



---

# NEMO coding conventions

---

– version 2 –

NEMO System Team - C. Lévy

July 2010

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview and general conventions</b>	<b>3</b>
<b>3</b>	<b>Architecture</b>	<b>3</b>
<b>4</b>	<b>Style rules</b>	<b>4</b>
4.1	Argument list format . . . . .	4
4.2	Array syntax . . . . .	4
4.3	Case . . . . .	4
4.4	Comments . . . . .	4
4.5	Continuation lines . . . . .	5
4.6	Declaration of arguments and local variables . . . . .	5
4.7	F90 Standard . . . . .	5
4.8	Free-Form Source . . . . .	5
4.9	Indentation . . . . .	5
4.10	Loops . . . . .	6
4.11	Naming Conventions: files . . . . .	6
4.12	Naming Conventions: modules . . . . .	6
4.13	Naming Conventions: variables . . . . .	6
4.14	Operators . . . . .	6
4.15	Pre processor . . . . .	7
<b>5</b>	<b>Content rules</b>	<b>7</b>
5.1	Configurations . . . . .	7
5.2	Constants . . . . .	7
5.3	Declaration for variables and constants . . . . .	7
5.3.1	Rules : . . . . .	7
5.3.2	Implicit None: . . . . .	8
5.3.3	Attributes: . . . . .	8
5.4	Headers . . . . .	8
5.5	Interface blocks . . . . .	8
5.6	I/O Error Conditions . . . . .	8
5.7	PRINT - ASCII output files . . . . .	8
5.8	Precision . . . . .	9
5.9	Structures . . . . .	9
<b>6</b>	<b>Packages coding rules</b>	<b>9</b>
6.1	Bounds checking . . . . .	9
6.2	Communication . . . . .	9
6.3	Error conditions . . . . .	9
6.4	Memory management . . . . .	10
6.5	Optimisation . . . . .	10
6.6	Package attribute: <i>PRIVATE, PUBLIC, USE, ONLY</i> . . . . .	10
6.7	Parallelism: using MPI . . . . .	10
<b>7</b>	<b>Features to be avoided</b>	<b>10</b>

## 1 Introduction

This document describes conventions used in NEMO coding and suggested for its development. The objectives are to offer a guide to all readers of the NEMO code, and to facilitate the work of all the developers, including the validation of their developments, and eventually the implementation of these developments within the NEMO platform.

A first approach of these rules can be found in the code in *NEMO/OPA\_SRC/module\_example* where all the basics coding conventions are illustrated. More details can be found below.

This work is based on the coding conventions in use for the Community Climate System Model <sup>1</sup> the previous version of this document (FORTRAN coding standard in the OPA System) and the expertise of the NEMO System Team which can be contacted for further information (*nemo\_st@locean-ipsl.upmc.fr*) After a general overview below, this document will describe :

- The style rules, i.e. the syntax, appearance and naming conventions chosen to improve readability of the code;
- The content rules, i.e. the conventions to improve the reliability of the different parts of the code;
- The package rules to go a step further by improving the reliability of the whole and interfaces between routines and modules.

## 2 Overview and general conventions

NEMO has different component: ocean dynamics (*OPA\_SRC*), sea-ice (*LIM\_SRC*), ocean biogeochemistry (*TOP\_SRC*), linear-tangent and adjoint of the dynamics (*TAM*) each of them corresponding to a directory. In each directory, one will find some FORTRAN files and/or subdirectories, one per functionality of the code: *BDY* (boundaries), *DIA* (diagnostics), *DOM* (domain), *DYN* (dynamics), *LDF* (lateral diffusion), etc...

All name are chosen to be as self-explanatory as possible, in English, all prefixes are 3 digits.

English is used for all variables names, comments, and documentation.

Physical units are MKS. Only exception for the temperature, which is expressed in degree Celsius, except in bulk formulae and part of LIM sea-ice model where it is in Kelvin. See *DOM/physct.F90* files for conversions.

