IFS_coding_guidelines Documentation

ECMWF

Oct 31, 2023

List of rules:

Developments to the Arpege/IFS codebase should aim to adhere to the guidelines and rules presented in this document.

Guidelines and rules described here are designed to make code easier to read, easier to maintain, easier to extend, and less prone to bugs. The code review process for submissions to Arpege and IFS could aim to check code against these guidelines. Rules are also written so as to allow implementation of automatic checking.

CHAPTER 1

Guidelines and recommendations

- Good code should not require large amounts of comments to be intelligible. Nevertheless targeted explanations of particular segments of interest are desirable. Each source file should have a homogeneous coding style.

>>PM: suggestion: remove all history comments at the beginning of each routine; these can be obtained from git history.<<

- Contours of a routine or module should be considered with care, avoiding excessive length or complexity.
- Routine call signatures or interfaces should be designed with care, respecting library contouring. Interfaces that are not internal to a component should privilege as much as possible native fortran datatypes rather than derived types.

>>PM: this does not make sense; most modern libraries (eg Atlas) use derived types.<<

- Naming of new variables, routines and modules should help the reader understand code as efficiently as possible. *Renaming of legacy / existing code?*
- Large arrays should be declared as allocatable, to avoid excessive stack usage. Small arrays, and in particular those declared in tight code (this should be avoided wherever possible!) should be automatic, to benefit from faster stack handling.

>>PM: definition of large and small arrays ? I recommand banning ALLOCATABLEs in all NPROMA routines (ie those which process a single NPROMA block).<<

- If an allocatable variable can be used rather than a pointer, opt for the allocatable for safety reasons.
- In order to make domain decomposition easier to follow, global variable names are suffixed by G, while subdomain-local variables are suffixed by L.
- Different meteorological data formats are used at ECMWF and Meteo-France. The choice between these formats should be based on logical keys LARPEGEF or LARPEGEF_xx (and not LECMWF).
- Aladin routines that are counterparts of IFS/Arpege ones should have the same name but prefixed with E. Aladin counterparts to IFS/Arpege SUxxx setup routines should be named SUE.
- Output that should appear in the main text output file should be written to NULOUT. Output to NULOUT must be deterministic and should not change according to the parallel distribution or the time at which the job is run. Error messages should be written to unit NULERR.

- Conditional clauses with multiple cases should be handled with SELECT CASE rather than IF statements followed by multiple ELSEIF statements.
- If execution is to be aborted by the code, a call to ABOR1, with a meaningful message, should be used.

>>PM: I think we should also discuss how source files are organized in the git repository; in particular, having an arpifs/module directory with all the modules does not make sense.<<

CHAPTER 2

Rules (as checked by norms checker)

Some of the rules relate to the idea of Single Column code, where algorithmic tasks can be expressed independently of horizontal position, and no horizontal dependencies exist. Code which maps to this concept can be modified at compile time by tooling (Loki, Fxtran) in line with architecture-specific requirements. Such tooling relies heavily on code formatting to determine required transformations, explaining the prescriptive rules for these areas of code.

Rules are organised into general language rules (Lw), IFS-specific rules (Ix), stylistic points (Sy), and Single-Column related rules (SCz).

2.1 L1 : implicit none

`IMPLICIT NONE` must figure in all scoping units

Once per module is sufficient

2.2 L2 : use module, only

Module imports via the USE statement shall contain an ONLY specifier.

To be avoided :

```
USE GEOMETRY_MOD
USE YOMRIP
```

Correct way to import from modules :

```
USE GEOMETRY_MOD, ONLY : TYPE_GEOMETRY
USE YOMRIP,       ONLY : NSTADD
```

2.2.1 Exceptions

- Fypp-based modules, as cmake's dependency analysis does not know what symbols a fypp module will expose.
- modules where ASSIGNMENT operator is overloaded

2.3 L3 : global variables

Only parameters to be declared as global variables.

>>PM: Does this mean that global variables are forbidden ? I think it is a good idea, but there is a lot of work to do to get there.<<

Example :

Listing 1: global variable usage

```
MODULE YOMLUN

INTEGER(KIND=JPIM), PARAMETER :: NULSTAT =  1
INTEGER(KIND=JPIM), PARAMETER :: NULNAM  =  4

END MODULE
```

2.4 L4 : new language features

New features from recent Fortran standards should not be used if they are not supported by GNU, Intel, & Nvidia compilers.

2.5 L5 : argument INTENT

All arguments to routines shall be declared with an INTENT.

