REAL, ALLOCATABLE :: a(:)
  ALLOCATE(a(n))
  ...
  DEALLOCATE(a)
  ...
END SUBROUTINE sub
```

• **Features to be avoided:** In terms of keeping code up to date and easier to maintain the code should always follow the current standards of FORTRAN and ANSI C. We decide to restrict the languages use to the elements, which are *not obsolete* or deleted -- even if they are still available with almost all compilers. In particular, the code must be such that during the compiling phase, no warning appears, so that user can be easily alerted of potential bugs when some appear in their configuration.). (there is still some warning → to be suppressed in a forthcoming release) Examples for FORTRAN 95 are:

- COMMON blocks - use the declaration part of MODULEs instead.
- EQUIVALENCE - use POINTERS or derived data types instead to form data structures. Please try to avoid this anyway as it is usually a source of bugs.
- Assigned and computed GOTOs - use the CASE construct instead.
- Arithmetic IF statements - use the block IF, ELSE, ELSEIF, ENDIF or SELECT CASE construct instead.
- Labeled DO constructs - use unlabeled END DO instead. Nevertheless non-number label can be used for big iterative loop of recursive algorithm.
- I/O routines END and ERR - use IOSTAT instead (the use is somehow restricted due compiler implementation dependent error numbering). (This is not yet done throughout the code → action!)
- FORMAT statements: use character parameters or explicit format- specifiers inside the READ or WRITE statement instead. (This is not yet done throughout the code → action!)
- GOTO and CONTINUE statement - use IF, CASE, DO WHILE, EXIT or CYCLE statements or a contained SUBROUTINE instead. If you feel you cannot avoid a GOTO and or CONTINUE statement, then add a clear comment to explain what is going on and why you need to use GOTO. (Still a few GOTO and CONTINUE → action!)
- PAUSE – just never use it.
- ENTRY statements: a subprogram must only have one entry point.

- RETURN – it is obsolete and so not necessary at the end of program units. (still exist in lib_mpp → action!)
- STATEMENT FUNCTION — this feature has become obsolescent. It's a pity since there is no true equivalence of it in FORTRAN 90. In OPA, most of the statement functions have been replaced by cpp macro which basically do the same thing as the FORTRAN 77 statement function but in a far less elegant way...
- Fixed source form – use free form instead
- Avoid functions with side effects. There are good reasons to avoid this. First, the code is easier to understand, if you can rely on the rule that functions don't change their arguments, second, some compilers generate more efficient code for PURE (in FORTRAN 95 there are the attributes PURE and ELEMENTAL) functions, because they can store the arguments in different places. This is especially important on massive parallel and as well on vector machines.
- DATA and BLOCK DATA - initialisers in FORTRAN 95 give this functionality. (This is not yet done throughout the code → action!)

Style and Layout Rules: generalities

Well thought out and structured code will make all the tasks of debugging, optimisation, parallelisation, etc. far easier and also those of porting, maintenance and adding new functionality. *a small extra effort at the start reaps rewards in the long term.* One can never underestimate the importance of good software practice when writing code. Not only will this reduce the possibility of bugs being introduced but it may well prolong the life of the code and make it more comprehensible and therefore likely to be more widely used.

Good coding standards are essential when working on larger projects like OPA with more than one person involved, when other people will have to use the code and/or add development to it. We don't have the pretension of having chosen the best style (if it exists!), we have just adopted what we think is good. The key idea here is to *be consistent* in term of coding rules *throughout the code*.

On the whole, the main contributor to code-level documentation and readability is not comments, but rather good programming style. Style includes good program structure, use of straightforward and easily understandable approaches, good variable names, good routine names, use of named constants instead of literals, clear layout, and minimisation of control-flow and data-structure complexity.

When a piece of code is written and you have checked informally that everything looks reasonable clean up the leftovers from the development process.

- Check that all parameters are used and that all input and output data is accounted for;
- Check layout and style, i.e. make sure there is white space to clarify the logical structure of the routine, expressions and parameter list;
- Check the documentation, including comments, and make sure that it is up to date and consistent.

This might be an iterative process. Try to understand each line of your code. Avoid the rush to completion and do not compile it until you have convinced yourself that the routine is likely to be correct. Otherwise you might get caught in a hacker-minded attitude leading to hasty, error-prone changes that take more time to debug in the long run.

Documentation - Comments

Documentation consists of putting information both inside and outside the source code. On large, formal projects, most of the documentation is outside the source code. However because the internal documentation is most closely associated with the code, it is the part most likely to remain correct as the code is modified. The thing to be really careful with is that misleading comments are even worse than having no comments at all. So, please report any misleading comments you found. Also try to avoid having superfluous comments such as:

```
A = A + 1    ! Increment A by one.
```

This is not adding anything that is not immediately obvious. A person looking through your code should only have to scan your comments to be able to get a good idea of what the code does and where to look for any special activity. A comment should always precede the code it describes or should be put at the end of the same line code. If necessary spread the comment over several lines – comments that are wrapped by printers on to new lines are not helpful.

Good comments do not just repeat verbally what is happening in the code or just explain it. They clarify its intent. They should explain at a higher level of abstraction what the code or what you are trying to do. In addition you should comment:

- Everything that gets around an error or an undocumented feature in a language or environment.
- Any violations of good programming style should be justified. This will ensure that any well-intentioned programmer does not break your code by changing the source to implement a “better” style.
- Comments are especially helpful at the end of long or nested loops. A comment can be used at the end of each loop to show which loop has ended and thus ease the clarity of the code.
- The lines before a control structure, e.g. IF, CASE, loop, etc. are a natural spot to comment and explain what these constructs are about to do.

Naming conventions

Another point, which may seem rather trivial, is the actual naming convention used for the routines. In general, a routine should have a clear, unambiguous name, describing what the routine does. For example:

- Using a verb followed by the object it operates on for a procedure name immediately conveys what the routine is actually doing: e.g. `print_report()`, `read_data()`, etc.
- As a function usually returns a value, the function should be named so as to relate some information about the value it is expected to return, e.g. `cos()`, `next_task_id()`;
- Ambiguous verbs that could cover several meanings should be avoided, e.g. `deal_with_input()`, `handle_calculation()`;
- Never use names that may clash with the operating system or language intrinsics, e.g. `read()` or `access()`.

With a little bit of thought you can, from the very start, add to the clarity of the code by choosing your function names, and this is equally applicable to your variables, in a meaningful way and not haphazardly to tickle your fancy.

- Use meaningful English variable and constant names. Choose your variable names so that they are readable, memorable and descriptive. Nonetheless, try to avoid names that are too long, e.g. `NumberOfIterationsOfTheMainLoop` which is just asking for repetitive strain injury. The use of upper case letter in a variable name is also prohibited. One letter variable is prohibited. **(This is not yet done throughout the code → action!)**. When possible, use the same naming convention as for modules and routines, i.e. long name should be read by block of 3 letters. Always use the DOCTOR norme convention for name Example:

Blahblahblah....

- *Be consistent in your naming convention within a program.* When you use an enumerated type, you can ensure that it is immediately clear that members of the same type all belong to the same group by using a common group prefix or suffix, e.g. COLOUR Red, COLOUR Green, COLOUR Blue are all of the type COLOUR. Give boolean variables names that imply True or False, this works fine for Done, but not for Status. When naming constants, name the abstract entity of what the constant represents rather than the number the constant refers to. In general, the degree of formality in naming conventions depends on the number of people working on a code its size and its expected life span.

Naming Conventions.

The purpose of the naming conventions is to use prefix letters to classify model variables. These conventions allow to know easily the variable type and to identify them rapidly:

type status	integer	real	logical	character	double precision	complex
global or common	m n <i>but not nam</i>	a b e f g h o q to x <i>but not fs</i>	l <i>but not</i> lk ln lp ld ll	c <i>but not</i> cp cd cl	d <i>but not</i> dp dd dl	y <i>but not</i> yp yd yl
dummy argument	k <i>but not</i> kf	p <i>but not</i> pp pf	ld	cd	dd	yd
local variable	i	z	ll	cl	cd	yl
loop control	j <i>but not</i> jp					
parameter	jp	pp	lp	cp	dp	yp
CPP keys		key_	lk_			
namelist	??	??	ln_			
cpp macro		<i>fs used substitution</i>				

To be done:

Find a letter for real and integer namelist parameter....

- lpwestobc → lp_obc_west
- lpeastobc → lp_obc_east
- lpnorthobc → lp_obc_north
- lpsouthobc → lp_obc_south

Pre-processor:

Where the use of a language pre-processor is required, it will be the C Pre-Processor (CPP). CPP is available on any UNIX platform, and many FORTRAN compilers have the ability to run CPP automatically as part of the compilation process. Nevertheless, use and abuse of CPP to defined numerical and physical option is not recommended. Caution, the *#ifdef* abbreviate form must not be used. It may have problems with some unix scripts. Use the standard *#if defined* form.

One of the most noticeable changes between OPA 8.2 and OPA 9.0 is that OPA 9.0 has a reduced number of CPP pre-processor options. The proliferation of *#if defined* in OPA 8.2 presented the user with a complex menagerie of logical structures (e.g., multiple and nested *#if defined*) to wade through in order to reveal the utilized FORTRAN code. A rough count of the CPP keys available in OPA 8.2 came to more than 100 options, with multiple permutations allowed! Furthermore, CPP options cannot be checked at compile time for typos. Locally, we experienced numerous occasions where an CPP keys was misspelled, yet the model continued to run using an undesired piece of code.

➔ ajouter test dans le script des cle pour misspelling staff. Créer un key_list_ref pour signaler tout ajout non standard...

We have done a few things to minimize the cpp keys. Firstly, we add in the compile script two checks: (1) the CPP keys used are compared to the actual list of existing keys (the compilation will not start when trying to specify an non-existing key) ; (2) if a change occurs in the CPP keys used in a given experiment, the whole compile phase is redone. **These 2 checks are not perfect! They have to be improved in forthcoming release.** Secondly, we have suppressed all the CPP option that what were not associated with an additional memory requirement. We have associated all the CPP keys with a logical parameter : key_optionname ⇔ lk_optionname, and we introduced FORTRAN if-tests in many places where *#if defined* formerly lived. Doing so allows for the FORTRAN compiler to detect typos, thus enhancing the quality control aspects of the model. Thirdly, where large chunks of physical parameterization code could be isolated, we made FORTRAN modules which have the same name as the CPP key (i.e. optionname.F90). These modules are the only place where a *#if defined* command appears which select either the whole FORTRAN code or a dummy module. For example, the TKE vertical physics, the module name is zdf_tke.F90, the CPP key is key_zdf_tke and the associated logical is lk_zdf_tke. The module looks like this:

```
MODULE zdf_tke
  !!=====
  !!                               *** MODULE zdf_tke ***
  !!=====
  #if defined key_zdf_tke  ||  defined key_esopa
  !!-----
  !!   'key_zdf_tke'                                     TKE scheme
  !!-----
```

expliquer le pourquoi de ce commentaire!

