Coding standards for Arpege/IFS/Aladin

R. El Khatib
METEO-FRANCE - CNRM/GMAP

November 19, 2003

Introduction

NWP software, which is often used for both operations and research, aims to be homogenous, portable, modifiable and maintained with much flexibility and ease.
This represents a lot of constraints, to be shared by all participants of the ARPEGE/IFS and ARPEGE/ALADIN projects, and even more when cross-collaborations with other projects are included.

While the computing environment of the ARPEGE/IFS software was originally a shared-memory, multi-processor vector machine, things have changed so that the computing environment is much wider: ranging from a single workstation to a cluster of high performance servers: scalar or vector processors, distributed memory or partially shared memory machines.

Last but not least, scientific and technical developments provide a strong justification for a perpetual evolution of the codes in order for the software to perform optimially, and this should be achieved using a minimum of time and human resources.

All these requirements need a set of agreed coding standards, that will be used by all participants of the project.

First of all we have to choose a language. But it is not enough to "just code in the same language". There are many ways to write the same program. Though different styles can give the same numerical result, they would not be equivalent once we consider further criteria telling that the software should be:

- well studied and well analysed
- well written (... but what does it means?!)
- properly documented
- portable
- efficient
- flexible
- possibly exchangeable

For a long time the best solution to achieve this was to use the so-called "DOCTOR norm" (DOCumentary ORiented norm) by J. K. Gibson from ECMWF. Time has proven its efficiency. Then the evolution of machines and languages as well as the increase of European collaborations has lead us to modify or extend the original norms. A European standard has been developed in order to facilitate the exchanges of code between meteorological organizations.

We can point out a few aspects of these norms:

- The existence of comments and especially comments at the start of each routine. Those concerning the modifications appear to be very useful to trace the history of the code. However, the 3-level structure to enable an automatic extraction of the comments never seems to have been used. Nevertheless an automatic extractor to document the ARPEGE/IFS/ALADIN namelist variables now exists.
- The structure in sections and sub-sections with homogenous labels improve the readability of the code, which is even more important than the norm itself.
- The convention of prefixes in order to rapidly identify the type (integer, real ...) and the nature (local or shared, dummy or not, ...) of the variables appeared to be the most popular aspect of the norm.

This document aims to collect all the conventions and customs used in the ARPEGE/IFS/ALADIN software. Hopefully it could be the starting point to code an automatic verificator of the norm. Finally only the norms tested in this automatic verificator would be the official norms while the others would be just recommendations. Developers should be aware that some rules could cause a lot of merging problems, especially where the same lines have been re-shuffled by more than one developers. Therefore we would not advocate to have all the standards systematically enforced by an automatic corrector.

In this document each item is referenced by a label composed of a topic and a number. There are four defined topics :

PRES for the presentation of the code,

NORM for the respect of the norm,

CTRL for the control of the code,

CCPT for the conception of the code.

This document is supposed to reflect the status of cycle 28 of the ARPEGE/IFS/ALADIN software.

Acknowledgements

Many thanks to François Bouttier, Claude Fischer, Mats Hamrud, Deborah Salmond, Eric Sevault, Yannick Tremolet and Karim Yessad for re-reading this document and fixing its "bugs", and for their interesting advice all over.

Contents

1 Specifications **8**
- 1.1 Documentation . . . 8
- 1.2 Code conception . . . 9
- 1.3 Code validation and maintenance . . . 9
- 1.4 Current code framework . . . 10

2 Design **12**
- 2.1 Typewriting style . . . 12
- 2.2 Basic layout . . . 12
  - 2.2.1 Executable statements . . . 12
  - 2.2.2 Comments . . . 13
  - 2.2.3 Entry point and exit points . . . 13
- 2.3 Header comments . . . 14
  - 2.3.1 Data modules . . . 14
  - 2.3.2 Procedures . . . 15
- 2.4 Declaring variables . . . 16
  - 2.4.1 Layout . . . 16
  - 2.4.2 Kinds . . . 18
  - 2.4.3 Specifications for data modules . . . 18
  - 2.4.4 Specifications for procedures . . . 18
  - 2.4.5 DOCTOR naming conventions . . . 19
  - 2.4.6 Further naming conventions . . . 20
- 2.5 General coding norms . . . 21
  - 2.5.1 Section comments and supplementary comments . . . 21
  - 2.5.2 Banned features . . . 21
  - 2.5.3 Loops . . . 22
  - 2.5.4 Conditional blocks . . . 22
  - 2.5.5 Linebreaking . . . 22
  - 2.5.6 Dynamic memory usage . . . 24
  - 2.5.7 Symbolic comparison operators . . . 24
  - 2.5.8 Fortran 90 intrinsic functions and procedures . . . 25
  - 2.5.9 Fortran 90 array syntax . . . 25
  - 2.5.10 Dummy and actual arguments . . . 26
- 2.6 Specific coding norms . . . 27
  - 2.6.1 Naming modules, procedures, namelists and derived types . . . 27
  - 2.6.2 Error handling . . . 30
  - 2.6.3 "Hook" function . . . 30
  - 2.6.4 Handling universal constants . . . 30
  - 2.6.5 Purpose and usage of the key `LECMWF` . . . 31

2.6.6 Purpose and usage of the key `LELAM` 31
2.6.7 Purpose and usage of the key `LRPLANE` 31
2.6.8 Model settings 31
2.6.9 Output messages 31
2.6.10 I/O raw data 32
2.6.11 Message passing interface 32

3 Source code management **34**

Index **40**

List of Figures

2.1 Examples of entry/exit points and errors handling. 14
2.2 Example of data modules. 15
2.3 Example of header documentation and variables declarations in a procedure. . . . 17
2.4 Conventional kind parameters for integers and reals. 18
2.5 Naming conventions. 19
2.6 Example of local versus global variables. 20
2.7 Example of computations in a procedure. 23
2.8 Fortran 90 specific comparison operators. 24
2.9 Some of the predefined functions or procedures specific to Fortran 90 25
2.10 Example of dummy arguments handling 28
2.11 Conventional prefixes and suffixes. 29

Chapter 1

Specifications

A well-thought out program is less difficult to code, produces fewer bugs and is often easier to maintain. So it is essential to specify the work with care. Three important aspects should be considered: the documentation, the code conception and its further enhancements, the code validation and maintenance.

1.1 Documentation

Documentation is essential: modification and maintenance will be much easier if everyone can understand not only the code, but also the design spirit behind the code. When changing the code the documentation should be updated immediately to avoid misunderstandings.
For the international cooperation to work, the documentation should be written in English.

There should be two kinds of documentation:

An external documentation which will be provided for a package of subroutines rather than an individual one. It should be written outside the code and divided into three parts[1]:

1. *Scientific documentation* describing the scientific aspects of the problem and the solution adopted in the current software. **This documentation should not refer to the code itself.**

2. *Technical documentation* describing the implementation of the solution adopted in the scientific documentation. This documentation should include a calling tree and a description (name, purpose) of all the modules which are used (subroutines, functions, data modules). This documentation should take over from the scientific one when technical aspects are concerned. Information for testing, modifying and maintaining the code should be included inside such documentation.

3. *A users guide* describing the user interface, switches and tunable variables of the software (access, default values and range).

Internal documentation which should be provided for individual modules (data modules or procedures). This documentation can be divided into three categories:

1. *Header comments* stating briefly the purpose of the module, the author, references to external documentation, list of modifications (author and purpose) since the creation of the module. When dummy arguments are used, this header should describe them.

[1] European standards, 1995

2. *Section comments* splitting the code into logical sections (that may be related to the scientific documentation). They indicate, section by section, the purpose of the code.

3. *Supplementary comments* which should help reading the code. There should not be many of them: source code which is interspersed with many comments is difficult to follow and to understand. If the code has been well designed and if the related external documentation has been properly written before coding, then the header and section comments should be sufficient.

Both kinds of documentation should be updated at the same time in order to avoid misunderstandings.

1.2 Code conception

- At the design stage, one should consider how the system should be tested, how it could be modified later and how it will be maintained. Future objectives of the system — not only the immediate objectives — should be considered: future enhancements should be anticipated and planned if possible, and should be made possible with a minimum of disturbance to the whole system.