- `INTENT(IN)` : variables which are not assigned to in the scope
- `INTENT(OUT)` : variables whose previous content is irrelevant, and which are written to in the scope
- `INTENT(INOUT)` : all other variables

>>PM: Structures like YDVARS which contain some pointers can have the INTENT(IN) attribute, and have their pointed values modified. This is a problem, as we will have such structures everywhere in the code. Furthermore, using the `INTENT(OUT)` attribute for such variables may cause some compilers (eg NAG) to wipe their contents. And YDVARS also contains some metadata that should remain constant. I think we do not have any other choice but to declare YDVARS (and other similar structures) with `INTENT(IN)` everywhere. <<

warning Particular care should be paid to intent of array variables: * arrays where only a few locations are updated but other locations contain required values, *must* be declared as INTENT(INOUT)

>>PM: no compiler can make the difference between INTENT(INOUT) and INTENT(OUT).<<

- arguments declared as allocatable may be deallocated at entry if declared as INTENT(OUT)

2.6 L6 : OPTIONAL argument position

Optional arguments to a routine shall be after non-optional ones.

To be avoided :

Listing 2: OPTIONAL argument position

```
SUBROUTINE OPT_ARG( ARG1, ARG2, ARG3)

INTEGER(KIND=JPIM), INTENT(IN)           :: ARG1
INTEGER(KIND=JPIM), INTENT(IN),OPTIONAL :: ARG2
INTEGER(KIND=JPIM), INTENT(OUT)          :: ARG3
```

2.7 L7 : OPTIONAL argument keyword usage

Optional arguments passed to a routine shall have their keyword specified.

To be avoided :

Listing 3: OPTIONAL argument keyword

```
!! interface of OPT_ARG subroutine is below
CALL OPT_ARG( IVAL1,IVAL2,IVAL3)


SUBROUTINE OPT_ARG( ARG1, ARG2, ARG3)

INTEGER(KIND=JPIM), INTENT(IN)           :: ARG1
INTEGER(KIND=JPIM), INTENT(OUT)          :: ARG2
INTEGER(KIND=JPIM), INTENT(IN),OPTIONAL :: ARG3
```

2.8 L8 : copying allocatable arrays

An array declared in scope as allocatable shall not be copied without dimension specification.

For example, copy 1D array as `PARR2(:)=PARR1(:)`, and *not* `PARR2=PARR1`. The latter form may lead to reallocation of the left-hand-side target array at runtime, which is error-prone, and costly.

2.9 L9 : interfaces for routines

Calls to subroutines and functions that are provided neither by a module nor by a CONTAINS statement, must have a matching explicit interface block.

Routines that may be called from outside the IFS should have an interface in the codebase, while routines called exclusively internally have their interfaces generated at build time.

2.10 L10 : declaration syntax

All declarations shall use the "::" notation.

2.11 L11 : array syntax

Fortran array syntax, eg :

`ZX(:)=ZY(:)+ZZ(:)`

is forbidden, except for the two following trivial cases:

- Array initialization: `X(:)=constant`
- Array copy : `X(:)=Y(:)`

2.12 L12 : array declarations in NPROMA routines

NPROMA routines (those processing a single NPROMA block) should only declare arrays of basic data types whose kinds are taken from parkind1.F90 and parkind2.F90, eg:

- `REAL(KIND=JPRB)`
- `REAL(KIND=JPRD)`
- `INTEGER(KIND=JPIM)`
- `LOGICAL`

The leading dimensions of these array should be `NPROMA`.

Any other arrays are forbidden in NPROMA routines. These other arrays (not depending on the meteorological situation) should be computed in the setup and passed as arguments to NPROMA compute routines, or integrated into sub-components of YDMODEL.

Arrays declared in NPROMA routines be automatic arrays; ALLOCATABLEs and POINTERs are forbidden.

2.13 L13 : array declarations in parallel routines

Parallel routines (those processing a list of NPROMA blocks) should wrap their field data into Field API objects.

In particular the legacy data representation for model state (PGMV/PGFL arrays) should be considered obsolescent and replaced by YDVARS wherever possible.

Parallel routines should never declare an array that will be involved in calculations (ie passed to NPROMA routines or used in OpenMP sections).