```
!! * Modules used
USE oce           ! ocean dynamics and active tracers
...

IMPLICIT NONE
PRIVATE

!! * Routine accessibility
PUBLIC zdf_tke   ! routine called by step.F90

!! * Share Module variables
LOGICAL, PUBLIC, PARAMETER :: lk_zdf_tke = .TRUE.   !: TKE vertical mixing flag
```

```

LOGICAL, PUBLIC ::          & !!! ** tke namelist (namtke) **
  ln_rstke = .FALSE.        !: =T restart with tke from a run without tke
  !                          ! a none zero initial value for en
REAL(wp), PUBLIC, DIMENSION(jpi,jpj,jpk) :: &
  en                        !: now turbulent kinetic energy

!! * Module variables
INTEGER ::                 &
  ...
REAL(wp) ::                & !!! ** tke namelist (namtke) **
  ...
!!-----
!!   OPA 9.0 , LODYC-IPSL   (2003)
!!-----

```

CONTAINS

```

SUBROUTINE zdf_tke ( kt )
All the Fortran code use when key_zdf_tke is defined...
END SUBROUTINE zdf_tke

```

#else

```

!!-----
!!   Dummy module :                               NO TKE scheme
!!-----

```

```

LOGICAL, PUBLIC, PARAMETER :: lk_zdf_tke = .FALSE. ! TKE flag

```

CONTAINS

```

SUBROUTINE zdf_tke( kt )          ! Empty routine
  WRITE(*,*) 'zdf_tke: You should never see this print! Error?', kt
END SUBROUTINE zdf_tke

```

#endif

```

!!=====
END MODULE zdf_tke

```

This approach minimizes the number of #if defined commands that appears in the source code. In the ideal world, the #if defined key_optionname should appear only once in the whole code. **We are far from having reached this stage, about half of the way have been done right now → action in forth coming release!**

The “dummy” routines allow for all major physics options to always be called from within the main time stepping loop for the tracer and velocity equations if lk_keyname=true.

Each module of physics has a different name, so that by defining the key_esopa, all modules of physics are called. This allows a fast check not only of the whole code compilation but also of its execution. Obviously, the results are non physics as for example the tracers advection trend will be added several time (as many time as there is different advective schemes).

Use if possible a logical having the same type of name as the key so that the portion of the code can be compile in any cases.

Finally, un mot sur les key_config et le cp_cfg et jp_cfg

Program units:

- Use meaningful English module and subroutine names using the “3 letters” naming convention.
- Always use the same name in lower case for a module file and the inside module. This simplified the scripts used to compile the system.
- Put one and only one module in a file.
- Always name program units and always use the corresponding END PROGRAM; END SUBROUTINE; END INTERFACE; END MODULE; etc. construct, again specifying the name of the program unit. This helps finding the end of the current

program entity.

- **Error conditions:** When an error condition occurs inside a package, a message describing what went wrong will be printed. The name of the routine in which the error occurred must be included. It is acceptable to terminate execution within a package, but the developer may instead wish to return an error flag through the argument list. If the user wishes to terminate execution within the package, a generic PRISM Coupling Software termination routine "endrun" should be called instead of issuing a FORTRAN "stop". Otherwise a message-passing version of the model could hang. Note that this is an exception to the package-coding rule that "A package should refer only to its own modules and subprograms and to those intrinsic functions included in the FORTRAN 95 standard".

File format

All the routines must be embedded in a module. Embedding multiple routines within a single file and/or module is allowed, encouraged in fact, if any of three conditions hold. First, if routine A and only routine A is routine B, then the two routines may be included in the same file. This construct has the advantage that inlining B into A is often much easier for compilers if both A and B are in the same file. Practical experience with many compilers has shown that inlining when A and B are in different files often is too complicated for most people to consider worthwhile investigating.

The second condition in which it is desirable to put multiple routines in a single file is when they are "CONTAIN"ed in a module for the purpose of providing an implicit interface block. This type of construct is strongly encouraged, as it allows the compiler to perform argument consistency checking across routine boundaries. An example is:

file 1: driver.f90

```
MODULE driver
  USE mod1
CONTAINS
  SUBROUTINE driver_init
    REAL :: X, Y ...
    CALL sub1(X,Y)
    CALL sub2(Y) ...
  END SUBROUTINE driver_init
END MODULE driver
```

file 2: mod1.f90

```
MODULE mod1
  IMPLICIT NONE
  PRIVATE
  PUBLIC sub1, sub2
CONTAINS
  SUBROUTINE sub1(A,B)
    ...
  END SUBROUTINE sub1

