Max-Planck-Institut für Meteorologie

Max-Planck-Institut für Meteorologie
Bundesstr. 53
D-20146 Hamburg

Deutscher Wetterdienst
Frankfurter Str. 135
D-63067 Offenbach

ICON Programming Standard

Luca Bonaventura[1], Monika Esch[1], Helmut Frank[2], Marco Giorgetta[1], Thomas Heinze[2], Peter Korn[1], Luis Kornblueh[1], Detlev Majewski[2], Andreas Rhodin[2], Pilar Rípodas[2], Bodo Ritter[2], Daniel Reinert[2], and Uwe Schulzweida[1]

[1] *Max-Planck-Institut für Meteorologie, Bundesstr. 53, D-20146 Hamburg, Germany*
[2] *Deutscher Wetterdienst, Frankfurter Str. 135, D-63067 Offenbach, Germany*

August 6, 2010

Contents

Part I.
ICON programming standard

1. Introduction

Programming standards have been developed earlier to support the development of big software packages following some technical standards and to increase readability. Examples are the *Rules for Interchange of Physical Parametrizations* by Kalnay et al. (1989) or the *European Standards For Writing and Documenting Exchangeable Fortran 90 Code* by Phillip et al. (1995) . The ICON programming standard is inspired by such standards. It is tailored for the development of the Fortran codes of the ICON models, which will be used on a wide variety of computer systems, Unix or Linux related operating systems, and compilers. The goal of this programming standard is to contribute to a stable, technically up-to-date and well readable code. This concerns:

- writing of readable/comprehensive source code
- modularizing of code with well structured dependencies between `MODULE`s
- standardizing the look and usage of `MODULE`s
- avoiding semantical errors
- reducing the maintenance cost
- creating machine independent source code
- well defined exception/error handling
- version control
- quality management and control

2. Programming language

Codes shall generally be written in Fortran 2003. Excellent textbooks are given by Metcalf et al. (2004) and Adams et al. (2009). However, some relevant compilers do not comply with the full standard, so that not all Fortran 2003 features can be used for ICON codes. Depreciated or obsolete features of Fortran 2003 shall not be used. Vendor specific extensions are not allowed. In cases, where Fortran 2003 is insufficient, e.g. for including libraries written in C, ANSI C with POSIX extensions can be used. This type of code should be stored in a separate directory.

3. Rules, conventions, and recommendations

The general objective behind a style guide is to write portable code that is easily readable and has a common style that can be maintained by a team of developers. Many rules follow common sense and should be obvious. We note, that many of the formatting suggestions are easily achieved if you use the GNU emacs (or xemacs) editor in Fortran 90 mode. The ICON coding standard comprises three sections:

- *Rules* comprise style features, which can be checked or diagnosed by scripts or compiler warnings. Some features may even be adjusted by scripts.
- *Conventions* comprise style features, which cannot be checked by a script but are highly recommended to follow.

- *Recommendations* refer to style features, which are mostly helpful but not necessarily are applicable in all cases.

3.1. Rules

- Use free format syntax.
- Characters
  - Keywords in upper case
  - Declared names (subroutines, functions, types, variables, etc.) in lower case
  - Do not use tab characters in your code: this ensures that the code looks as intended, independent of local tab definitions.
- Line continuation
  - Lines have a maximum length of 99 characters. This is convenient for viewing and printing.
  - Continuation lines start with the "&" sign.
  - Split equation so that the operator ("+", "-", ...) follows the "&" sign on the following line.
  - Align the end-of-the-line "&" of a continuation block.
  - Align the beginning-of-the-line "&" of a continuation block.
- Variables, constants , and operands
  - Variable declarations always with "::" syntax
  - Variable declarations always without DIMENSION attribute
  - Variables and numbers of type `REAL` are declared with an explicit kind specifier to be used from module `mo_kind`, see Appendix.
  - Use the working precision `wp`.
  - No implicit casting, i.e. all operands of an equation must be of the same kind.

  Example for the use of `wp`:

```
USE mo_kind, ONLY: wp
IMPLICIT NONE
! Declaration of a constant  of type REAL
REAL(wp), PARAMETER :: a_hour = 3600._wp  ! 1 hour in seconds
! Declaration of a constant  of type INTEGER
INTEGER,  PARAMETER :: two    = 2
! Declaration of a 2d field
REAL(wp), POINTER   :: z_snowcover(:,:)
! Declaration of a local variable
REAL(wp) :: z
! make all operands of the same type as the variable z
z = 4.0_wp * REAL(two,wp)
```

  - Names of variables of type `LOGICAL` start with `l_`
  - Names of types start with `t_`
  - Variables used as constants should be declared with the `PARAMETER` attribute and used always without copying to local variables. This prevents from using different values for the same constant.