2.14 L14 : notations

Notations should homogeneous for the loop index, NPROMA array leading dimension, first iteration, last iteration:

- `JLON/KLON/KIDIA/KFDIA` in ARPEGE physics
- `JL/KLON/KIDIA/KFDIA` in ECMWF physics
- `JROF/NPROMA/KST/KEND` in dynamics routines (call_sl.F90, cpg_gp.F90, cpg_dyn.F90)

2.15 L15 : dummy/actual array dimensions

If an actual argument is an array, then :

- its rank should match the rank of the corresponding dummy argument
- its dimensions should match those of the corresponding dummy argument

Notable exceptions are the interpolation routines of the Semi-Lagrangian:

- `laitli.F90`
- `laitri.F90`
- etc.

2.16 L16 : INTENT attribute in NPROMA routines

In NPROMA routines (those processing a single NPROMA block), only NPROMA data should be allowed to have an INTENT different of IN. All other argument data should have the `INTENT(IN)` attribute.

This, for instance should be forbidden:

```
SUBROUTINE LAPINEA(&
& YDGEOMETRY, YDML_GCONF,YDML_DYN,KST,KPROF,YDSL,KIBL,PB1,PB2,PWRL9,&
& KVSEPC,KVSEPL,&
& PSAVEDP,PCCO,PUF,PVF,KL0,KLH0,PLSCAW,PRSCAW,KL0H,PLSCAWH,PRSCAWH,&
& PSCO,PGFLT1,KNOWENO)

INTEGER(KIND=JPIM),INTENT(INOUT) :: KVSEPC
INTEGER(KIND=JPIM),INTENT(INOUT) :: KVSEPL
```

In particular, `YDMODEL` and `YDGEOMETRY` as well as all their sub-components passed as arguments (eg `YDDYNA`) should have the `INTENT(IN)` attribute in NPROMA routines.

2.17 L17 : Pointers in NPROMA routines

The use of Fortran pointers in compute NPROMA routines should be forbidden except for the following pattern:

```
SUBROUTINE GPRCP_EXPL (YDCST, YDCPG_BNDS, YDCPG_OPTS, PCP, PR, PKAP, YDVARS, KGFLTYP)

REAL(KIND=JPRB),OPTIONAL,TARGET,INTENT(OUT)    :: PCP(YDCPG_OPTS%KLON,YDCPG_OPTS
↪%KFLEVG)
REAL(KIND=JPRB),OPTIONAL,TARGET,INTENT(OUT)    :: PR(YDCPG_OPTS%KLON,YDCPG_OPTS%KFLEVG)
REAL(KIND=JPRB),OPTIONAL,       INTENT(OUT)    :: PKAP(YDCPG_OPTS%KLON,YDCPG_OPTS
↪%KFLEVG)

REAL(KIND=JPRB), POINTER :: ZR(:,:)
REAL(KIND=JPRB), TARGET  :: ZR0(YDCPG_OPTS%KLON,YDCPG_OPTS%KFLEVG)

IF (PRESENT (PR)) THEN
  ZR => PR
ELSEIF (PRESENT (PKAP)) THEN
  ZR => ZR0
ELSE
```

(continues on next page)

(continued from previous page)

```
  CALL ABOR1 ('GPRCP_EXPL: EXPECTED PR OR PKAP')
ENDIF
```

And such pointers should always point to arrays with the same shapes.

2.18 L18 : Design of NPROMA routines

NPROMA routines should be individual subroutines, allowing for :

- automatic interface generation
- separate compilation
- easiness of automatic analysis and transformation

In particular implementing a routine or a set of subroutines inside a module in order to benefit from the automatic generation of an interface in a .mod file should be prohibited.

The use of modules should be restricted to derived types definition and implementation (methods).

2.19 L19 : Design of derived types

Dervived types should be implemented in modules. Their methods (the minimal code accessing their private attributes) should be implemented in the module where they are defined.

The SEQUENCE statement is forbidden in the definition of complex (embedding some POINTER or ALLOCATABLE members) derived types meant to be used on accelerators, as the SEQUENCE statement forbids the implementation of object methods.

2.20 I1 : naming of variables, subroutines, modules

The IFS has a historical naming convention, which should be adhered to where sensible.

