Publication Year	1999
Acceptance in OA	2023-03-09T11:00:57Z
Title	FORTRAN 90 Programming Guidelines for PLANCK/LFI
Authors	MARIS, Michele
Handle	http://hdl.handle.net/20.500.12386/34013
Volume	LFI-OAT- 0002.01

FORTRAN 90

Programming Guidelines

for

PLANCK/LFI

Michele Maris

Published at Osservatorio Astronomico di Trieste (OAT), Trieste, Italy

Abstract

A set of rules for FORTRAN 90 programming are given together with many rules for FORTRAN 77 programmers helpful to write code FORTRAN 90 compatible.

Document History

Issue 0.1

Checked by: C. Vuerli, R. Smareglia **Date:** 8 March 1999

Approved by: F. Pasian **Date:** 8 March 1999

Changes: First Issue

Introduction

In this document a set of rules to produce uniform and homogeneous FORTRAN 90 software for Planck/LFI is described, part of them comes from the experience gained during the porting of the simulation software for Planck/LFI from UNIX/VAX to Windows/NT (PIR 99). These rules are intended as an help to maximize software readability, maintainability, portability and formal correctness. This document is related to programming rules only, standard libraries, software development tools, etc. will be defined on subsequent documents. All over the document FORTRAN 77, 90 and 95 will be abbreviated with F77, F90 and F95 respectively.

As other documents about the software organization for the LFI project (Barfoed 1998; O'Mullane, Hazell, Barfoed 1998) this document deals with the *development phase* of the simulation software and not with the *production phase*. The first one being characterized by intensive code modifications followed by frequent tests during which a limited amount of data, often taken from a set of representative case studies, is processed. The second one is characterized by the use of a quite stable code to process a large amount of data.

The rules described in this document are not only for FORTRAN 90 programmers but also for FORTRAN 77 users whose task is to produce a code which is as more as possible FORTRAN 90 compatible. In addition, since it is strongly recommended for any FORTRAN 77 programmer to migrate to FORTRAN 90. The prompt application of these guidelines to FORTRAN 77 programs will simplify the subsequent porting and/or integration of FORTRAN 77 code into FORTRAN 90.

To simplify the application of these guidelines, software tools may be created or applied. As an example: porting of old F77 codes into F90 may be simplified writing a pre-processor able to translate comments and lines continuations in the old F77 code accordingly to the suggestions expressed in these guidelines.

Style

Programs must be written in an homogeneous style, accordingly to a set of rigorous rules which specifies how to compose the various parts of the program.

1. Indentation have to be used whenever possible.
2. Non standard language features have to be discarded.

Naming

Programs must be readable and comprehensible by each user which knows the underline theory.

1. Names must be UNIX like.
2. Variables names, routine names, program names must recall the scope of their content or the function.
3. Short mnemonics must be explicitly explained commenting the program.

The degree of self explanation for a mnemonic (and so their length) must be tuned taking in account the frequency of use of the possibility that it may be used far from its definition.

4. Mnemonics used *far* from their definitions must be more self explanatory (more long) than mnemonics used *near* their definition.
5. Mnemonics *rarelly* used must be more self explanatory (more long) than mnemonics used *frequently* .
6. A hierarchy for mnemonics length is:

Mnemonics in commons or global variables

Longer than

Mnemonics used in long blocks

Longer than

Local variables in subroutines

7. Mnemonics and names shall be composed in a consistent way, through a rigorous grammar and a syntax, taking in account the previous rules about mnemonics length.

As an example, a list of values of the quantity ρ may be stored in a variable named: `RhoLst` which is the composition of two words from the established dictionary: "`Rho`"."`Lst`".

8. In a loop, the index related to the main list of values to be scanned or variable to be sampled may be labeled `I_`*`variable-name`*`.`

As an example: the index to scan the list of ρ values `RhoLst` is `I_Rho` (read as: *I of Rho*), so that the elements of `RhoLst` are addressed by `RhoLst(I_Rho)`.

i) Standard scopes may be declared using specific names or prefix or suffixes.
ii) String scratch variables may be named `VCS1, VCS2, ...` where the prefix `VC` recall that those variables are for scratch, `S` that they are strings and numbers distinguish between different variables.
iii) Pointers are labeled using the `P_` prefix.

As an example: the pointer to the object `ScanCircle` is named `P_ScanCircle`.

9. Composed names must be readable.
10. Names have to be written in lower case, the first letter of each word in the name be written in Upper case.

As an example, write: `SpinRatio`, instead of `SPINRATIO` or `spinratio`. Alternatively use underscores "_" to split long words: `SpinRatio_One`.

11. Rule 9) is relaxed when another name composition allows a better correspondence between the program object and the physical object it represents.

As an example: $t_{Detection}$ is better represented by `tDetection` or `t_Detection` rather than by `TDetection`.

12. Even in F77, variables and routine names longer than 6 characters have to be used whenever required.

It has to be noted that rule 11) is an explicit violation of F77 standard, but it is acceptable since it improves the code readability, it is F90 compatible and all the tested F77 compilers has compilation options to handle names longer than 6 characters. Solutions for F77 compilers unable to properly handle names longer than 6 characters will be proposed time by time if required.

The creation of a standard dictionary of terms is beyond the scopes of this document and has to be addressed in a separated work.