- The different parts of the system should be analysed and planned. The parts should be designed in a modular way, with interaction between them based on a hierachical and tree-like structure. There should not be any duplication of code: neither real duplications (when a piece of code is copied then pasted) nor virtual duplications (when more than one procedure has the same purpose). Duplication of code increases the problems of maintenance. Whenever code duplication is found the incriminated part of code should be re-designed.

- The relationship between modules should be simple. Individual modules should not be complex. Whenever a module is becoming complex after enhancements, the system should be re-examined and the modularity re-designed. The longer subroutines are, the less readable they are and the more difficult they are to maintain.

- Derived types should be used where appropriate as that they make the code more robust and easier to maintain. Derived types naturally contribute to a more object-oriented code.

- Dataflow is a recurrent problem whenever portability is concerned. Therefore special attention should be given to it. Data inputs, outputs or transfers should be confined to a set of data handling modules, separated from the application modules.

- Non-standard statements of the language should not be used. In case they have to be, they should be confined into a subset of modules to limit the problems of portability. The same rule should apply to the invocation of routines coming from an external package.

1.3 Code validation and maintenance

- While validating and evaluating new code, the following questions should be considered:
  - Does the source code comply with the coding standards?
  - Is the code easy to understand?
  - Is the code unnecessarily complex?
  - Is the interface to the subroutine straightforward?

- Does the routine produce the expected results?
- Can modifications be made easily if required?
- Are ALL error cases detected and properly acted upon?
- Are ALL aspects of the calculation (ie: direct model, tangent linear, adjoint, limited area model versus global, ECMWF versus METEO-FRANCE setups) properly treated?
- Is the routine efficient in terms of memory and CPU consumption?
- Are the final integrated tests (ie: the operational configurations of the software) sucessful?

- The primary maintenance documentation is the source code. Only the source code is guaranteed to be up to date. Thus, **it is of vital importance to update source code comments while modifying the source code.**
- When code undergoes development for scientific and technical reasons, at some point it is likely to become unecessarily complex. It is important to recognise this and when it occurs review and if necessary re-design and re-write the code package concerned.

1.4 Current code framework

- The ARPEGE/IFS/ALADIN code is written in Fortran 90 and C. The following document mainly applies to the part of the code written in Fortran, this being the main coding language in which the bulk of the code is written.
- The code is designed to perform well on both vector and cache based processors. This feature has to be maintained for the foreseable future.
- It is parallelised for distributed memory computers using MPI.
- It is also parallelised for shared memory computers using OpenMP inside MPI. This implies that the code inside the OpenMP regions has to be written in a thread-safe way.

Chapter 2

Design

2.1 Typewriting style

Following the European standards[1] (1995) which are more restrictive than the ANSI standard the Fortran keywords may be written in upper case only, or with initial letter in upper case and the rest in lower case. The names of variables may be written in mixed lower or upper case and the names of namelists, modules, programs or subroutines may be written in mixed lower or upper case. No recommendation has been made concerning comments.
However the survey of the existing code shows that the whole executable lines are preferably written with upper case characters. Comments are preferably written with lower case characters except the first letter, or the first letter after each full stop. Emphasized words are written in upper case characters and bracketed with asterisks[2].
It is recommended to stick to a conventional typewriting style inside the whole code because this convention enables the developers to concentrate upon the semantic of the code and makes easier the use of automatic tools to manipulate the code.[3]

In view of the apparent lack of rule concerning the typewriting style, the recommendation is to stick to the apparent habit:

PRES(01) Executable lines should be written using upper case characters.

PRES(02) Comments should be written with lower case characters except the first letter, or the first letter after each full stop. Emphasized words may be written in upper case characters.

The minimum recommendation would be to follow a consistent style throughout each module or subroutine.

2.2 Basic layout

2.2.1 Executable statements

NORM(01) The code should be written in Fortran 90 free format, at least as far as science is concerned.
The use of Fortran ("FORmula TRANslator") is almost obligatory since:

[1] European standards for writing and documenting exchangeable Fortran 90 code
[2] Apparently the heritage of an automatic documentation extractor.
[3] Hill, 1995

- it fits the scientific topics
- it is portable
- it is well-known by most of the developers !
- there exist well optimised Fortran compilers for vector and scalar processors.

C language can advantageously be used for low-level subroutines.

PRES(03) The code should start at the column 1, unless it comes to any of the indentation norms as they will be described below.

PRES(04) The ending statement of a module or subroutine should repeat its name. For example: "`SUBROUTINE SUCTO ... END SUBROUTINE SUCTO''`

NORM(02) Tabulations must not be used: this ensures the code will look indented as desired whenever ported (also the use of the tab character is non ANSI).

PRES(05) One should avoid writing more than one statement per line (ie: avoid using the separator ";"). More than one statement per line might penalize the readability. If several statements should be grouped together then one may write a tiny subroutine that would be private and contained in the current subroutine.

CCPT(01) Subroutines should not have more than 300 executable statements[4] (there are projects which are even more strict[5]). Each subroutine should have a maximum score of 400, based on the following measure[6]:
- each subroutine call has score 5
- other executable lines have score 1

There is no recommendation for a minimum score per subroutine.

CCPT(02) When the code is modified, it is easier to add or remove lines than modify existing ones. This is of special importance when merging code modifications from several developers. Therefore one should write the code in such a way that consecutives lines are as independent as possible. This would make the future merge of source code easier. Unfortunately this can make the code unecessarily long. So this rule should be applied with care. Obviously it fits short codes.

2.2.2 Comments

PRES(06) The comments should be written in English. They should not be written twice (for example: English and the programmer's native language), because it makes the code less readable.

PRES(07) Blank lines should remain empty: they should not start with "`!`"

2.2.3 Entry point and exit points

CTRL(01) Each procedure should contain one entry point and at most two kinds of exit points: one normal return and one abnormal termination.

CTRL(02) The entry point should be at the top of the procedure.

[4] Gibson, 1986
[5] Hill, 1995
[6] Gibson, 1986

CTRL(03) The normal return should be the bottom of the procedure. Consequently the use of the statement `RETURN` is discouraged. If there is only one `RETURN` statement at the end of a procedure it should be removed.

CTRL(04) Abnormal termination should be invoked with the specific subroutine `ABOR1` because this subroutine enables one to flush the output buffer and to release the processors which are not affected by the abnormal termination.

There can be more than one abnormal termination in the body of the subroutine.

Figure 2.1 shows examples of entry/exit points.

Figure 2.1: Examples of entry/exit points and errors handling.

```
SUBROUTINE ERROR_DETECTION
   .
   .
USE MHOOK    , ONLY : LHOOK
   .
   .
IF (LHOOK) CALL DRHOOK('ERROR_DETECTION',0)
   .
   .
IF (IWORD /= ILEN) THEN
  CALL ABOR1 ('ERROR_DETECTION : MESSAGE 1 RECEIVED WITH WRONG LENGTH')
ENDIF
   .
   .
   .
IF (IERR > 0) THEN
  CL='MESSAGE 2 RECEIVED WITH WRONG LENGTH'
  WRITE(NULOUT,*) CL
  WRITE(NULERR,*) CL
  CALL ABOR1 ('FROM ERROR_DETECTION')
ENDIF
   .
   .
   .
IF (LHOOK) CALL DRHOOK('ERROR_DETECTION',1)

END SUBROUTINE ERROR_DETECTION
```

2.3 Header comments

2.3.1 Data modules

CCPT(03) Each data module should begin with documentation describing the general content of the module and the purpose of each declared variable.

PRES(08) In order to improve the readability, the namelist variables in a data module should be separated from the internal ones.

PRES(09) Each description line should be independent to enable an automatic extraction of the documentation.

PRES(10) The documentation should be separated from the starting statement with a blank line and it should finish with a comment line filled with minus signs.

Figure 2.2 shows an example of a data module.