2.20.1 Variable naming

- Local variables
- integer : `Ixxx`
- real : `Zxxx`
- logical : `LLxxx`
- derived type : `YLxxx`
- Argument variables
- integer : `Kxxxx`
- real : `Pxxx`
- logical : `LDxxx`
- derived type : `YDxxx`

- Name suffixes
- L suffix corresponds to local / subdomain variables
- G suffix corresponds to global / full-domain variables

2.20.2 Routine naming

- setup routines should start with subroutines
- TL and AD routines should end with tl and ad respectively

Module naming New modules should end with _mod. The name of the file should match the name of the module it contains (e.g. `MODULE EINT` should be in file eint_mod.F90)

2.21 I2 : KIND specification

Real and integer variables in the IFS shall have an explicit KIND specifier, with a kind defined in PARKIND1 or PARKIND2, or in a similar file for stand-alone projects (ecTRANS, WAM, etc)

2.22 I3 : derived types in modules

Derived type definitions shall not be outside of a module

A module may contain multiple type definitions, but a subroutine should not contain a type definition.

2.23 I4 : Excessively long interfaces

IFS routines should not have more than 50 **(arbitrary value, but used in historical document, and sounds already long to me)** arguments.

Control flow routines should use derived types to group long lists of related variables.

Algorithmic kernels should be tight enough in scope to avoid the need for more than 50 arguments.

Lightweight *_layer* routines can be used as bridge between control flow code and kernel code.

2.24 I5 : MPL and message passing

All operations related to message passing internal to the IFS shall be done via the MPL library.

Nemo code, WAM code, and master.F90 are not concerned by this rule.

2.25 I6 : MPL and string argument

Calls to MPL methods should have string argument identifying call site

2.26 I7 : DRHOOK instrumentation

All IFS routines shall have DRHOOK calls as first and last statement, unless referenced explicitly in exceptions list.

The string argument to these calls must give the name of subroutine. In the case of a contained subroutine, the string should be constructed from the name of the parent routine and the contained routine, separated with a percent symbol (%).

The exceptions list should only contain routines that are called in time-critical / tight contexts, where DRHOOK overhead would not be acceptable.

2.27 SC1 : horizontal indexing

Variables refering to horizontal indices should be named consistently, *JL*, *KPROMA*

The Loki tool relies on elements of code style in order to identify loops needing to be manipulated for architecture specialisation.

2.28 SC2 : horizontal looping

All loops over the innermost, horizontal array dimension should be explicit.

The Loki tool relies on elements of code style in order to identify loops needing to be manipulated for architecture specialisation.

2.29 SC3 : function calls from inside KPROMA loops

Inside tight horizontal loops of type DO `JL=KIDIA,KFDIA`, calls should be restricted to intrinsics.

If a function construct must be used inside the loop, the function shall be pure elemental or defined in a statement function, so as not to inhibit compiler vectorization of the loop.

2.30 SC4 : no horizontal indirection

Where relevant (Single Column), indirect addressing on the innermost, horizontal array index shall not be used.

The Loki tool relies on elements of code style in order to identify loops needing to be manipulated for architecture specialisation.

>>PM: some parts of cucalln.F90 rely on indirect addressing, and can nevertheless transformed. I think this constraint should be relaxed.<<

2.31 S1 : END IF / ENDIF

Two-word control flow constructs should be written as single words.

`ENDIF` rather than `END IF` `ENDDO` rather than `END DO` `ENDWHERE` rather than `END WHERE`

(this one makes me cry a little)

2.32 S2: order of argument declarations

Arguments to a routine shall be declared in the same order as that of the call signature.

To be avoided :

Listing 4: declaration order

```
SUBROUTINE WRONG_ORDER( ARG3, ARG2, ARG1)

INTEGER(KIND=JPIM), INTENT(IN)  :: ARG3
INTEGER(KIND=JPIM), INTENT(IN)  :: ARG2
INTEGER(KIND=JPIM), INTENT(OUT) :: ARG1
```

2.33 S3 : line continuation

Optional arguments to a routine shall be after non-optional ones.

Listing 5: Good line continuation

```
 CALL ROUTINE_WITH_VERY_LONG_AND_BORING_NAME ( IVAL1,IVAL2,IVAL3, &
  &     PVAL_1,PVAL2,PVAL3, &
  &     LDVAL1,LDVAL2)
```

Listing 6: Bad line continuation

```
 CALL ROUTINE_WITH_VERY_LONG_AND_BORING_NAME ( IVAL1,IVAL2,IVAL3, &
       PVAL_1,PVAL2,PVAL3, &
       LDVAL1,LDVAL2)
```

2.34 S4: no unqualified END statements

END statements shall all be qualified. END SUBROUTINE shall state subroutine name.