- Indentation
  - Comments have to be aligned with the source code.
  - Indentation by 2 blanks has to happen when scope changes.
  - The leading "&" character of continuation lines is indented by 2 characters w.r.t. the very beginning of the continued line. Text following the leading "&" character should be vertically aligned across the continuation lines belonging together.
- Do not use
  - STOP! Instead use subroutine finish of `mo_exception` (The only exception is `mo_mpi`)
  - PRINT and WRITE! Instead use subroutine message of `mo_exception`.
- Intrinsics, programs, functions and subroutines
  - Use generic intrinsic functions only
  - `USE` statements of a Fortran module are collected in a block directly following the `MODULE` or `PROGRAM` statement.
  - `USE` statements are always used as `USE` @module, `ONLY:`@ with an explicit list of the used items.
  - Each Fortran module or program contains a single `IMPLICIT NONE` statement, following directly the `USE` block, if existing, or otherwise following directly the `MODULE` statement.
  - Each Fortran module contains a single `PRIVATE` or `PUBLIC` statement, following directly the `IMPLICIT NONE` statement. This `PRIVATE` or `PUBLIC` statement defines the default external accessibility of the items (parameters, variables, procedures, ...) defined in this module. (It is generally safer to use `PRIVATE` as default.)
  - The default `PRIVATE` (or `PUBLIC`) statement is followed by a `PUBLIC :: ...` (or `PRIVATE :: ...`) statement listing all items for which the default setting does not hold.
  - Subroutines and functions must follow a `CONTAINS` statement, i.e. they must be embedded in Fortran modules.
  - Arguments of subroutines, which are not of type `POINTER`, are declared with `INTENT(in|out|inout)`
  - Arguments of functions, which are not of type `POINTER`, are declared with `INTENT(in)`

3.2. Conventions

- All new Fortran codes are based on the ICON Fortran template files
- Like the templates suggest always name program units and always use the `END PROGRAM`; `END SUBROUTINE`; `END INTERFACE`; `END MODULE`; etc constructs, again specifying the name of the program unit. This helps finding the end of the current program entity. `RETURN` is obsolete and so not necessary at the end of program units.
- Programming and commenting in English. Use meaningful English variable names.
- Never use a Fortran 2003 keyword as a name of a routine or variable!
- CPP keys are only used for:
  - Differentiating codes in relation to computer architecture and compiler properties
  - Controlling the access to external codes (e.g. MPI and CDI library).

- Documentation of almost all used compiler directives can be found in the following manuals. The list is not complete, because NEC refuses to publish it's compiler documentation to the general public.
  - Standard pre-defined C/C++ Compiler Macros
  - Intel compiler documentation
  - Sun compiler documentation
  - PGI compiler documentation
  - NAG compiler documentation
  - IBM compiler documentation
  - GCC compiler documentation
  - OpenMP specification
- `CHARACTER(len=*) , PARAMETER :: ' $ module_subroutine_name $ID: n/a$ ´`, needs a script for the existing code and ..., use it with message and finish. Unify current usage.
- Separate the information to be output from the formatting information on how to output it on I/O statements. E.g. don't put text inside the brackets of the I/O statement.

3.3. Recommendations

- In general subroutines or functions should not exceed a few hundred lines and each programming unit should begin with a header explaining the given sections.
- Any date follows the ISO 8601 standard. That is: YYYY-MM-DD HH:MM:SS. Depending on the case, time information HH:MM:SS or its SS portion can be omitted.
- 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
i=1
z_meaningfulname=3.0_wp
z_SillyName=2.0_wp
```

  write:

```
! Initialize variables
i                = 1
z_meaningfulname = 3.0_wp
z_silly_name     = 2.0_wp
```

- Try to avoid using transcendental functions (`EXP, SIN, COS, ...`). If possibel use tables and interpolations, or use linearized versions.
- Try to prevent `IF`'s in a loop, instead of `IF` use, if possible, `SELECT CASE`, or `MERGE`
- Avoid passing strings in the argument list (except for passing file names - special treatment would be fine)
- 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's shape should be shown in brackets, e.g.:

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

- Use of >, >=, ==, <, <=, /= instead of .GT., .GE., .EQ., .LT., .LE., .NE. in logical comparisons is recommended. The new syntax, being closer to standard mathematical notation, should be clearer.
- We recommend against the use of recursive routines for efficiency reasons for computational intensive routine.
- 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 ICON coupler termination routine `finish` must be called instead of issuing a Fortran STOP. Otherwise a message-passing version of the model could hang.

4. Naming convention

In order to have a readable an easily understandable code, we decided on a naming convention for files as well as for variables, prefixes and so on.

4.1. Pre- and suffixes

- Fortran files
  - driver programs: `atm_master.f90`, `oce_master.f90`, `cpl_master.f90`, ...
  - module files: `mo_<name>.f90`
  - utility interface files: `util_<name>.c` => `mo_util_<name>.f90`
- include files for namelists
  - collect in `icon-dev/include/`
  - fortran include files are named `<atm|oce|lnd|cpl>_<name>_ctl.inc`
- functions/subroutines
  - `init_...`: allocating and setting time invariant variables, done once only at the beginning of the model run
  - `setup_...`: set default values for namelists, read namelists, consistency checks, ...
  - `prepare_...`: within the time loop, set time variant fields/switches
  - `clean_...`: deallocate variables, ...
- types: `t_<name>`
- variables
  - English, concise
  - Variables related to the parallelization have the prefix `p_`.
  - Variables of type LOGICAL have the prefix "`l_`".
  - Local scratch variables: `z_aux, i_aux, l_aux`, (+ suffixes for grid position, if useful)
- optional prefixes
  - derivatives
    - `ddt_` : temporal dervatives

    - ddxn_ : horizontal derivatives in normal direction
    - ddxt_ : horizontal derivatives in tangential direction
    - ddz_ : physical space vertical derivative (height coords.)
    - ddp_ : physical space vertical derivative (pressure coords.)
  - fluxes
    - flx_
- optional suffixes: order: <process><time>_<vertical pos.><horizontal pos.>
  - processes: <name>_rad etc.
  - 2m: _2m
  - surface: _sfc
  - snow: _snow
  - ice: _ice
  - horizontal position
    - _c: cell center
    - _e: edge
    - _v: vertex
  - vertical position
    - _m: main level/mid level/full level
    - _i: interface between layers/half levels
  - time position
    - _old
    - _now
    - _new
  - vector representations
    - _o: orthogonal
    - _q: contravariant
    - _p: covariant
  - There should be as few as possible suffixes (as many as necessary). If there are more than one suffixes needed, their ordering follows the list sequence above.
- constants
  - _dp, _sp floating point convention

The important variables with global scope are supposed to follow the CF-Conventions as they are adopted as well for PRISM, ESMF, and other international projects. The tables can be found at: http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html

5. Automatic documentation

Each function, subroutine, or module based on the template files (see Appendix) includes a prologue instrumented for use with the Doxygen documentation tool (http://www.doxygen.org).

Doxygen can extract information on the structure of the code and compile lists of the declared objects, as for example types, variables, routines, modules,, and can show dependencies. In addition Doxygen can compile texts embedded as comments in Fortran codes if marked by appropriate formatting instructions. Running `make doc` in the ICON base directory (after the Makefile has been generated by `./configure`) generates an HTML formatted Doxygen documentation.

6. Parallelization issues

ICON models are parallelized by MPI and/or OpenMP parallelization. MPI parallelization means that separate model executables are running on sub-domains and communicating with each other following the MPI standard. Following the MPI standard each such model is named a process. A MPI parallelized ICON model is hence split into a number of processes numprocs, or for short nprocs. OpenMP parallelization means that the workload of a single model executable is shared by a number of processors or so-called threads. If both methods are combined, the total workload is shared by num_procs_ * num_threads "CPUs". The naming convention used below is borrowed from the terms and names used in the MPI and OpenMP standards. For variables in Fortran codes, the convention is to use names with prefix "`p_`".

6.1. Fine-grained parallelism using OpenMP

Any OpenMP parallelization should be based on the most recent OpenMP 3.0 standard.
http://www.openmp.org/mp-documents/spec30.pdf
Here are the most important rules that we would ask you to adhere to, when implementing OpenMP directives. Due to the fact that different components coupled in an MPMD fashion may require different total numbers of threads, a single environment variable `OMP_NUM_THREADS` needs later to be replaced by specific environment varaibles OMP_ATM_THREADS and `OMP_OCE_THREADS`.

	Environment variables in shell scripts	**Name in Fortran code**
number of threads per process	`OMP_NUM_THREADS`, later: `OMP_ATM_THREADS` and `OMP_OCE_THREADS`	`p_nthreads`
ID of a single MPI process	n.a.	`p_threads`

Table 1: OMP of ICON.

- All OpenMP compiler directives must start with the directive sentinel `!$OMP` when using the (recommended) free format syntax.
- Do not use combined parallel worksharing constructs like `!$OMP PARALLEL DO` or `!$OMP PARALLEL WORKSHARE`. Instead use the parallel construct (`!$OMP PARALLEL`) followed by the desired worksharing construct (e.g. `!$OMP DO`). This will increase readability of the code.
- Explicitly include loop control variables of a parallel `DO` loop or a sequential loop enclosed in a parallel construct into a `PRIVATE` clause. Although those variables are `PRIVATE` by default, please stick to this since it will increase readability.

- Be aware of Race-Conditions!
  Definition: Two threads access the same shared variable **and** at least one thread modifies the variable **and** the accesses are concurrent, i.e. unsynchronized.
  Prototypical example:

```
a(1) = 0._wp
!$OMP PARALLEL
!$OMP DO
DO i=2, n
  a(i) = 2.0_wp * i * (i-1)
  b(i) = a(i) - a(i-1)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

  Note that this can lead to unexpected results. It is not ensured that $a(i-1)$ has already been calculated, when accessed for the calculation of $b(i)$.

  ICON-specific example:

```
#ifdef TEST_OPENMP
!$OMP PARALLEL PRIVATE(i_startblk,i_endblk)
#endif
i_startblk = ptr_patch%cells%start_blk(rl_start,1)
i_endblk   = ptr_patch%cells%end_blk(rl_end,i_nchdom)
#ifdef TEST_OPENMP
!$OMP DO PRIVATE(...)
#endif
.
 Do some work
.
#ifdef TEST_OPENMP
!$OMP END DO
!$OMP END PARALLEL
#endif
```

  Even in this case it is necessary to declare `i_startblk` and `i_endblk` as `PRIVATE`, although each thread writes the same result to `i_startblk` and `i_endblk`. On the IBM-machine writing to the same shared variable simultaneously leads to unexpected results.
- Check for strong sequential equivalence (i.e. bitwise identical results) of your parallelized code whenever possible. If a reduction operator (`reduction(operator: list)`) is used, strong sequential equivalence is unlikely to occur. Then, at least, check for weak sequential equivalence (equivalent mathematically, but due to the quirks of floating point arithmetic, not bitwise identical).
- Note: reduction operators as well as the `WORKSHARE` directive may still be computationally inefficient depending on the compiler. Careful runtime testing is required if those constructs are used.

6.2. Parallelism using MPI

Scripts may need to distinguish the number of processes used by different models, like the atmosphere and ocean models, and the coupler. Fortran variables and subroutines/functions related to MPI parallelization are named with a prefix "`p_`".

	Variables in shell scripts	**Name in Fortran code**
number of threads per process	NPROCS, later: ATM_NPROCS, OCE_NPROCS and CPL_NPROCS	p_nprocs
ID of a single process	n.a.	p_rank

Table 2: MPI of ICON.

7. Literature

Adams, J.C., W.S. Brainerd, R.A. Hendrickson, R.E. Maine, J.T. Martin, and B.T. Smith, The Fortran 2003 Handbook, Springer, 712 p., 2009. Kalnay, E. et al., Rules for Interchange of Physical Parametrizations, Bull. A.M.S., 70 No. 6, p 620, 1989. Metcalf, M., J. Reid and M. Cohen, Fortran 95/2003 explained, Oxford University Press, 412 p., 2004. Phillip, A., G. Cats, D. Dent, M. Gertz, and J. L. Ricard, European Standards For Writing and Documenting Exchangeable Fortran 90 Code, Version 1.1,1995.
http://www.meto.gov.uk/research/nwp/numerical/fortran90/f90_standards.html

Part II.
Appendix

A. Template Fortran files

Template Fortran files are provided to support the programing following the programming style described above:

- Main programs: `src/templates/template_main.f90`
- Modules: `src/templates/template_module.f90`
- Subroutines: `src/templates/template_subroutine.f90`
- Functions: `src/templates/template_function.f90`

B. Kind specifiers for REALs and INTEGERs

All REAL numbers and variables must be defined or declared, respectively, with a specified REAL kind. For integers this is advised for loop indices only. Kind specifiers are provided in:

- `src/shared/mo_kind.f90`

`mo_kind` provides the following kind specifiers:

REAL kind	precission	assumed number of bits
`sp`	6 digits	32
`dp`	12 digits	64
`wp=dp`	working precission	
INTEGER kind	**exp. range**	**assumed number of bits**
`i4`	4	32
`i8`	8	64

Note: The bit sizes given are not mandatory. The kind value is defined by the precision/range, which results on current systems in these bit sizes. This may change in future.

C. Constants

Universal mathematical and physical constants are defined in modules. Such constants can be used elsewhere by USE association, and they must not be defined locally.

- Mathematical constants: See file `src/shared/mo_math_constants.f90`
- Physical constants: See file `src/shared/mo_physical_constants.f90`

D. Editor customization

Some editors support customized formatting for line indentation and adjustments of keyword upper-case typing etc. This can be employed to facilitate programming in the style described above. Find below settings for Emacs and Vim.

D.1. Emacs

Add the following lines to the ".emacs" file in your Unix home directory to support the Fortran programming following the standard described above.

```
(custom-set-variables
  ;; custom-set-variables was added by Custom.
  ;; If you edit it by hand, you could mess it up, so be careful.
  ;; Your init file should contain only one such instance.
  ;; If there is more than one, they won't work right.
 '(f90-mode-hook (quote (f90-add-imenu-menu)))
 '(f90-auto-keyword-case (quote upcase-word))
 '(f90-comment-region "!!$")
 '(f90-directive-comment-re "![\\$hHdDcCoOI][oOpPiIdDcCB][mMfFrRiIlL]")
 '(f90-indented-comment-re "!")
 '(f90-program-indent 2)
 '(f90-type-indent 2)
 '(f90-associate-indent 2)
 '(f90-do-indent 2)
 '(f90-if-indent 2)
 '(f90-continuation-indent 2)
 '(f90-beginning-ampersand t)
 '(f90-break-before-delimiters t)
 '(f90-break-delimiters "[-+\\*/><=,%␣␣␣]")
 '(f90-smart-end (quote blink))
 '(show-paren-mode t)
 '(line-number-mode t)
 '(column-number-mode t)
 '(size-indication-mode t)
)
(custom-set-faces
  ;; custom-set-faces was added by Custom.
  ;; If you edit it by hand, you could mess it up, so be careful.
  ;; Your init file should contain only one such instance.
  ;; If there is more than one, they won't work right.
 )
(if (window-system)
    (set-frame-width (selected-frame) 99)
)
```

D.2. Vim

Add the following lines to the Vim setup file in your Unix home directory::

```
set nocompatible
filetype plugin indent on
syntax enable
set shiftwidth=2
set smarttab
set autoindent
set expandtab
```

E. Preprocessor directives

Preprocessor directives are most useful for distinguishing architectures and operating system dependent code variants. However, they are usually not meant for selecting code options, with

the exception of debugging. Use preprocessor directives as seldom as possible and as often as necessary! The following compiler predefined preprocessor macros are available:

IBM xlf, xlc	__xlC__
NEC f90	__SX__
SUN f95	__SUNPRO_F95
GCC ($\geq$ 4.3.0)	__GFORTRAN__
PGI pgf95	__PGI
Intel	__INTEL_COMPILER

If you like to set groups, make that at the beginning of a source code file like:

```
#if defined (__PGI)
#define __ASYNC_GATHER 1
#define __ASYNC_GATHER_ANY 1
#endif
!
! Reshape is not that powerful implemented in some compiler
!
#if defined (__sun) || (__SX__) || defined (__PGI) || defined (__GFORTRAN__)
#define __REPLACE_RESHAPE 1
#endif
!
! Switch on explicit buffer packing and unpacking, allowing vectorization
!
#if defined (__SX__) || defined(__PGI) || (defined __xlC__)
#define __EXPLICIT 1
#endif
!
! Select communication type, if not defined NON BLOCKING is selected
!
#if defined (__PGI)
#define __SENDRECV 1
#endif
```