  SUBROUTINE sub2(A) ...
  END SUBROUTINE sub2
END MODULE mod1
```

The compiler automatically checks the number, type, and dimensions of the arguments passed to sub1 and sub2.

The final reason to store multiple routines and their data in a single module is that the scope of the data defined in the module can be limited to only the routines which are also in the module. This is accomplished with the "private" clause.

If none of the above conditions hold, it is not acceptable to simply glue together a bunch of functions or subroutines in a single file.

- **Module names:** Modules MUST be named the same as the file in which they reside. The reason to enforce this as a hard rule has to do with the fact that dependency rules used by "make" programs are based on file names. For example, if routine A "USE"s module B, then "make" must be told of the dependency relation which requires B to be compiled before A. If one can assume that module B resides in file B.o, building a tool to generate this dependency rule (e.g. A.o: B.o) is quite simple. Put another way, it is difficult (to say nothing of CPU-intensive) to search an entire source tree to find the file in which module B resides for each routine or module which "USE"s B.

Note that by implication multiple modules are not allowed in a single file.

The use of common blocks is deprecated in FORTRAN 95 and their continued use in the PRISM component is strongly discouraged. Modules are a better way to declare static data. Among the advantages of modules is the ability to freely mix data of various types, and to limit access to contained variables through use of the ONLY and PRIVATE clauses.

- **I/O error conditions:** I/O statements which need to check an error condition will use the "iostat=<integer variable>" construct instead of the outmoded end= and err=. Note that a 0 value means success, a positive value means an error has occurred, and a negative value means the end of record or end of file was encountered.

- **namelist input.** The point behind this rule is that packages should not have to know details of the surrounding model data structures, or the names of variables outside of the package. A notable exception to this rule is model resolution parameters. The reason for the exception is to allow compile-time array sizing inside the package. This is often important for efficiency.

Variables declaration:

- **Implicit None:** All subroutines and functions will include an "implicit none" statement. Thus all variables must be explicitly typed. It also allows the compiler to detect typographical errors in variable names. For MODULEs, one IMPLICIT NONE statement in the modules definition section is sufficient.

Improper data initialisation is another common source of errors. A variable could contain an initial value you did not expect. This can happen for several reasons, e.g. the variable has never been assigned a value, its value is outdated, memory has been allocated for a pointer but you have forgotten to initialise the variable pointed to. Some compilers initialise variables to zero but when you port your code to another computer that does not do this previously working code will no longer work this can take some time to diagnose and longer to resolve. To avoid such mishaps initialise variables as close as possible to where they are first used.

- **Private / public :** **blah blah blah**

- **Package attribute:** Modules variables and routines should be encapsulated by using the PRIVATE attribute. What shall be used outside the module can be declared PUBLIC instead. Use USE with the ONLY attribute to specify which of the variables, type definitions etc. defined in a module are to be made available to the using routine. Of course you don't need to add the ONLY attribute if you include the complete module or almost all of its public declarations.

Thus, any variable declared real(dp) will be of sufficient size to maintain 12 decimal digits in their mantissa. Likewise, integer variables declared integer(i8) will be able to represent an integer of at least 13 decimal digits.

• **Constants and magic numbers:** Magic numbers should be avoided. Physical constants (e.g. pi, gas constants) must NEVER be hardwired into the executable portion of a code. Instead, a mnemonically named variable or parameter should be set to the appropriate value, probably in the setup routine for the package. We realize that many parameterizations rely on empirically derived constants or fudge factors, which are not easy to name. In these cases it is not forbidden to leave such factors coded as magic numbers buried in executable code, but comments should be included referring to the source of the empirical formula.

Hard-coded numbers should never be passed through argument lists. One good reason for this rule is that a compiler flag, which defines a default precision for constants, cannot be guaranteed. FORTRAN 95 allows specification of the precision of constants through the "_" compile-time operator (e.g. 3.14_dp or 365_i8). So if you insist on passing a constant through an argument list, you must also include a precision specification in the calling routine. If this is not done, a called routine that declares the resulting dummy argument as, say, real(dp) or 8 bytes, will produce erroneous results if the default floating point precision is 4 byte.

The PRISM software will distribute some physical variables/constants, which are shared between several PRISM model components.

Interface to other languages

• **Parameter declaration:** Variables used as constants should be declared with attribute PARAMETER and used always without copying to local variables. This prevents from using different values for the same constant or changing them accidentally. (Not yet the case throughout the code, cf phycst)

- Usage of the DIMENSION statement or attribute is required⁵ in declaration statements.
- Attribute SAVE is banned. This simplifies the use of AGRIF software. Since all the subroutine are embedded into a module, the variables which value have to be preserved between 2 calls can be declared at the module level.
- The "::" notation is quite useful to show that this program unit declaration part is written in standard FORTRAN syntax, even if there are no attributes to clarify the declaration section. Always use the notation <blank>::<three blanks> to improve readability.
- Declare the length of a character variable using the CHARACTER (len=xxx) syntax - the len specifier is important because it is possible to have several kinds for characters (e.g. Unicode using two bytes per character, or there might be a different kind for Japanese e.g.. NEC).
- Never use a FORTRAN keyword as a routine or variable name.
- For all global data (in contrast to module data, that is all data that can be access by other module) must be accompanied with a comment field expressed with a "!" characters followed by the comment text all on the same line as the declaration itself.

For example:

```
REAL(wp), DIMENSION(jpi,jpj,jpk) :: ua      &  !: i-horizontal velocity  
(m/s)
```

This allows a easy research of where and how a variable is declared using the unix command: "grep var *90 |grep !:". (Not yet the case throughout the code)

• **Precision:** Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size. The f95 KIND feature should be used instead.

In order to improve portability between 32 and 64 bit platforms, it is necessary to make use of kinds by providing a specific module declaring the "kind definitions" to obtain the

⁵ This is different from the PRISM standard where is it not necessary. PRISM recommends to declare the shape and size of arrays inside parentheses after the variable name on the declaration statement.

required numerical precision and range as well as size of INTEGER. It should be noted that constants need to have attached a `_kindvalue` to have the according size.

Example, with `dp` as a real kind with 12 significant digits (usually double precision, 8 byte), and `wp` a working precision:

```
USE my_kind, ONLY: dp, wp
IMPLICIT NONE
! Declaration of a constant
REAL(dp), PARAMETER :: pi = 3.14159265358979323846_dp
! Declaration of a 3d field
REAL(wp), POINTER :: geopotential(:,:,:)
! Declaration of a local variable
REAL(wp) :: a
...
a = 4.e0_wp
...
```

Possible kinds in a proposed `my_kind`:

Note:

The bit sizes given are not mandatory. The kind value is selected regarding a given precision, which results on current systems in this bit sizes. This may change in future. A portable way to build a module providing several kinds is given in the following example:

<<<tab à reprendre et inserrer>>>>

```
MODULE my_kind

  IMPLICIT NONE

  ! Number model from which the
  !   SELECTED_*_KIND are requested:
  !
  !   4 byte REAL    8 byte REAL
  !   CRAY:    - precision = 13
  !             exponent = 2465
  !   IEEE:  precision = 6    precision = 15
  !             exponent= 37  exponent = 307
  !
  ! Most likely this are the only possible models.

  ! Floating-point section
  INTEGER, PARAMETER :: sp = SELECTED_REAL_KIND(6,37)
  INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(12,307)
  INTEGER, PARAMETER :: wp = dp    ! working precision

  ! Integer section
  INTEGER, PARAMETER :: i4 = SELECTED_INT_KIND(9)
  INTEGER, PARAMETER :: i8 = SELECTED_INT_KIND(14)
  ! Pointer section, the cp type should provide an integer
  ! with a size for transporting a C pointer through Fortran
#ifdef key_cp4
  INTEGER, PARAMETER :: cp = i4
#else
  INTEGER, PARAMETER :: cp = i8
#endif

END MODULE my_kind
```

• **Intent:** All dummy arguments must include the `INTENT` clause in their declaration. This is extremely valuable to someone reading the code, and can be checked by compilers. A common mistake is to put the wrong type of variable in a routine call. So, develop the habit of checking types of arguments in parameter lists. Many modern compilers, especially

FORTRAN 90, check for consistent use within a file or across files using inter-procedural analysis. FORTRAN 90 will also flag up errors at link time if there are explicit or implicit interfaces. Instead of ordering parameters randomly or alphabetically, list all parameters that are input-only first, input-and output second, and output-only third. You can ensure that parameters are used as intended by using the FORTRAN 90 INTENT attribute with the parameters passed down to subroutines. It is dangerous to use the parameters passed to a routine as working variables. Use local variables instead. An example is:

```
MODULE sub1
  IMPLICIT NONE
CONTAINS
  SUBROUTINE sub_1( px, py, pz )
    REAL(wp), INTENT( in  ) :: px    ! short description
    REAL(wp), INTENT( out ) :: py    ! short description
    REAL(wp), INTENT(inout) :: pz    ! short description
    py = px
    pz = pz + px
  END SUBROUTINE sub_1
END MODULE sub1
```

- **Interface blocks:** Explicit interface blocks are required between routines if optional or keyword arguments are to be used. They also allow the compiler to check that the type, shape and number of arguments specified in the CALL are the same as those specified in the subprogram itself. FORTRAN 95 compilers can automatically provide explicit interface blocks for routines contained in a module.

Content rules

Layout

There is a considerable difference that can be made in the readability and the correctness of the interpretation of a piece of code just by its layout.

An enormous difference can be made purely by the use of white space and the judicious use of brackets. A good visual layout shows the logical structure of a program. Typically, programmers use indentation and other white space to show the logical structure:

- White spaces allow you to ensure that related statements are grouped together;
- Unrelated statements can be separated from each other by inserting blank lines. You can also use blank lines to divide groups of related statements into paragraphs, to separate routines from each other, and to highlight comments;
- Align elements that belong together, such as equal signs in groups of related assignments. Do not align elements if they do not belong together, that is only misleading; Align data declarations. Use only one data per line for important variables. It is easier to put a comment next to each declaration and modify each declaration accordingly as it is self-contained. Avoid aligning similar items under the same type name (e.g. gathering all integers under one integer-declaration).
- You can use declarations to judiciously group together connected variables; You should indent statements under those statement to which they are logically subordinate;
- Indent a comment to lie with its corresponding code. Otherwise, you will break up the logical structure which was meant to be stressed by indentation. Set off each comment with at least one blank line. Also avoid extremely long comments that will be wrapped when printed as these too will clutter the program flow. Do not be afraid to spread a comment over multiple lines;
- For complicated expressions, put separate conditions on separate lines. Do not be afraid

- to use brackets to remove any ambiguity. Be as explicit as you can;
- Separate each routine from other routines in the same file or module with at least two blank lines to produce a visual difference between single blank lines that are part of a routine.

Sometimes a statement has to be broken across lines, e.g. because it is longer than programming standards allow. *Make it obvious that the part of the statement on the first line is only part of a statement. The easiest way to do that is to break up the statement so that the part on the first line is blatantly incorrect syntactically if it stands alone (e.g. end the first line with '.and.', '+', ','). Nonetheless, try to keep things together that belong together, such as array references and arguments to a routine.*

Declare arguments separately and in the same order in which they are listed in the subroutine's argument list. Do not mix declarations of arguments, shared variables (PUBLIC) with declarations of local variables.

Avoid over emphasising comments within modules. If every routine and comment is marked with a line of asterisks, so many things are highlighted that nothing is really clear. In addition, they are more difficult to maintain. If you need other separators than blank lines, develop a hierarchy of characters (from densest to lightest).

- **Argument comments:** Input arguments and local variables will be declared 1 per line, with a comment field expressed with a "!" character followed by the comment text all on the same line as the declaration. Multiple comment lines describing a single variable are acceptable when necessary. Variables of a like function and type may be grouped together on a single line. For example:

```
INTEGER ::  ji, jj, jk  ! dummy loop indices
```

- **Commenting style:** Short comments may be included on the same line as executable code using the "!" character preceded by at least three blank space and followed by the description. If the description is too long, and additional line can be used with proper alignment. For example:

```
zx = zx *zzy  ! Describe what is going on and if it is
!             ! too long use another '!' for proper
!             ! alignment with automatic indentation
```

More in-depth comments should be written in the form:

```
! Describe what is going on
```

or

```
!             !-----!
!             ! Describe what is going on !
!             !-----!
```

Key features of this style are 1) it starts with a "!" in the column required for proper automatic indentation; 2) The text starts after the '!'; and 3) the text is offset above and below by a blank line or a content line built for underlying.

- **FORTRAN keywords and intrinsic function or routine:** FORTRAN keywords and intrinsic function or routine are written in upper case, the remaining code in lower case. This participates to a better readability of a code. Note that for historical reason, we use END DO instead of ENDDO statment, but ENDDIF and not END IF statements. Please respect these choices.

- **Loops:** Loops should be structured with the DO - END DO construct as opposed to numbered loops. In case of long loop, a self-descriptive label can be used (i.e. not just a

number!).

- **Continuation lines:** Continuation lines can be used on multi-dimensional array declarations that take up many columns. For example:

```
REAL(wp), DIMENSION(jplond,jplev), INTENT(in) :: &
  array1, & ! array1 is blah blah blah
  array2 ! array2 is blah blah blah
  ! ! and even longer blahblah
```

Note that the ISO/IEC 1539-1:1997 FORTRAN 95 standard defines a limit of 39 continuation lines.

Code lines, which are continuation lines of assignment statements, must begin to the right of the column of the assignment operator. Due to the possibility of automatic indentation in some editor (emacs for example), use a ‘&’ as first character of the continuing lines to maintain the alignment. For example,

```
arr1 = arr2 + arr3 * arr4 + ... &
      + arr7 * arr8 + arr9 * arr10
```

becomes with automatic indentation

```
arr1 = arr2 + arr3 * arr4 + ... &
      + arr7 * arr8 + arr9 * arr10
```

while the expression below remains unchanged with automatic indentation

```
arr1 = arr2 + arr3 * arr4 + ... &
      & + arr7 * arr8 + arr9 * arr10
```

Similarly, continuation lines of subroutine calls and dummy argument lists of subroutine declarations must have the arguments aligned to the right of the "(" character. Examples of each of these constructs are:

```
CALL sub76( px, py, pz, pw, pa, &
           & pb, pc, pd, pe )
```

```
SUBROUTINE sub76( px, py, pz, pw, pa, &
                 & pb, pc, pd, pe )
END SUBROUTINE sub76
```

- **Indentation:** Code as well as comment lines within loops, if-blocks, continuation lines, MODULE or SUBROUTINE statements will be indented 3 characters for readability. (except for CONTAINS that remains at first character)

```
MODULE mod1
  REAL(wp) xx
CONTAINS
  SUBROUTINE sub76( px, py, pz, pw, pa, &
                  & pb, pc, pd, pe )
    <instruction>
  END SUBROUTINE sub76
END MODULE mod1
```

- **Spacing conventions**

- Use blank space, in the horizontal and vertical, to improve readability. In particular try to align related code into columns. For example, instead of:

```
! Initialize Variables
zx=1.e0_wp
meaningful_name=3.e0_wp
SillyName=2.e0_wp
write:
! Initialize variables
zx          = 1.e0_wp
meaningful_name = 3.e0_wp
silly_name  = 2.e0_wp
```

- Use a blank space after a comma (i.e. "a, b, c") except for the indices of an array.
- Use a blank space before and after all operator (+, -, *, =, /), including logical operators (.AND., .OR., ==, /=, etc.).

- Use only one blank space put on the interior side of a bracket, except for brackets related to arrays where no space is needed.
- Use a blank space before and after all operator
- Do not use tab characters (an extension!) in your code: this will ensure that the code looks as intended when ported.
- **Formatted I/O:** Avoid separating the information to be output from the formatting information on how to output it on I/O statements.
- **Argument list format:** Routines with large argument lists will contain 5 variables per line. This applies both to the calling routine and the dummy argument list in the routine being called. The purpose is to simplify matching up the arguments between caller and callee. In rare instances in which 5 variables will not fit on a single line, a number smaller than 5 may be used. But the per-line number must remain consistent between caller and called. An example is:

```
CALL linemsbc ( u3 (i1,1,1,j,n3m1), v3(i1,1,1,j,n3m1), &
& t3 (i1,1,1,j,n3m1), q3(i1,1,1,j,n3m1), &
& qfcst(i1,1,m,j) , xxx )

SUBROUTINE linemsbc ( pu , v , &
& pt , q , &
& pqfcst, xxx )
```

- **Array arguments:** Do not implicitly change the shape of an array when passing it into a subroutine. Although actually forbidden in the FORTRAN 77 standard it was very common practice to pass n dimensional arrays into a subroutine where they would, say, be treated as a one dimensional array. Though officially banned in FORTRAN 95, this practice is still possible with external routines for which no interface block is supplied. The danger of this method is that it makes certain assumptions about how the data is stored.

- **Operators:** Use of the operators <, >, <=, >=, ==, /= is strongly recommended instead of their deprecated counterparts, lt., .gt., .le., .ge., .eq., and .ne. The motivation is readability. In general use the notation: <Blank><Operator><Blank>

- **Array syntax:** Array notation should be used whenever possible. This should help optimization regardless what machine architecture is used (at least in theory) and will reduce the number of lines of code required. To improve readability the array shape must be shown in brackets, e.g.:

```
onedarraya(:) = onedarrayb(:) + onedarrayc(:)
twodarray(:, :) = scalar * anothertwodarray(:, :)
```

When accessing sections of arrays, for example in finite difference equations, do so by using the triplet notation on the full array, e.g.:

```
twodarray(:,2:len2) = scalar &
& * ( twodarray2(:,1:len2-1) &
& - twodarray2(:,2:len2) ) )
```

j'aime pas... le dire!!!

Note: In order to improve readability of long, complicated loops, explicitly indexed loops should be preferred. In general when using this syntax, the order of the loops indices should reflect the following scheme: (best usage of data locality).

```
DO jk = 1, jpk
  DO jj = 1, jpj
    DO ji = 1, jpi
      array(ji,jj,jk) = ...
    END DO
  END DO
END DO
```



```

!! ** Purpose :   Brief description of the routine
!!
!! ** Method  :   description of the methodology used to achieve the
!!                objectives of the routine. Be as clear as possible!
!!
!! ** Action   : - first action (share memory array/variable modified
!!                in this routine
!!                - second action .....
!!                - .....
!!
!! References :
!! Give references if exist otherwise suppress these lines
!!
!! History :
!! 9.0 ! 03-08 (Author names) Original code
!!      ! 02-08 (Author names) brief description of modifications
!!-----
!! * Modules used   (module used only in this subroutine)
USE toto_module           ! description of the module

!! * arguments
INTEGER, INTENT( in ) :: &
    kt                    ! describe it!!!

!! * local declarations
INTEGER ::   ji, jj, jk   ! dummy loop arguments
INTEGER ::   &
    itoto, itata,        & ! temporary integers
    ititi                ! please do not forget the DOCTOR rule:
    !                    ! local integer: name start with i
REAL(wp) ::   &
    zmlmin, zbbrau,      & ! temporary scalars
    zfact1, zfact2, zfact3, & ! " "
    zbn2, zesurf,        & ! local scalar: name start with z
    zemxl                !
REAL(wp), DIMENSION(jpi,jpk) :: &
    ztoto                ! temporary workspace
!!-----

IF( kt == nit000 ) CALL zdf_tke_init   ! Initialization (first time-step only)

! Local constant initialization
zmlmin = 1.e-8
zbbrau = .5 * ebb / rau0
zfact1 = -.5 * rdt * efave
zfact2 = 1.5 * rdt * ediss
zfact3 = 0.5 * rdt * ediss

SELECT CASE ( npdl )

CASE ( 0 )           ! describe case 1
DO jk = 2, jpkml
DO jj = 2, jpjml
DO ji = fs_2, fs_jpim1 ! vector opt.
    avmv(ji,jj,jk) = ....
END DO
END DO
END DO

CASE ( 1 )           ! describe case 2
DO jk = 2, jpkml
DO jj = 2, jpjml
DO ji = fs_2, fs_jpim1 ! vector opt.
    avmv(ji,jj,jk) = ...
END DO
END DO
END DO

```

```

END SELECT

! Lateral boundary conditions (avmu) (unchanged sign)
CALL mpp_lnk( avmu, 'U', 1. )

END SUBROUTINE exa_mpl

# endif

SUBROUTINE exa_mpl_init
!!-----
!!          ***  ROUTINE exa_mpl_init  ***
!!
!! ** Purpose :   initialization of ....
!!
!! ** Method  :   blah blah blah ...
!!
!! ** input   :   Namelist namexa
!!
!! ** Action  :   ...
!!
!! history :
!!  9.0 ! 03-08 (Autor Names) Original code
!!-----
!! * local declarations
INTEGER ::   ji, jj, jk, jit   ! dummy loop indices

NAMELIST/namexa/ exa_v1, exa_v2, nexa_0
!!-----

! Read Namelist namexa : example parameters
REWIND ( numnam )
READ   ( numnam, namexa )

! Control print
IF(lwp) THEN
  WRITE(numout,*)
  WRITE(numout,*) 'exa_mpl_init : example '
  WRITE(numout,*) '~~~~~'
  WRITE(numout,*) '          Namelist namexa : set example parameters'
  WRITE(numout,*) '          brief description   exa_v1 = ', exa_v1
  WRITE(numout,*) '          brief description   exa_v1 = ', exa_v1
  WRITE(numout,*) '          brief description   nexa_0 = ', nexa_0
ENDIF