To be avoided :

Listing 7: declaration order

```
SUBROUTINE WRONG_ORDER( ARG3, ARG2, ARG1)

INTEGER(KIND=JPIM), INTENT(IN)  :: ARG3
INTEGER(KIND=JPIM), INTENT(IN)  :: ARG2
INTEGER(KIND=JPIM), INTENT(OUT) :: ARG1
```

2.35 S5 : unused variables

Variables that are declared and not used, or included and not used, should be pruned.

Warning: the use of statement functions can lead to module variables that appear unused, but that are referenced in statement function.

2.36 S6 : no TABS

TAB characters are not to be used

2.37 Coding norms 2011 rules

2.37.1 R1 : Encapsulation rules

status: not a lintable rule, covered in preamble

Modules should be split up in a sensible manner to avoid too long Fortran files or too complex modules. One recommendation is to separate the data structures, the operators and the descriptive parameters (including setup if existing). The number of entities in a single module is not limited, but a reasonable total number should be considered always (about 10 to 20 entities ?).

2.37.2 R2 : Subroutine length

status : can this remotely be considered enforceable, from current IFS?!

Subroutines should have no more than 300 executable statements. For a module containing several entities, this limit of 300 executable statements is applicable for each inner subroutine.

2.37.3 R3 : Cosmetic changes

status : is this still relevant with git development everywhere, and shared history?

Avoid cosmetic changes that will make merges difficult (such as re-ordering argument lists and USE statements, or changing the indentation of large blocks of code). As an exception, cosmetic changes can happen when a routine is heavily modified, and only one well identified developer will contribute to the code for the next common cycle.

2.37.4 R4 : Unused variables

status : covered in S5

Declarations of unused variables must be removed

2.37.5 R5 : Variable name suffixes

status : not a lintable rule, mentioned in preamble

Variables suffixed with L are local in the sense of the parallel distribution. Variables suffixed with G are global.

2.37.6 R6 : Array syntax

status : covered in L11

The use of array syntax is not recommended except for initialization and very basic computations.

2.37.7 R7 : Cut-n-paste

status : not a rule

Cut-and-paste of existing piece of code should be avoided. Common code should be extracted to a separate subroutine or function.

2.37.8 R8 : LECMWF variable

status : very far from true in current IFS?!

The variable LECMWF should be used only in setup subroutines.

2.37.9 R9 : LELAM variable

status : very far from true in current IFS?!

The variable LELAM is not to be used below SCAN2M.

2.37.10 R10 : LFI or GRIB

status : not a lintable rule, so moved to preamble

The choice between LFI/LFA or GRIB format should be made using the variables LARPEGEF or LARPEGEF-xx (and not LECMWF).

2.37.11 R11 : message passing

status : mentioned elsewhere (I4)

The MPL package must be used as the interface for any message passing.

2.37.12 R12 : Derived types in a module

status : mentioned elsewhere (I5)

Derived types should be declared in a module.

2.37.13 R13 : Code must be threadsafe

status : not a lintable rule, and should be enforced via testing!

2.37.14 R14 : Abnormal termination

status : mentioned in preamble

Abnormal termination must be invoked by ABOR1.

2.37.15 R15 : The save statement

status : no longer applicable

Variables in data modules must be saved using the SAVE statement.

2.37.16 R16 : argument passing characteristics

status : until MPL uses F08 bindings, this is not fully respected in IFS

Array shape and Variable type must not be changed when passed to a subroutine.

2.37.17 R17 : SELECT CASE

status : mentioned in preamble

Use SELECT CASE when possible instead of IF/ELSEIF/ELSE/ENDIF.

2.37.18 R18 : include interfaces

status : reworded, mentioned elsewhere (L9)

For each called routine there must be a “”#include” statement that includes an explicit interface block for the routine. Note that the files containing the explicit interface blocks are automatically generated during compilation.

2.37.19 R19 : number of arguments to routines

status : mentioned elsewhere (I3)

Routines should have a small number of dummy arguments. Routine with more than 50 dummy arguments are not allowed.

2.37.20 R20 : names of variables

status : mentioned in preamble

Variable names should be meaningful to an English reader. Very short names should be reserved for loop indices.

2.37.21 R21 : Name prefixes and suffixes

status : covered elsewhere (I1)

Conventional prefixes or suffixes are to be used for all variables except derived types, as described in table 1 in section 4. There is no naming convention for derived types.

2.37.22 R22 : Aladin routine names

status : covered elsewhere

Aladin subroutines that are counterparts of IFS/Arpege ones should have the same name but prefixed with E. Aladin setup routines that are counterparts of IFS/Arpege (prefixed SU) should be prefixed SUE.