Figure 2.2: Example of data modules.

```
MODULE YOMDATA

! Module showing the coding standards in Arpege/Ifs/Aladin.

!   NUMBER : Key telling what to do :
!            NUMBER = 0 => don't do anything
!            NUMBER = 1 => do this
!            NUMBER = 2 => do that
!   VALUE  : Tunable variable to do what you wish
!   ARRAY  : Mysterious data array

!-----------------------------------------------------------------------------

USE PARKIND1 , ONLY : JPIM     ,JPRB

IMPLICIT NONE

SAVE

INTEGER(KIND=JPIM)            :: NUMBER
REAL(KIND=JPRB)               :: VALUE
REAL(KIND=JPRB), ALLOCATABLE :: ARRAY(:,:)

!-----------------------------------------------------------------------------

END MODULE YOMDATA
```

2.3.2 Procedures

PRES(11) Each procedure should begin[7] with a documentation header as a set of comments containing:

- the *purpose* of the procedure

[7] European standards, 1995.

- the *interface* details, describing the dummy arguments in the same order as they are in the interface
- the *externals* or other subroutines called
- the *method* used in the application, where there is no further documentation to refer to.
- a *reference* to further documentation
- the *author and date* of creation of the procedure
- the *modifications* applied since the creation of the procedure, with the author and date of modifications

PRES(12) The header documentation should be separated from the entry point statement with an empty line and it should finish with a comment line filled with minus signs.

PRES(13) The *modifications* comments should start with the template: "`! Modifications`" and should end withe the template: "`! End Modifications`".
In between all the modifications description should be written in the same style:
day(2 digits), month(3 characters), year(four digits) separated with a minus sign, then the author, then a description.

Figure 2.3 shows a header documentation for a procedure.

2.4 Declaring variables

2.4.1 Layout

NORM(03) The use of `IMPLICIT NONE` statement is mandatory. It improves the portability of the code and helps in the detection of errors.

NORM(04) Hard-coded variables increase the problem of maintenance and can even be the cause of bugs, especially when they are used in a subroutine interface.
Hence it is much better to write: `CALL POSNAM(NULNAM,CLNAME)` than
`CALL POSNAM(4,'NAMCTO')`

PRES(14) The declaration of variables should be separated from the header documentation with an empty line. It should finish with a comment line filled with minus signs.

PRES(15) The declarations of variables should be grouped according to their type and attributes.

NORM(05) The *statement* `DIMENSION` should not be used (but the *attribute* `DIMENSION` can be). The shape and size of arrays should be declared inside brackets after the variable name on the declaration statement.

NORM(06) The notation "`::`" should be systematically used after the type and attribute declaration, and before the name of the variable.

PRES(16) All the attributes of a given variable should be grouped within the same instruction. This makes it possible to visualize in a glance the characteristics of a variable or of a family of variables.

Figure 2.3: Example of header documentation and variables declarations in a procedure.

```
SUBROUTINE CODESTY(KERR)

! Purpose :
! -------
!   *CODESTY* : CODE STYle : To show coding standards in Arpege/Ifs/Aladin.

! Interface :
! ---------
!   KERR   : Output error code of the subroutine

! Externals :
! ---------
!   None.

! Method :
! ------

! Reference :
! ---------
!   Coding standards in Arpege/Ifs/Aladin.

! Author :
! ------
!   19-Jul-2002 Ryad El Khatib        *METEO-FRANCE*

! Modifications :
! -------------
!   30-Oct-2003 M. Hamrud             Cleaning for Cycle 28
!   30-Feb-0000 R. Randriamampianina  Imaginary modification ;-)
! End Modifications
!-------------------------------------------------------------------------------

USE PARKIND1 , ONLY : JPIM

USE YOMDATA  , ONLY : NUMBER   ,VALUE

IMPLICIT NONE

INTEGER(KIND=JPIM), INTENT(OUT) :: KERR

INTEGER(KIND=JPIM), PARAMETER :: JPLEN=16  ! Length of the local message
CHARACTER(LEN=JPLEN) :: CLMESS    ! A local message
!-------------------------------------------------------------------------------
   .
   .
END SUBROUTINE CODESTY
```

PRES(17) Templates like "`Dummy scalar arguments :`", or "`Local integer arrays :`", etc. on top of any group of variables declarations are not necessary: the Fortran attributes declarations, if used as recommended in this document, are self-documenting. Also the complete list of such templates is so wide that using them can make the code less readable.

CCPT(04) Actually unused variables (local in a used module) should be removed from the current procedure : this makes the code clearer and can reduce the dependencies complexity.

2.4.2 Kinds

NORM(07) Variables or constants are preferably declared with explicit kind.
In practice conventional parameters have been defined for various kinds (see modules `PARKIND1` and `PARKIND2`): refer to figure 2.4.

Figure 2.4: Conventional kind parameters for integers and reals.

`KIND` value	`KIND` parameter
`SELECTED_INT_KIND(2)`	`JPIT`
`SELECTED_INT_KIND(4)`	`JPIS`
`SELECTED_INT_KIND(9)`	`JPIM`
`SELECTED_INT_KIND(12)`	`JPIB`
`SELECTED_INT_KIND(18)`	`JPIH`
`SELECTED_REAL_KIND(2,1)`	`JPRT`
`SELECTED_REAL_KIND(4,2)`	`JPRS`
`SELECTED_REAL_KIND(6,37)`	`JPRM`
`SELECTED_REAL_KIND(13,300)`	`JPRB`
`SELECTED_REAL_KIND(28,2400)`	`JPRH`

Refer to figure 2.7 for an example of constants usage.

2.4.3 Specifications for data modules

CTRL(05) In data modules all variables should be saved in order to preserve their values. This is achieved by the use of the statement `SAVE`.

PRES(18) Each variable should be declared separately.

Refer to Figure 2.2 for an example of declarations in a data module.

2.4.4 Specifications for procedures

NORM(08) The variables should be used or declared in the following order:

1. The variables used from modules (this is enforced by Fortran standard)
2. the dummy arguments
3. the local variables

NORM(09) When using a data module, the resources should be restricted to the actually used variables in order to avoid latent conflicts. This is achieved by the use of the keyword `ONLY`.

PRES(19) While enumerating the used variables, the items should be regularly spaced in order to respect a general alignement: this improves the readability. The space used is currently 9 characters per variable, but it could be a multiple of 9 to preserve the general alignment when long names are used. If lines should be broken the separating commas should be at the end of the lines, not the beginning of them.

PRES(20) The declaration of dummy arguments and the presentation of the dummy arguments in the subroutine interface should be the same, in order to improve the readability.

PRES(21) If lines should be broken in the `SUBROUTINE` variable lists then the separating commas should be at the end of the lines, not the beginning of them.

PRES(22) New lines in the `CALL` variable lists should be the same as "line breaks" in the subroutine arguments list.

Refer to Figure 2.3 for examples of variables declarations in a procedure.

2.4.5 DOCTOR naming conventions

The purpose of the following naming conventions is to convey, through the use of prefix letters, the type, status and scope of all variables within the program. Since the original definition of the DOCTOR system, a few minor changes have been made to reflect:

- the increase use of facilities of Fortran, especially `CHARACTER` type
- the rationalization in the light of experience
- the wish to restrict prefixes to a single letter as far as possible.

The use of a prefix convention to indicate the status or scope of the variable enables differentiation at a glance.

NORM(10) The type of a variable is indicated by the first — or first two — letter(s) which compose(s) its name, according to figure 2.5.

Figure 2.5: Naming conventions.

Status or scope / Type	Variable in data module	Dummy argument	Local variable	Loop control	Any Parameter
`INTEGER`	`M, N`	`K`	`I`	`J` but not `JP`	`JP`
`REAL`	`A, B, E` to `H, O, Q` to `X`	`P` but not `PP`	`Z`	-	`PP`
`LOGICAL`	`L` but not (`LD,LL,LP`)	`LD`	`LL`	-	`LP`
`CHARACTER`	`C` but not (`CD,CL,CP`)	`CD`	`CL`	-	`CP`
Derived type	`Y` but not (`YD,YL,YP`)	`YD`	`YL`	-	`YP`

Note:

NORM(11) Double precision variables, which are prefixed with `D` according to the DOCTOR norm, are no more used in the current software: instead of that, the type and kind of the variables are declared explicitly.

NORM(12) Elementary variables composing a derived type should follow the naming conventions for local or global variables.