! Parameter control
IF( lk_toto ) THEN (lk_toto logical key by a description of the CPP key)
IF(lwp) WRITE(numout,cform_err)
IF(lwp) WRITE(numout,*) '          this part and key_toto are incompatible'
nstop = nstop + 1
#endif

! Check nexa_0 values
IF( nexa_0 < 0 ) THEN
  IF(lwp) WRITE(numout,cform_err)
  IF(lwp) WRITE(numout,*) '          bad flag: nmxl is < 0 or > 3 '
  nstop = nstop + 1 explication requise!!
ENDIF

END SUBROUTINE zdf_tke_init

#else
!!-----
!!  Default option :                               Dummy module
!!-----
CONTAINS
  SUBROUTINE exa_mpl          ! Dummy routine
  END SUBROUTINE exa_mpl
#endif

```

```
! =====  
END MODULE exampl
```

References

- Mark Baker (ed.), Cluster Computing White Paper, University of Portsmouth, UK, December 2000 <http://www.dcs.port.ac.uk/mab/tfcc/WhitePaper>
- Kent Beck, Extreme Programming Explained; Embrace Change, Boston, Mass.: Addison-Wesley, 2000
- Barry Cipra, 'Self-tuning' software adapts to its environment, Science, 286 p.35, October 1999
- Steve C. McConnell. Code Complete: A Practical Handbook of Software Construction. Microsoft Press, March 1993.
- David E. Culler et al., Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers Inc., August 1998
- European Cooperation for Space Standardization, Space Engineering: Software, Technical Report ECCS-Q-40A, ECSS, Requirements and Standard Division, April 1999
- European Cooperation for Space Standardization, Space Product Assurance: Software Product Assurance, Technical Report ECCS-Q-80A, ECSS, Requirements and Standard Division, April 1996
- Kernighan, B. W., and R. Pike. The Practice of Programming. Addison Wesley Longman Publishing Co., 1999.
- Michael J. Flynn, Some computer organizations and their effectiveness, IEEE Transactions on Computing, C-21:948-960, 1972
- Rupert W. Ford and Graham D. Riley, The Met Office FLUME Project - Model Coupling Requirements, Manchester Informatics Ltd., The University of Manchester, January 2002
- Ian Foster, The Grid: A New Infrastructure for 21st Century Science, Physics Today, Feb 2002, pp.42 ff
- IEEE Standard Glossary of Software Engineering Terminology, Technical Report IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, December 1990
- Capers Jones, Programming Productivity, New York, New York: McGraw-Hill, 1986
- Bill Joy, Ken Kennedy et al., Report to the President: Information technology research: Investing in our future, Technical Report, PITAC, President's Information Technology Advisory Committee, Feb. 1999 <http://www.hpcc.gov/ac/report/>
- Angelo Mangili et al., PRISM REDOC III.2: Review of Current and Future HPC Architectures, Technical Report , PRISM, June 2002
- Steve McConnell, Code Complete, Redmond, Wash.: Microsoft Press, 1993
- Steve McConnell, Rapid Development, Redmond, Wash.: Microsoft Press, 1996
- Polcher, J., K. Laval, L. Dumenil, J. Lean and P. R. Rowntree (1996) Comparing three land surface schemes used in general circulation models, Journal of Hydrology, 180,373-394,.
- Polcher et al. (1998) A proposal for a general interface between land-surface schemes and general circulation models. Global and Planetary Change, 19: 263-278
- Frank S. Preston, A peta ops era computing analysis, Technical Report CR-1998-207652, March 1998 <http://techreports.larc.nasa.gov/ltrs/>
- James M. Rosinski, David L. Williamson, The Accumulation of Rounding Errors and Port Validation for Global Atmospheric Models, SIAM Journal on Scientific Computation, Vol. 18, No. 2, March 1997, pp.552-564
- Aad J. van der Steen et al., Overview of Recent Supercomputers, 2001 <http://www.top500.org/ORSC/2001/>

A metre ou pas ?

FORTRAN code parallelization issues

- **Parallelization paradigms:** Parallelization should be done with well-documented and commonly accepted paradigms like MPI, OpenMP and Pthreads.
- **Thread-safe:** The coupling software and other modules callable by any component of the PRISM system should be thread-safe to allow for a flexible usage of hierarchical programming models.
- **Scalability:** Each module of the coupling software and the components to be coupled should follow coding strategies for parallelization, which allow for runs on high processor counts. Parallelized model components should be programmed flexible with regard to the number of processors, which can be used for parallel runs. If necessary hierarchical programming techniques should be used to achieve that goal.
- **Data locality:** Regarding to parallelization the coding techniques should be concerned about memory hierarchies of modern computer architectures. A key to parallel efficiency is the locality of the memory references. On a SMP-like architecture one should constrain memory references within a compute node as much as possible rather than excessively accessing remote memory location since networks between compute nodes of these SMP-like architectures are slower than intra-node memory paths in general.
A similar point of view is valid for intra-node memory hierarchies like caches.
- **Data and process placement:** The environment of a parallel run (scripts, configuration files, etc.) and the parallelized components of a PRISM coupled system should allow for a flexible placement or mapping of the various tasks of a coupled simulation onto the computer resources like compute nodes, file systems, etc.