2.37.23 R23 : NULOUT & NULERR usage

status : mentioned in preamble

The logical unit for output listing is NULOUT. Output to NULOUT must be deterministic and should not change according to the parallel distribution or the time at which the job is run. Error messages should be written to unit NULERR.

2.37.24 R24 : universal constants

status : no longer applicable, yomcst is being passed as argument

Universal constants must be stored, saved and initialized in data module YOMCST. They cannot be modified elsewhere and should not be accessed via dummy arguments.

2.37.25 R25 : MPL string argument

status : covered elsewhere (I6)

Calls to MPL subroutines should provide a CDSTRING identifying the caller.

2.37.26 R26 : code structure and file location

status : non-lintable, but mentioned in preamble

Source code is partitioned into projects. Each source file must be put in the proper directory for its project.

2.37.27 R27 : namelist usage

status : scope of this is unclear; IFS picks up a lot of environment variables for example

> Runtime specification of variables must be done using name-lists.

2.37.28 R28 : DATA statement

status : covered elsewhere

DATA statement should be avoided if possible and is allowed only for small lists.

2.37.29 R29 : F90 free format

status : is mention of free format still necessary? Fortran 90 is also perhaps misleading.

The code should be Fortran 90 free format.

2.37.30 R30 : consistent style

status : not a lintable rule, but suggested in preamble.

Use a consistent style throughout each module and subroutine.

2.37.31 R31 : no TAB usage

status : mentioned elsewhere (I7)

The TAB character is not allowed.

2.37.32 R32 : IMPLICIT NONE

status : mentioned elsewhere (L1)

IMPLICIT NONE is mandatory in all routines

2.37.33 R33 : no hard-coded array dimensions

status : ambiguous, possibly no longer relevant

Array dimensions must not be hard-coded.

2.37.34 R34 : "::" notation

status : mentioned elsewehere (L10)

Declarations must use the notation "::".

2.37.35 R35 : mandatory KIND specification

status : updated and mentioned elsewhere (I8)

Variables and constants must be declared with explicit kind, using the kinds defined in PARKIND1 and PARKIND2.

2.37.36 R36 : USE, ONLY

status : updated and mentioned elsewhere (L2)

All USE statements must include an "ONLY" clause, except for modules that override ASSIGNMENT, where this is dangerous.

2.37.37 R37 : PARAMETER wherever possible

status : guideline rather than lintable rule; placed in preamble

Constants should be PARAMETERs wherever as possible

2.37.38 R38 : variable name prefixes

status : covered elsewhere

Variable names should follow the prefix convention defined in table 1.

2.37.39 R39 : banned statements

status : we actually use DIMENSION a **lot**

The following statements are banned :(a)STOP (b)PRINT (c)RETURN (d)ENTRY (e)DIMENSION (f)DOUBLE PRECISION (g)COMPLEX (h)GO TO (i)CONTINUE (j)FORMAT (k)COMMON (l)EQUIVALENCE

2.37.40 R40 : no implicit array sizing

status : is this still actually valid?

Arrays should not be declared with implicit size : "A(*)".

2.37.41 R41 : automatic and allocatable arrays

status : not a lintable rule, mentioned in preamble

Large arrays should be allocatable. Small or low-level arrays should be automatic.

2.37.42 R42 : deallocation of allocatables

status : is this still valid? Would also be extremely difficult to check automatically

All allocated arrays should be explicitly deallocated.

2.37.43 R43 : comparison operators

status : mentioned elsewhere, but ... WHYYYYYY?

Use Fortran 90 comparison operators (e.g. == rather than .EQ.).

2.37.44 R44 : value comparisons

status : guideline rather than rule, mentioned in preamble

Explicitly set variables (parameters, constants, namelist variables,...) should be always exactly compared (using==or=, etc). Evaluated variables (that might be subject to roundoff error) should be tested against a reference using a threshold.

2.37.45 R45 : argument intents are compulsory

status : covered elsewhere

All dummy arguments must specify the INTENT attribute

2.37.46 R46 : order of optional arguments

status : covered elsewhere

Optional arguments must be called in the same order they are declared.

2.37.47 R47 : ENDIF / END IF

status : covered elsewhere

END statements for blocks should not have a space after END. For example an IF block should end with ENDIF, not "END IF".

2.37.48 R48 : removal of dead code

status : covered elsewhere

Inactive (e.g. commented-out) code must be removed.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search