2.4.6 Further naming conventions

Names of variables should be as meaningful as possible.

At the time the DOCTOR norm was first specified, the distributed memory machines were not in operations. While programming on a distributed memory machine, the developers have to consider a new scope of variables:

- those which are *local* in the sense of the distribution: such variables can be shared by several subroutines and so they can be declared in a data module. But their values may differ between processors.
- those which are *global* in the sense of the distribution: such variables can be local to a subroutine. But they have a *physical* meaning so that their values will be the same on all processors.

Concerning this scope the naming convention is the following:

CCPT(05) Variables which are suffixed with the letter `L` are local in the sense of the distribution.

CCPT(06) Variables which are suffixed with the letter `G` are global in the sense of the distribution.

Note: the reverse assumption is not true, that is: variables which are local in the sense of the distribution are not always suffixed with `L` (actually *all* variables are local in the sense of the distribution). In the same way variables which are global in the sense of the distribution are not always suffixed with `G`. This is justified since the distribution concerns the data and not all the applications: the mentioned rule applies only to those variables related to the dimensions concerned by the distribution. For instance the model time step is not governed by this rule. Also a loop index would not be governed by this rule, while the loop bounds could be.

Figure 2.6 gives an example of such variables.

Figure 2.6: Example of local versus global variables.

	Variable local to a subroutine	Variable in data module
Local variable in the sense of the distribution	A loop index	The number of gridpoints treated by a processor
Global variable in the sense of the distribution	A value gathered among all processors	The total number of gridpoints in the model

2.5 General coding norms

2.5.1 Section comments and supplementary comments

PRES(23) The body of the code should be split into numbered sections and subsections. The numbering should be so that the M^{th} subsection of the N^{th} section would be labelled `N.M`

PRES(24) Each *section* should be clearly separated from the previous one and should begin with its section number and an underlined title

PRES(25) Each *subsection* should be clearly separated from the previous one and should begin with its subsection number and title

PRES(26) Supplementary comments should be placed either immediately before or on the same line as the code they are commenting.

2.5.2 Banned features

Several Fortran features should not — or no more — be used, as their past usage showed their detrimental effect in programming, or because they are becoming obsolescent and thus can disappear in future versions of the compilers.

NORM(13) `GO TO` should not be used because it is detrimental to the readability and is obsolescent. Fortran 90 provides instructions like `DO WHILE`, `EXIT`, `CYCLE` and the conditional block `SELECT CASE` which can replace `GO TO`.

NORM(14) `FORMAT` statement should not be used any more as it is becoming obsolescent. Format descriptors should be used instead. For example, one can replace:

```
WRITE(*,99) 'Hello !'
99 FORMAT(A7)
```

by

```
CLFMT='(A7)'
WRITE(*,CLFMT) 'Hello !'
```

NORM(15) `COMMON` should not be used. `MODULE` should be used instead, because it is a more robust, flexible statement.

NORM(16) `EQUIVALENCE` should not be used because it may cause problems of readability or sometimes portability. `POINTER` or `TYPE` data can replace it.

NORM(17) `COMPLEX` type should not be used since the resulting code is not efficient[8].

CTRL(06) One should not implicitly change the shape of an array while passing it into a subroutine, because this works only after assumptions about how the data is stored. In such situation the code should be properly re-written. If this is not possible `RESHAPE` should be used instead, but this statement involves extra cost.

CTRL(07) One should not implicitly change the type of a variable while passing it into a subroutine, because this works only after assumptions about how the data is stored. In such situation one should use `TRANSFER` instead.

NORM(18) To declare a character string, the syntax `CHARACTER*`n should no more be used because it is becoming obsolescent. Hence the syntax should be: `CHARACTER(LEN=`n`)`.

[8] Clochard, 1988

NORM(19) Arrays should not be declared with implicit *size*:
like "`REAL(KIND=JPRB) :: A(*)`"
but they may be declared with implicit *shape*:
like "`REAL(KIND=JPRB) :: A(:)`"
Note that such declaration requires an interface block.

2.5.3 Loops

NORM(20) One should use only the "block loop" construct, ie starting with `DO` and ending with `ENDDO`. Loop boundaries should stand out, finishing with `ENDDO` statement, in order to make future modifications inside the loop easier.

PRES(27) `DO` and `DO WHILE` loops should be indented with 2 blank spaces to improve the readability.

PRES(28) In case of complex loops nesting, it is recommended to use a character label for each loop.

CCPT(07) Loops should be as plain as possible: complexity may destroy the vectorization of the loop.

Figure 2.7 shows indentations for loops.

2.5.4 Conditional blocks

CTRL(08) Use the `SELECT CASE` statement when possible, rather than `IF/ELSEIF/ELSE/ENDIF` statements because the condition relies on the value of only one expression which is compared to constant values, and thus overlapping values can be detected at compilation time.

PRES(29) Conditional blocks should be indented with 2 blank spaces to improve the readability.

PRES(30) Nesting of conditional blocks should not be more than 3 levels deep: deeper nesting destroys the understandability of the code[9]. In case of complex nesting, it is recommended to use a character label for each elementary blocks.

PRES(31) Conditional block boundaries should stand out, in order to make future modifications below a condition easier. However, if the conditional instruction is nothing but a plain branch like `EXIT` or `CYCLE` then this recommendation may be ignored.

Refer to Figure 2.7 for examples of usage of conditional blocks.

2.5.5 Linebreaking

PRES(32) Though Fortran 90 allows up to 132 characters on a line, the length should be limited to 80 characters per line in order for the code to be viewed easily on any terminal, or to be easily read, when printed on A4 paper.

NORM(21) The continuation character `&` should appear both at the end of each line to be continued and at the beginning of each continuation line. In this way we have a systematic rule which allows the inclusion of blank space.

[9] Gibson, 1986

Figure 2.7: Example of computations in a procedure.

```
SUBROUTINE COMPUTE
   .
   .
   .

! 1. Initialization
!    --------------

IST   = LBOUND(ZA)
IEND  = UBOUND(ZA)
ZSCAL = 5._JPRB

ZB(:) = 2._JPRB
ZC(:) = 4._JPRB


! 2. Computation and selection
!    -------------------------

DO JI=IST,IEND
  IF (LDONE(JI)) CYCLE
  ZA(JI) = ZB(JI) + ZC(JI)
  ZD(JI) = 1.0_JPRB-LOG(ZA(JI))
  ZE(JI) = ZSCAL*ZB(JI) &
   &     + (ZA(JI)-ZD(JI))
ENDDO

SELECT CASE (NOPTION)
  CASE(1:)
    SELECT CASE (ALL(LDONE))
      CASE(.FALSE.)
        CALL ROUTINE(        &
         & NOPTION,IST,IEND, &
         & ZA,ZB,ZC,ZD,ZE)
    END SELECT
  CASE DEFAULT
    CALL ABOR1('COMPUTE : ILLEGAL VALUE NOPTION')
END SELECT

CALL MPL_BARRIER(CDSTRING='COMPUTE:')
   .
   .
   .
END SUBROUTINE COMPUTE
```

PRES(33) The continuation lines should be indented with one supplementary blank space to improve the readability.

PRES(34) Lines should be broken in a readable manner (ie: do not break a variable name). It is better to start a continuation line with an operator rather than to end one with an operator.

PRES(35) The continuation characters should be aligned on the same columns to improve the readability.

Refer to Figure 2.7 for examples of linebreakings.

2.5.6 Dynamic memory usage

CCPT(08) The use of dynamic memory (automatic or explicit allocation) is preferred to static one (arrays dimensioned with `PARAMETER` statement) because:

- it enables the re-use (and thus the saving) of memory
- it enables the same executable file to be run for different resolutions (which is basically the configuration of a multi-incremental 4D-var assimilation)

However, to avoid potential memory inefficiency, further recommendations should be considered while using dynamic memory allocation:

NORM(22) Automatic arrays should be preferred to explicitly allocated arrays/pointers (except for very large arrays[10]) because they are automatically released at the end of the subroutine they are declared in.