## 3 Architecture

Within each directory, organisation of files is driven by orthogonality, i.e. one functionality of the code is intended to be in one and only one directory, and one module and all its related routines are in one file. The functional modules are:

- SBC surface module
- IOM management of the I/O
- NST interface to AGRIF (nesting model) for dynamics and biogeochemistry
- OBC, BDY management of structured and unstructured open boundaries
- C1D 1D (vertical) configuration for dynamics, sea-ice and biogeochemistry
- OFF off-line module: passive tracer or biogeochemistry alone
- CFG tutorial and reference configurations
- DOC documentation

For example, the file *domain.F90* contains the module *domain* and all the subroutines related to this module (*dom\_init*, *dom\_nam*, *dom\_ctl*).

<sup>1</sup>[http://www.cesm.ucar.edu/working\\_groups/Software/dev\\_guide/dev\\_guide/node7.html](http://www.cesm.ucar.edu/working_groups/Software/dev_guide/dev_guide/node7.html)

## 4 Style rules

### 4.1 Argument list format

Routines argument lists will contain a maximum 5 variables per line, whilst continuation lines can be used. 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.

```
SUBROUTINE tra_adv_eiv( kt, pun, pvn, pwn )
    CALL tra_adv_eiv( kt, zun, zvn, zwn )
```

### 4.2 Array syntax

Except for long loops (see below), array notation should be used if possible. 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 ) )
```

For 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
```

### 4.3 Case

All FORTRAN keywords are in capital :

```
DIMENSION, WRITE, DO END DO, NAMELIST
```

All other parts of the NEMO code will be written in lower case.

### 4.4 Comments

Comments in the code are useful when reading the code and changing or developing it.

The full documentation and detailed explanations are to be added in the reference manual (TeX files, aside from the code itself).

In the code, the comments should explain variable content and describe each computational step.

Comments in the header start with `!!`. For more details on the content of the headers, see Content rules/Headers in this document.

Comments in the code start with `!”`.

All comments are indented (3, 6, or 9 blank spaces).

Short comments may be included on the same line as executable code, and an 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
```



## 4.10 Loops

Loops, if explicit, should be structured with the do-end do construct as opposed to numbered loops. Nevertheless non-number label can be used for a big iterative loop of recursive algorithm. In case of long loop, a self-descriptive label can be used (i.e. not just a number).

## 4.11 Naming Conventions: files

A file containing a module will have the same name as the inside module.

## 4.12 Naming Conventions: modules

Use meaningful English name and the 3 letters naming convention: first 3 letters for the code section, and last 3 to describe the module. For example, zdf\_tke, where zdf stands for vertical diffusion, and tke for turbulent kinetic energy. Modules must be called with the same name as the file in which they reside, because dependency rules used by "make" programs are based on file names<sup>2</sup>.

Note that by implication multiple modules are not allowed in a single file. The use of common blocks is deprecated in Fortran 90 and their use in NEMO 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 attributes.

## 4.13 Naming Conventions: variables

All variable should be named as explicit as possible in English. The naming convention concerns prefix letters of these name, in order to identify the variable type and status.

Never use a FORTRAN keyword as a routine or variable name.

Table below lists the starting letter(s) to be used for variable naming, depending on their type and status:

Type / Status	integer	real	logical	character	double precision	complex
public or module variable	<b>m n</b> <i>but not</i> <b>nn_</b>	<b>a b e f g h o</b> <b>q to x</b> <i>but not</i> <b>fs rn_</b>	<b>l</b> <i>but not</i> <b>lp ld ll ln_</b>	<b>c</b> <i>but not</i> <b>cp cd cl cn_</b>	<b>d</b> <i>but not</i> <b>dp dd dl dn_</b>	<b>y</b> <i>but not</i> <b>yp yd yl</b>
dummy argument	<b>k</b> <i>but not</i> <b>kf</b>	<b>p</b> <i>but not</i> <b>pp pf</b>	<b>ld</b>	<b>cd</b>	<b>dd</b>	<b>yd</b>
local variable	<b>i</b>	<b>z</b>	<b>ll</b>	<b>cl</b>	<b>cd</b>	<b>yl</b>
loop control	<b>j</b> <i>but not</i> <b>jp</b>					
parameter	<b>jp</b>	<b>pp</b>	<b>lp</b>	<b>cp</b>	<b>dp</b>	<b>yp</b>
namelist	<b>nn_</b>	<b>rn_</b>	<b>ln_</b>	<b>cn_</b>	<b>dn_</b>	
CPP macro	<b>kf</b>	<b>sf</b>				

## 4.14 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 >

<sup>2</sup>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.

## 4.15 Pre processor

Where the use of a language pre-processor is required, it will be the C pre-processor (cpp).

The cpp key is the main feature used, allowing to ignore some useless parts of the code at compilation step.

The advantage is to reduce the memory use; the drawback is that compilation of this part of the code isn't checked. The cpp key feature should only be used for a few limited options, if it reduces the memory usage. In all cases, a logical variable and a FORTRAN *IF* should be preferred. When using a cpp key *key\_optionname*, a corresponding logical variable *lk\_optionname* should be declared to allow FORTRAN *IF* tests in the code and a FORTRAN module with the same name (i.e. *optionname.F90*) should be defined. This module is the only place where a #if defined command appears, selecting 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 following syntax:

```
#if defined key_optionname
!! Part of code conditionally compiled if cpp key key_optionname is active
#endif
```

Is to be used rather than the #ifdef abbreviate form since it may have conflicts with some Unix scripts.

Tests on cpp keys included in NEMO at compilation step:

- The CPP keys used are compared to the previous list of cpp keys (the compilation will stop if trying to specify a non-existing key)
- If a change occurs in the CPP keys used for a given experiment, the whole compilation phase is done again.

## 5 Content rules

### 5.1 Configurations

The configuration defines the domain and the grid on which NEMO is running. It may be useful to associate a cpp key and some variables to a given configuration, although the part of the code changed under each of those keys should be minimized. As an example, the "ORCA2" configuration (global ocean, 2 degrees grid size) is associated with the cpp key *key\_orca2* for which

```
cp_cfg = "orca"
jp_cfg = 2
```

### 5.2 Constants

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, 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.

### 5.3 Declaration for variables and constants

#### 5.3.1 Rules :

Variables used as constants should be declared with attribute PARAMETER and used always without copying to local variables, in order to prevent from using different values for the same constant or changing it accidentally.

- Usage of the DIMENSION statement or attribute is required in declaration statements
- Attribute SAVE is banned. This simplifies the use of AGRIF software. Since all the subroutines are embedded into a module, the variables whose value have to be preserved between two calls can be declared in the module interface. 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 <sup>3</sup>
- 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 on the same line. <sup>4</sup>

For example:

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

### 5.3.2 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 needed. Improper data initialisation is another common source of errors. <sup>5</sup> To avoid problems, initialise variables as close as possible to where they are first used.

### 5.3.3 Attributes:

*PRIVATE/PUBLIC*: All resources of a module are *PUBLIC* by default. A 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 the routines which are in the same module. This is accomplished with the *PRIVATE* attribute.

*INTENT*: All dummy arguments of a routine must include the *INTENT* clause in their declaration in order to improve control of variables in routine calls.

## 5.4 Headers

Prologues are not used in NEMO for now, although it may become an interesting tool in combination with ProTeX auto documentation script in the future. Rules to code the headers and layout of a module or a routine are illustrated in the example module available with the code: *NEMO/OPA\_SRC/module.example*

## 5.5 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.

## 5.6 I/O Error Conditions

I/O statements which need to check an error condition will use the *iostat* =< *integervariable* > 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.

## 5.7 PRINT - ASCII output files

Output listing and errors are directed to *numout* logical unit =6 and produces a file called *ocean.output* (use *ln\_prt* to have one output per process in MPP). Logical *lwp* variable allows for less verbose outputs. To output an error from a routine, one can use the following template:

```
IF( nstop /= 0 .AND. lwp ) THEN    ! error print
  WRITE(numout, cform_err)
  WRITE(numout,*) nstop, ' error have been found'
ENDIF
```

<sup>3</sup>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).

<sup>4</sup>This allows a easy research of where and how a variable is declared using the unix command: `grep var *90 —grep !:`

<sup>5</sup>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.

## 5.8 Precision

Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size. The `f95KIND` 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 using a specific module (`OPA_SRC/par_kind.F90`) declaring the "kind definitions" to obtain the 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.

Thus `wp` being the "working precision" as declared in `OPA_SRC/par_kind.F90`, declaring real array `zpc` will take the form:

```
REAL(wp), DIMENSION(jpi, jpj, jpk) :: zpc      ! power consumption
```

## 5.9 Structures

The TYPE structure allowing to declare some variables is more often used in NEMO, especially in the modules dealing with reading fields, or interfaces. For example

```
! Definition of a tracer as a structure
TYPE PTRACER
  CHARACTER(len = 20) :: sname !: short name
  CHARACTER(len = 80) :: lname !: long name
  CHARACTER(len = 20) :: unit  !: unit
  LOGICAL              :: lini  !: read in a file or not
  LOGICAL              :: lsav  !: ouput the tracer or not
END TYPE PTRACER

TYPE(PTRACER) , DIMENSION(jptra) :: tracer
```

Missing rule on structure name??

## 6 Packages coding rules

### 6.1 Bounds checking

NEMO is able to run when an 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 is forbidden also because it effectively disables array bounds checking.

### 6.2 Communication

A package should refer only to its own modules and subprograms and to those intrinsic functions included in the Fortran standard.

All communication with the package will be through the argument list or namelist input.<sup>6</sup>

### 6.3 Error conditions

When an error condition occurs inside a package, a message describing what went wrong will be printed (see PRINT - ASCII output files). 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, see `stpctl.F90`.

<sup>6</sup>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.

## 6.4 Memory management

The main action is to identify and declare which arrays are PUBLIC and which are PRIVATE. Dynamic memory allocation should be avoided, unless necessary. Indeed, it may be desirable in some occasions. However, this type of memory allocation can reduce performance on some machines, and some debuggers may get confused when trying to diagnose the contents of such variables.

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.<sup>7</sup> An example of an automatic array is:

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

Whereas the same example with *ALLOCATE* is:

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

## 6.5 Optimisation

Considering the new computer architecture, optimisation cannot be considered independently from the computer type. In NEMO, portability is a priority, before any too specific optimisation. Some tools are available to help: For vector computers:

- using *key\_vectopt\_loop* allows to unroll a loop

## 6.6 Package attribute: PRIVATE, PUBLIC, USE, ONLY

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.

## 6.7 Parallelism: using MPI

NEMO is written in order to be able to run on one processor, or on one or more using MPI (i.e. activating the cpp key *key\_mpp\_mpi*, and defining the number of subdomains in latitude and longitude. The domain decomposition divides the global domain in cubes (see NEMO reference manual). Whilst coding a new development, the MPI compatibility has to be taken in account (see *lib\_mpp.F90*) and should be tested.

## 7 Features to be avoided

The code must follow the current standards of FORTRAN and ANSI C. In particular, the code should not produce any WARNING at compiling phase, so that user can be easily alerted of potential bugs when some appear in their new developments. ). Below is a list of features to avoid:

- COMMON blocks (use the declaration part of MODULES instead)
- EQUIVALENCE (use POINTERS or derived data types instead to form data structures)
- 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)

<sup>7</sup> *POINTER* and *ALLOCATE* are widely used in AGRIF, and in the routine reading input files.

- FORMAT statements (use character parameters or explicit format- specifiers inside the READ or WRITE statement instead)
- GOTO and CONTINUE statement (use IF, CASE, DO WHILE, EXIT or CYCLE statements or a contained
- PAUSE
- ENTRY statements: a subprogram must only have one entry point.
- RETURN it is obsolete and so not necessary at the end of program units
- STATEMENT FUNCTION
- Avoid functions with side effects. <sup>8</sup>
- DATA and BLOCK DATA - (use initialisers)

---

<sup>8</sup>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.