NORM(23) Local arrays allocated explicitly in a subroutine must be explicitly deallocated before leaving the subroutine

CCPT(09) One should not repeat sequences like:
`ALLOCATE`, `DEALLOCATE`, `ALLOCATE` again ...
many times: it is better to compute the maximum size of the array and allocate it once.

2.5.7 Symbolic comparison operators

NORM(24) The Fortran 90 specific comparison operators should be used because since this syntax is closer to the mathematical notation the resulting code should be more readable. Figure 2.8 lists them.

Figure 2.8: Fortran 90 specific comparison operators.

Equal	Not equal	Less than	Greater than	Less equal	Greater equal
==	/=	<	>	<=	>=

NORM(25) The operators `==` and `/=` should not be used to compare real variables because the result depends of the precision of the machine. This kind of comparison should be used only when absolutely necessary.
Instead of: `( Z1 == Z2 )` one should write: `( ABS(Z1-Z2) < ZSCAL*SPACING(Z1) )`
where `ZSCAL` is a scaling factor greater than 1.

[10] Huge automatic arrays can break the stack limit

2.5.8 Fortran 90 intrinsic functions and procedures

The Fortran 90 language provides a large number of predefined functions or procedures. These can make the code shorter, more readable, more portable and sometimes more efficient. Figure 2.9 recalls several of these functions and their behaviors for zero-element arrays.

Figure 2.9: Some of the predefined functions or procedures specific to Fortran 90

Function	Purpose	Behavior for zero-element arrays
`ADJUSTL`	To adjust a string on the left side without leading blank (not leading "space") characters	normal
`ADJUSTR`	To adjust a string on the right side without trailing blank (not trailing "space") characters	normal
`ALL`	To find out if all the values of an array are `.TRUE.`	`.TRUE.`
`ANY`	To find out if any value of an array is `.TRUE.`	`.FALSE.`
`COUNT`	To count the number of true elements in an array	0
`DOT_PRODUCT`	Scalar product of two vectors	0
`EPSILON`	Precision of the machine	normal
`HUGE`	largest number of the machine	normal
`MAXLOC`	To localize the maximum value in an array	0
`MAXVAL`	To find out the maximum value in an array	less equal (- `HUGE`)
`MINLOC`	To localize the minimum value in an array	0
`MINVAL`	To find out the minimum value in an array	`HUGE`
`RESHAPE`	To reshape an array	Possible error
`SHAPE`	Shape of an array	0
`SIZE`	Size of an array	0
`SUM`	To sum the content of an array	0
`SYSTEM_CLOCK`	To get information from the system clock	-
`TINY`	Smallest number of the machine	normal
`TRANSFER`	To tranfer a variable into another type	?
`TRIM`	To remove the trailing blank (not the trailing "space") characters of a string	Error

NORM(26) *generic* names should be used for intrinsic procedures, not specific names.

2.5.9 Fortran 90 array syntax

Fortran 90 array syntax makes the code more compact and sometimes more readable, but in most cases the result is slower, or at least not faster than the Fortran 77 style do loops.
The reason is the compiler's inability to fuse several array statements and re-use common sub-expressions, registers, etc. With the current level of maturity of Fortran 90 compilers there is no reason to believe that the situation will improve dramatically in the future.
Therefore :

CCPT(10) The use of array syntax is not recommended, except for simple operations, like initializing or copying whole arrays.

Refer to Figure 2.7 for examples of recommended computations for arrays. Note that in the F90 style, `ZA(:)` has a precise meaning: it means that we consider the whole array. The lower and upper bound are then respectively `LBOUND(ZA)` and `UBOUND(ZA)`.

2.5.10 Dummy and actual arguments

In Fortran 90 there are two ways of of associating arguments when a subroutine is called, the Fortran 90 way and for compatibility the Fortran 77 way [11]. The main difference lies in the way arrays are passed, in the Fortran 90 way it is by strict type, kind, rank and extent matching whereas in the Fortran 77 way it is done by Array Element Sequence association. It is important to know that the Fortran 90 way is only used when you have an explicit interface block and the arrays are declared with assumed shape.
The Fortran 90 way of passing arguments is much more secure, the compiler will detect any mismatch between actual and dummy arguments thanks to the explicit interface block. Unfortunately the use of explicit interface blocks/module procedures is still very limited within the ARPEGE/IFS/ALADIN code, one reason being that it introduces more dependencies between separately compiled units and thus increases the complexity and possibly cost of the compiling system.

CTRL(09) When an explicit interface block is being used for a routine, the interface body should be in an independent separate file (ie: starting with `"INTERFACE"` and ending with `"END INTERFACE"` and introduced in the calling routine with an `#include` statement. The interface body should be extracted from the routine itself by an automatic procedure to ensure that they conform.

NORM(27) The `INTENT` attribute should be used for all dummy arguments: this improves the auto-documentation and the security of the code.

CTRL(10) The number of dummy arguments should be kept as small as possible. As excessive number of arguments degrades the readability and increases the problems of maintenance whenever arguments are added or removed.
To retain the modularity of subroutines there are alternatives:

- to re-design a set of elementary arguments as a new derived type.
- to identify the arguments which are internal to a set of subroutines and to use them via a data module. Care has to be taken that this does not cause problems with the thread safeness of the code.

The standard should be: the number of dummy arguments should not exceed 9.

CTRL(11) The arguments of a subroutine should be presented following a conventional order because this improves the readability, the maintainability and sometimes also the portability (the dimensioning of dummy arguments should appear ahead in order to improve the portability). For the time being such a rule has been applied only for the physical package of Météo-France, with the convention: input, then input/output, then output arguments. Other orders can be considered, for instance: to order the arguments according to their types and attributes, including the `INTENT` attribute. Concerning the tangent linear and adjoint subroutines the initial recommendation was to follow the order of arguments as in the direct code, then to add the trajectory variables in the same order. Such rule can conflict with other general norms.

[11] Adams et.al.,pp 509-529

Finally the achievable norm in this context seems to be: the arguments of a subroutine should be ordered *at least* with integer scalar first.

NORM(28) The preferred method for passing array subsections is to use an explicit interface block. This method allows array sections to be passed safely with no extra cost. If instead an array section is passed when using the "Fortran 77" way of passing arguments extra copying will take place before and after the subroutine call (the compiler will generate the code) incurring extra cost. If the bounds of the array section is specified wrongly this copying may also cause memory overwriting. Thus when using the "Fortran 77" way of passing arguments, when the section of the array you want to pass is contigous in memory and there is no explicit interface block declared, one should pass the start adress of the section e.g. `ZARG(1,2)` rather than the array section - `ZARG(:,2)`. Passing array sections, not contigous in memory, should be completely avoided when using the "Fortran 77" way of passing arguments.
However the use of an explicit interface block with assumed shape arrays can cause memory overwriting too, because the compilers become unable to check bounds. Thus when using assumed shape arrays one should take care that the array subscript never goes out of the interval given by the function `LBOUND` and `UBOUND`.

NORM(29) Use of array sections (using F90 array syntax) is encouraged when calling intrinsic routines, as all intrinsic F90 subroutines have explicit interfaces. This makes the code more readable and enables the compiler to check bounds properly.

Figure 2.10 illustrates the handling of dummy arguments.

2.6 Specific coding norms

2.6.1 Naming modules, procedures, namelists and derived types

Names of modules, procedures, namelists or types should be as meaningful as possible.

CTRL(12) Conventional prefixes or suffixes are recommended for names. Refer to figure 2.11.

Note: additive standards concern the radical of names:

CTRL(13) The radical of a type definition module should be the name of the type it defines[12]. For instance the type `TYPE_GFLD` is defined in the module `TYPE_GFLDS`

CTRL(14) The radical of a procedure module should be the name of the procedure it encapsulates. For instance the module `SUPOL_MOD` encapsulates the procedure `SUPOL`.

CTRL(15) For a subroutine in the spherical geometry of ARPEGE/IFS, its counterpart subroutine in the toroïdal geometry of ALADIN should have the same name prefixed with an `E`.

CTRL(16) Considering a namelist, its content should be saved in a specific data module and initialized in a specific subroutine. All three should be named with the same radical. For example: the content of the namelist `NAMCT0` is saved in the module `YOMCT0` and initialized in a subroutine `SUCT0`.

[12] ambiguous if there are more than one type defined in the module

Figure 2.10: Example of dummy arguments handling

```
SUBROUTINE PROCEDURE(KLEN,PIN1,PIN2,PIN3,PIN4,PIN5,PIN6,PIN7,POUT1,POUT2)

  .
  .
  .
USE PARKIND1  , ONLY : JPIM     ,JPRB

IMPLICIT NONE

INTEGER(KIND=JPIM), INTENT(IN)  :: KLEN
REAL(KIND=JPRB),    INTENT(IN)  :: PIN1(KLEN), PIN2(KLEN), PIN3(KLEN)
REAL(KIND=JPRB),    INTENT(IN)  :: PIN4(KLEN), PIN5(KLEN), PIN6(KLEN)
REAL(KIND=JPRB),    INTENT(IN)  :: PIN7(KLEN)
REAL(KIND=JPRB),    INTENT(OUT) :: POUT1(KLEN), POUT2(KLEN)

INTEGER(KIND=JPIM), PARAMETER :: JPNAME=9
INTEGER(KIND=JPIM), PARAMETER :: JPMESS=30
CHARACTER(LEN=JPNAME), PARAMETER :: CLNAME='PROCEDURE'
CHARACTER(LEN=JPMESS) :: CLMESS=' : INVALID NUMBER OF ARGUMENTS'
!-----------------------------------------------------------------------

! 1. Computation
!    -----------
  .
  .
  .
END SUBROUTINE PROCEDURE
```

Figure 2.11: Conventional prefixes and suffixes.

Prefix	*Entities*	*Suffix*
TYPE_	Types names	
TYPE_	Types definitions modules	S
PAR	Parameters modules	
YOE	Data modules specific to ECMWF physics	
QA	Data modules specific to CANARI	
YEM	Data modules specific to ALADIN	
TPM_	Data modules specific to spectral transforms packages	
MPL_	Data modules specific to MPL (message passing) package	
YOM	Data modules not specific to ECMWF physics, CANARI, ALADIN, spectral transforms or MPL package	
	Procedure modules	_MOD
NAE	Namelists specific to ECMWF physics	
NEM	Namelists specific to ALADIN	
NAC	Namelists specific to CANARI	
NAM	Namelists not specific to ECMWF physics, ALADIN or CANARI	
SUEC	Setup procedures specific to ECMWF physics	
SUE, not SUEC	Setup procedures specific to ALADIN	
SU, not SUE	Setup procedures not specific to ECMWF physics or ALADIN	
SL	Calculation procedures for any horizontal interpolations system	
LA	Calculation procedures specific to the semi-lagrangian scheme	
AC	Calculation procedures specific to ARPEGE/ALADIN physics ("Arpege Calcul")	
PP	Calculation operators for the post-processing or the analysis	
FP	Procedures specific to FULLPOS	
CA	Procedures specific to CANARI	
FA	Procedures specific to the Files Arpege package (FA)	
LFI	Procedures specific to the Indexed Files Library (LFI)	
MPL_	Procedures specific to the Message Passing Library (MPL)	
SI	Procedures specific to the semi-implicit scheme	
GNH	Procedures specific to non-hydrostatic gridpoint calculations	
CP or GP	Non-specific gridpoint calculation procedures	
SP	Spectral calculation procedures	
COMM, GATH, ISND, IRCV, OSND, ORCV, BR, DI or TR	Procedures dealing with inter-nodes communications ("COMMunicate", "GATHer", "Input SeND", "Input ReCeiVe", "Output SeND", "Output ReCeiVe", "BRoadcast", "DIstribute", "TRanspose")	
RE or RD	Procedures to read data	
WR	Procedures to write data	
E	Procedures specific to ALADIN ("Elliptic")	
	Tangent linear of a procedure	TL
	Adjoint of a procedure	AD
	Inverse of a procedure	IN

2.6.2 Error handling

Proper management of the errors during the execution of the program help finding them more quickly.

CTRL(17) On error detection, a brief message describing the error should be written out to the conventional error file and output file with logical unit numbers are respectively `NULERR` and `NULOUT`.
On one hand it is important to write out the error message on `NULERR` otherwise if only processors other than 1 abort we have no information about the abort, unless we ask for all the output files (one per task). But in this case the number of files can be so large that the debugging would not be easier.
On the other hand, writing twice the error message (on `NULERR` and `NULOUT`) can confuse the user, and there can even be a huge number of identical error messages in the listing if all processors abort for the same reason.

CTRL(18) Then, if abnormal termination is required, the subroutine `ABOR1` should be called with a message (a character string) as argument, indicating the error location. The use of the subroutine `ABOR1` gives time to flush the output buffer and to release the processors not causing `ABOR1`. It writes out a message on `NULERR`, and possibly on `NULOUT` if an argument is provided.

CTRL(19) Sometimes it can be advantageous to postpone the abnormal termination until the end of the subroutine in order to output all the errors detected to the output file before actually aborting.

NORM(30) The statement `STOP` should not be used in case of an error because it reports a normal termination code.

Refer to Figure 2.1 for examples of error handling.

2.6.3 "Hook" function

CTRL(20) Each subroutine should start and end with a conditional call to a "hook" subroutine. One main usage for it may be finding really awful bugs where we do not get any traceback because the stack has been trashed, but there are also many other potential uses, for statistics gathering, doing 'checksumming' for early catching of problems, etc.

Figure 2.1 shows an example of "hook" function.

2.6.4 Handling universal constants

CTRL(21) Universal constants are stored in a data module named `YOMCST`. To access them one should use this module.

CTRL(22) Universal constants should not be redefined at any other place in the code, to avoid any potential inconsistency after a redefinition.

CTRL(23) Universal constants should not be accessed via dummy arguments because there would be a risk to overwrite them through the subroutine interface.

CTRL(24) To make it more robust all universal constants should be declared and initialized in a unique module (fusion of the module `YOMCST` and the subroutine `SUCST` of today).

2.6.5 Purpose and usage of the key LECMWF

In order to simplify user namelist files a different default setup is performed according to the value of the logical key LECMWF. If LECMWF is .TRUE. then the selected default setup corresponds to the framework of ECMWF; else it corresponds to the framework of METEO-FRANCE.

CCPT(11) The key LECMWF should appear only in the setup routines and should be used only to initialize namelists variables in order to preserve the scientific flexibility of the code.

2.6.6 Purpose and usage of the key LELAM

The logical key LELAM enables the selection of the limited area model (ALADIN) instead of the global model (ARPEGE/IFS). Thus this key controls branches of the code related to the limited-area versus global aspects of the model.

CCPT(12) The key LELAM should be used only in the setup and control subroutines (ie: not below SCAN2MDM) in order to minimise the scientific generality of the code.

CCPT(13) The code below the key LELAM should be modular as far as possible in order to preserve the visibility of the ALADIN specific code from those who are not ALADIN partners.

CCPT(14) Use of LELAM should be as rare as possible. If a routine uses lots of LELAM keys then it should have its own ALADIN counterpart subroutine called under a single LELAM key.

2.6.7 Purpose and usage of the key LRPLANE

The logical key LRPLANE selects the plane geometry instead of the spherical one. Therefore this key has a strong relationship with the key LELAM.

CCPT(15) Contrary to the key LELAM, the key LRPLANE can be used at any place in the code, but to preserve the scientific generality of the code it should not replace the key LELAM. It can be used outside a LELAM section to treat in a general way low-level parts of the code (for example: in the semi-lagrangian scheme).

Note that LELAM=.TRUE. together with LRPLANE=.FALSE. would indicate that ALADIN is run in spherical latitudes-longitudes geometry instead of the usual projected plane. This facility has been abandoned in practice for quite a few years but should remain possible in principle.

2.6.8 Model settings

CTRL(25) User variables for setting up the model should be accessed via a conventional formatted sequential file containing namelists. Its logical unit number is: NULNAM (NULNAM=4).

CCPT(16) Namelist variables should be read from the namelist file and initialized only at one place in the software, in order to prevent redefinition of variables.

CCPT(17) To enable an easy control of the variables used in the program, all the namelist variables should be printed out to the listing file and not be redefined later in the code.

2.6.9 Output messages

CTRL(26) Messages should be written to the conventional formatted sequential file with logical unit number: NULOUT. The standard output ("*" or unit 6) should not be used as it would mix the messages coming from different processors.

CCPT(18) Important messages may be written out to the standard error file which logical number in the software is `NULERR`. In that case messages coming from the different processors will be mixed.

CCPT(19) Verbosity should be controlled by the specific namelist variable `NPRINTLEV`, running between 0 (minimum prints and default value) to 2 (maximum prints).

2.6.10 I/O raw data

CCPT(20) Observation files are binary files to be handled with the ODB software.

CCPT(21) Restart files are binary files to be handled with the PBIO software, which uses C I/O and gives pure binary files without any Fortran record structure.

CCPT(22) Movies (Meteo France only) are Fortran sequential binary files.

CCPT(23) In the ECMWF framework other user's I/O raw data should be accessed via GRIB files, using the PBIO software.

CCPT(24) In the ARPEGE/ALADIN framework other user's I/O raw data should be accessed either via FA files if the horizontal format of the data corresponds to the model settings; else via LFI files. These are unformatted indexed sequential files.

CCPT(25) It is recommended to use the logical key `LARPEGEF` rather than the key `LECMWF` to select the files format FA/LFI versus GRIB.

More generally, the recommendation is to use C I/O to improve the portability (today almost all computers adhere to the IEEE standard).

2.6.11 Message passing interface

CCPT(26) One should use the MPL package as interface for any message passing.

CTRL(27) For an easier control of the code, each MPL subroutine call should have its argument `CDSTRING` explicitely documented as the name of caller routine. Figure 2.7 shows an example of this.

Chapter 3

Source code management

The source code is stored in a database managed by the ClearCase[1] software package.
Among other advantages, the use of this software makes it possible to maintain an accurate view of the history of the code, and to simplify and make code merging operations safer. Thus, **it is of vital importance to use ClearCase to modify the code.**

A few standards should be considered while handling the source code files:

CTRL(28) The whole source code is partioned into *projects*. Below each project the source code is partioned into directories. Each directory contains elementary files which are either compilable source files or pieces of source files ("include files") to be included in other source files.

CTRL(29) Each elementary file should contain only one module or only one procedure: this makes the maintenance easier (but a procedure may include more than one subroutine via the instruction `CONTAINS`).

CTRL(30) All procedures which are internal to a package should be encapsulated inside a module: through the recompilation of the dependencies this enables the compiler to check automatically the interfaces for all the depending procedures. This has already been done for the spectral transform package.

CCPT(27) Each elementary file should be put in the proper project and below the directory which best fits its topic. For example: dynamics routines should be put in the ARPEGE/IFS project directory `adiab`.

CCPT(28) `NAMELIST` statements should be declared in a module containing the namelist variables (data part) as well as the subroutine initializing these variables (via the `CONTAINS` statement): this would make the maintenance and developments easier.

CTRL(31) The basename of each compilable source file should be the name (in small letters) of the module or procedure it contains. For example: the file `suct0.F90` contains the subroutine `SUCT0`.

CCPT(29) Derived types should be declared in a module because this manner is more robust than using the attribute `SEQUENCE` and it makes the maintenance easier (no duplication of code). There should be one module dedicated to the declaration of each derived type (or group of derived types if they are closely related), and vice-versa. These modules could also contain "primitive" operations on the type(s) like allocation or deallocation of

[1] http://www.rational.com/products/clearcase/index.jsp

its components, etc. The *structures* defined by this or these type(s) should not be in this module, only type(s) definitions and basic operations on the type(s) should be.

CTRL(32) Each namelist should be contained in a specific include file, which basename should be the name of the namelist (in small letters). For example: the file `namct0.h` contains the namelist `NAMCT0`.

CTRL(33) Each explicit interface should be contained in a specific include file, with basename the name of the subroutine it contains. For example: the file `suspec.h` contains the interface block of the subroutine `SUSPEC`. Note: an explicit interface is necessary whenever a pointer variable is used as a dummy argument. Interfaces should be computer-generated.

CTRL(34) Useless files should be deleted.

Index of standards for the presentation of the code

PRES(01)	Executable lines should be written using upper case characters.	12
PRES(02)	Comments should be written with lower case characters ...	12
PRES(03)	Indentation rules.	13
PRES(04)	The ending statement of a module or subroutine should repeat its name.	13
PRES(05)	One should avoid writing more than one statement per line.	13
PRES(06)	The comments should be written in English only.	13
PRES(07)	Blank lines should remain empty	13
PRES(08)	Namelist and internal variables in data module to be separated.	15
PRES(09)	Each description line should be independent.	15
PRES(10)	The documentation should be separated from the starting statement.	15
PRES(11)	Each procedure should begin with a documentation header	15
PRES(12)	Header documentation to be separated from the entry point statement.	16
PRES(13)	Template for *modifications* comments	16
PRES(14)	Declaration of variables to be separated from the header documentation.	16
PRES(15)	Declarations of variables to be grouped according to type & attributes.	16
PRES(16)	All attributes of a variable to be grouped in the same instruction.	16
PRES(17)	Templates like "`! Dummy scalar arguments` :" etc. are not necessary.	18
PRES(18)	Each variable should be declared separately.	18
PRES(19)	Items to be regularly spaced in used variables lists.	19
PRES(20)	The declaration and the presentation of dummy arguments to be the same.	19
PRES(21)	Separating commas at the end of lines in the `SUBROUTINE` variable lists.	19
PRES(22)	New lines in the `CALL` variable lists as new lines in the subroutine.	19
PRES(23)	Code body to be split into numbered sections and subsections.	21
PRES(24)	Each *section* should be clearly separated from the previous one	21
PRES(25)	Each *subsection* should be clearly separated from the previous one	21
PRES(26)	Comments to be placed just before or on the same line as the code.	21
PRES(27)	`DO` and `DO WHILE` loops should be indented with 2 blank spaces.	22
PRES(28)	Use a character label for each loop in case of complex loops nesting.	22
PRES(29)	Conditional blocks should be indented with 2 blank spaces.	22
PRES(30)	Nesting of conditional blocks should not be more than 3 levels deep.	22
PRES(31)	Conditional block boundaries should stand out.	22
PRES(32)	The length should be limited to 80 characters per line.	22
PRES(33)	Continuation lines to be indented with one supplementary blank space.	24
PRES(34)	Lines should be broken in a readable manner.	24
PRES(35)	The continuation characters should be aligned on the same columns.	24

Index of standards for the respect of the norm

NORM(01)	Usage of Fortran 90 free format and C	12
NORM(02)	No use of tabulations.	13
NORM(03)	Mandatory use of `IMPLICIT NONE`.	16
NORM(04)	No hard-coded variables.	16
NORM(05)	No use of the *statement* `DIMENSION`	16
NORM(06)	Mandatory use of the notation "`::`"	16
NORM(07)	Variables or constants are preferably declared with explicit kind	18
NORM(08)	Variables to be used or declared in a conventional order	18
NORM(09)	Use `ONLY`.	19
NORM(10)	Prefix convention for variables	19
NORM(11)	No `DOUBLE PRECISION` variables.	20
NORM(12)	Prefix convention for elementary variables of a derived type.	20
NORM(13)	No use of `GO TO`.	21
NORM(14)	No use of `FORMAT`.	21
NORM(15)	No use of `COMMON`.	21
NORM(16)	No use of `EQUIVALENCE`.	21
NORM(17)	No use of `COMPLEX`.	21
NORM(18)	Character strings to be declared with the syntax `CHARACTER(LEN=`n`)`	21
NORM(29)	Arrays should not be declared with implicit *size*.	22
NORM(20)	Mandatory use of `DO` ... `ENDDO` block loop.	22
NORM(21)	Continuation character `&`	22
NORM(22)	Automatic arrays preferred to explicitly allocated arrays.	24
NORM(23)	Local arrays to be deallocated at the end of the subroutine	24
NORM(24)	Mandatory use of the Fortran 90 specific comparison operators	24
NORM(25)	No use of the operators `==` and `/=` to compare real variables	24
NORM(26)	*Generic* names to be used for intrinsic procedures	25
NORM(27)	Mandatory use of `INTENT` attribute	26
NORM(28)	Passing array subsections to a subroutine	27
NORM(29)	Use array sections when calling intrinsic routines	27
NORM(30)	No use of `STOP` in case of error.	30

Index of standards for the control of the code

CTRL(01)	Only one entry point and at most two kinds of exit points	13
CTRL(02)	The entry point should be at the top of the procedure.	13
CTRL(03)	Usage of `RETURN` statement is discouraged.	14
CTRL(04)	Abnormal termination to be invoked `ABOR1`.	14
CTRL(05)	All variables in data modules to be saved `SAVE` statement.	18
CTRL(06)	Shape of arrays should not be changed when passed to a subroutine.	21
CTRL(07)	Type of variables should not be changed when passed to a subroutine.	21
CTRL(08)	Usage of `SELECT CASE`.	22
CTRL(09)	Position of explicit interface blocks.	26
CTRL(10)	The number of dummy arguments should not exceed 9.	26
CTRL(11)	Actual/dummy arguments to be presented following a conventional order.	26
CTRL(12)	Conventional prefixes or suffixes are recommended for names.	27
CTRL(13)	Radical of a type definition module name.	27
CTRL(14)	Radical of a procedure module name.	27
CTRL(15)	Prefix of ALADIN subroutines which are counterparts of ARPEGE/IFS ones.	27
CTRL(16)	Namelists handling.	27
CTRL(17)	Error detection handling: messages and output units.	30
CTRL(18)	Error detection handling: usage of `ABOR1`.	30
CTRL(19)	Postponing of abnormal termination.	30
CTRL(20)	"Hook" function	30
CTRL(21)	Universal constants to be stored in data module `YOMCST`.	30
CTRL(22)	Universal constants not be redefined at any other place than `YOMCST`.	30
CTRL(23)	Universal constants not to be accessed via dummy arguments.	30
CTRL(24)	Universal constants to be saved and initialized in a unique module `YOMCST`.	30
CTRL(25)	User access to variables va namelists.	31
CTRL(26)	Conventional output unit for messages.	31
CTRL(27)	MPL subroutines to have their argument `CDSTRING` explicitely documented.	32
CTRL(28)	Partionment of the source code.	34
CTRL(29)	Each elementary file should contain only one module or only one procedure.	34
CTRL(30)	All internal procedures to be encapsulated inside a module.	34
CTRL(31)	File basename to be the name of the module/procedure it contains.	34
CTRL(32)	Each namelist to be contained in a specific include file.	35
CTRL(33)	Position of explicit interface blocks.	35
CTRL(34)	Useless files should be deleted.	35

Index of standards for the conception of the code

CCPT(01)	Subroutines should not have more than 300 executable statements.	13
CCPT(02)	It is easier to add or remove lines than to modify existing ones.	13
CCPT(03)	Each data module should begin with a documentation header.	14
CCPT(04)	Actually unused variables (local in a used module) should be removed.	18
CCPT(05)	Variables suffixed with `L` are local in the sense of the distribution.	20
CCPT(06)	Variables suffixed with `G` are global in the sense of the distribution.	20
CCPT(07)	Loops should be as plain as possible.	22
CCPT(08)	Usage of dynamic memory.	24
CCPT(09)	Do not repeat sequences like: `ALLOCATE/DEALLOCATE/ALLOCATE`	24
CCPT(10)	The use of array syntax is not recommended.	25
CCPT(11)	Usage of the key `LECMWF`.	31
CCPT(12)	Usage of the key `LELAM`.	31
CCPT(13)	The code below the key `LELAM` should be modular as far as possible.	31
CCPT(14)	Use of `LELAM` should be as rare as possible.	31
CCPT(15)	Usage of the key `LRPLANE`.	31
CCPT(16)	Namelist variables to be read and initialized only once.	31
CCPT(17)	Namelist variables to be printed out to the listing.	31
CCPT(18)	Important messages may be written out to the standard error file.	32
CCPT(19)	Verbosity to be controlled by a specific namelist variable.	32
CCPT(20)	Observation files to be handled with ODB.	32
CCPT(21)	Restart files be handled with PBIO.	32
CCPT(22)	Movies files are Fortran sequential binary files.	32
CCPT(23)	For IFS other user's I/O raw data are GRIB files	32
CCPT(24)	For ARPEGE/ALADIN other user"s I/O raw data are FA or LFI files.	32
CCPT(25)	Usage of the key `LARPEGEF`.	32
CCPT(26)	MPL package to be used as interface for any message passing.	32
CCPT(27)	Files to be put in the proper project and below the proper directory.	34
CCPT(28)	`NAMELIST` statement to be declared in a data/procedure module.	34
CCPT(29)	Derived types to be declared in a module.	34

Index

A

ABOR1, 14, 30
ADJUSTL, 25
ADJUSTR, 25
ALL, 25
ALLOCATE, 24, 34
ANY, 25

C

CHARACTER, 19, 21
COMMON, 21
COMPLEX, 21
CONTAINS, 34
COUNT, 25
CYCLE, 21, 22

D

DEALLOCATE, 24, 34
DIMENSION, 16
DO, 21, 22
DOT_PRODUCT, 25

E

EPSILON, 25
EQUIVALENCE, 21
EXIT, 21, 22

F

FORMAT, 21

G

GO TO, 21

H

HUGE, 25

I

IMPLICIT NONE, 16
INTEGER, 19, 27
INTENT, 26

K

KIND, 18

L

LARPEGEF, 32
LECMWF, 31, 32
LELAM, 31
LOGICAL, 19

M

MAXLOC, 25
MAXVAL, 25
MINLOC, 25
MINVAL, 25
MODULE, 14, 18, 19, 21, 29, 34

N

NAMELIST, 27, 34, 35
NULERR, 30, 32
NULNAM, 31
NULOUT, 30, 31

O

ONLY, 19

P

PARAMETER, 18, 19, 24, 29
POINTER, 21, 24, 35

R

REAL, 19
RESHAPE, 21, 25
RETURN, 14

S

SAVE, 18
SELECT CASE, 21, 22
SELECTED_INT_KIND, 18
SELECTED_REAL_KIND, 18
SHAPE, 25
SIZE, 25
SPACING, 24
SUM, 25
SYSTEM_CLOCK, 25

T

TINY, 25

TRANSFER, 21, 25
TRIM, 25
TYPE, 9, 19–21, 26, 27, 29, 34

Bibliography

[1] Gibson J.K.: *Standards for software development and maintenance*, ECMWF Operations department technical memorandom nr 120, 1986

[2] Clochard J.: *Norme de codage "DOCTOR" pour le projet ARPEGE*, Note de travail "ARPEGE" nr 4, 1988

[3] Kalnay et al: *Rules for Interchange of Physical Parametrizations*, Bull. A.M.S., 70 No. 6, 1989

[4] Adams, J.C. et al. : *Fortran 90 Handbook, McGraw-Hill*, 1992.

[5] Andrews P. (UKMO), G. Cats (KNMI/HIRLAM), D. Dent (ECMWF), M. Gertz (DWD), J.-L. Ricard (METEO-FRANCE): *European standards for writing and documenting exchangeable Fortran 90 code*, version 1.1, 1995.
http://www.met-office.gov.uk/research/nwp/numerical/fortran90/f90_standards.html

[6] Hill L.: *Règles essentielles pour l'utilisation du langage Fortran 90*, CNES, 1995.

[7] Ajjaji R. (Maroc-Météo), J. Boutahar (Maroc-Météo) and J.-F. Geleyn (Météo-France): *Aladin phaser's guide*, 1998.
http://www.cnrm.meteo.fr/aladin/concept/phasersguide.html

[8] Zagar M. (Hydrometeorological Institude of Slovenia) and C. Fischer (Météo-France): *The ARPEGE/ALADIN Tech'Book: Implications of LAM aspects on the global model code, CY25T1/AL25T1*, 2002.
http://intra.cnrm.meteo.fr/gmod/modeles/Tech/Aladin_implement/al2ec/al2ec.html