Modularity and Functional Blocks Classes

To describe modularity a simple model of a program is given in order to fix terminology. In this model a high level description of the program from the user point of view is assumed. In this framework the program is described by the sequence of tasks the algorithm has to accomplish to reach its target, each program is decomposed into a sequence of *functional blocks* or simply *blocks*. Blocks are defined as follow:

1. A program is a collection of functional blocks organized by the algorithmic flow.
2. A functional block is a collection of FORTRAN statements with a well defined task in the high level description of the algorithm realized by the code.

3. A FORTRAN statement is a string of characters which may be interpreted by the compiler as a single FORTRAN operation.

While the rules for the functional blocks composition are:

4. Each program block must accomplish one operation only and must belong to one of the block classes described below.
5. The use of flags to change the nature of the operations executed by each block or part of it have to be avoided.

As an example, rule 5) prevents the implementation of Input blocks which may be used for Output too and discourages the use of flags to choose between different algorithmic paths during the execution (as an example different integration algorithms and so on). It should be better that different algorithmic paths should be accomplished by different executable programs, in order to enhance readability, to reduce the number of parameters required in input and to prevent subtle errors due to miss interpreted flags.

6. Blocks have to be short to improve readability, a good block should be as short as possible.
7. If a long block is required it should be better to split its content in a set of simple routines.
8. Each subroutine, function or include file must performs the task or part of the task of a single functional block.

About rules 6) and 7) a good practice to assure readability and a proper program decomposition would be to fix a maximum block length, a good practical limit for the length of a block would be one or two pages (about 55 – 100 lines). Blocks of executable statements significantly longer than that limit are hard to be read and maintained and it would be better to split them accordingly to rule 4). Of course this

suggestion has to be regarded as an orientation only, since longer blocks are admitted whenever they allow a better programming.

Functional blocks may be classified in different ways, the classification adopted here is coherent with the model described in the introduction of this paragraph.

9. Functional blocks are classified into four classes accordingly to their purpose.
10. Blocks must belong to one and no more than one of such classes, i.e. one block may have one and only one purpose.
11. As a consequence multipurpose functions, subroutines or include files are not allowed.

Example: a block or a subroutine can not at the same time receive input from the keyboard and execute a computation.

12. Up to five purposes corresponding to four classes are described here, if required, new purposes and classes will be introduced.

i) *Input / Output* blocks are used to receive in input data and to put them in output with a specific format.

ii) *Error Trapping* blocks are used to manage exceptions and error situations.

iii) *Initialization* blocks are used to initialize data structures (as an example commons). Of course in a given program only one initialization block associated to a given structure is allowed i.e. one data structure can not be initialized by two initialization blocks.

iv) *Computation* blocks are used to execute the computation.

v) *Main block,* its task is to encapsulate and connect all the other blocks to form one executable program, a library of standardized *Main* blocks to be used as a template may be suggested to produce different programs may be suggested.

Of course, any single executable program may have one and only one main block, the class v) is added just to suggest the creation of a specific repository for the *Main* blocks which forms the various executable programs.

Errors and Warnings

During the execution programs must manage in a different way *errors* and *warnings.*

Error:

One event is classified as an *error* when it prevents the program to continue its execution.

1. When one *error occurs* the program has to be *stopped* and one *error message* has to be produced.

1.1 A clean *stop* must be guaranteed, files must be properly closed and resources deallocated before the process is completely closed.

1.2 The error message must be one line long, usually of no more than 80 characters.

1.3 It must have the form:

`Error: routine-name error-code, explanation`

1.3.1 `routine-name` is the name of the routine or block where the error occurred.

1.3.2 `error-code` identifies the type of error, the error code musts allow to locate the place where the error occurs inside the routine or block.

1.3.3 `explanation` is a short explanation of the error.

1.3.4 Eventually the `explanation` should contain more detailed indications about the error condition, as the loop index at which the error happens, the particular value of the variable checked to trap the error and so on.

Warning:

One event is classified as a *warning* when it does not prevent the program to continue its execution at the time in which it occurs, but potentially may

cause the program to produce one true error condition, or one erroneous result, or one efficiency loss, or any other negative consequence which should be better (but not mandatory) to avoid during the subsequent execution.

2. When a *warning occur* it must be notified by a *warning messages* but the execution is *not stopped*.

2.1 The warning message must be one line long, usually of no more than 80 characters.

2.2 It must have the form:

`Warning: routine-name warning-code, explanation`

2.2.1 *`routine-name`* is the name of the routine or block where the warning occurred.

2.2.2 *`error-code`* identifies the type of warning, the warning code musts allow to locate the place where the warning occurs inside the routine or block.

2.2.3 *`explanation`* is a short explanation of the warning.

2.2.4 Eventually the *`explanation`* should contain more detailed indications about the warning condition, as the loop index at which the warning happens, the particular value of the variable checked to trap the warning and so on.

The standardization of messages for warnings and errors allows the automatic scan of the log – file looking for errors or warning conditions at the end of the simulation process.

Examples:

The failure in convergence of an integration program is an *error* condition.

If there is a control parameter *P* for which in the range of allowed values [*a*, *b*] the program may or may not converge in an unpredictable manner, a *warning* condition will occur when a value in the range [*a*, *b*] will be assigned to *P*.
Of course, if there is one value *z* of *P* for which the program surely does not converge, then to assign *P* = *z* will be one error.
A *warning* condition would occur also when, for some value of *P*, the program is slowed significantly or the amount of memory required is quite high or in other similar cases.

Structures

The program must be described in terms of standard constructs as those of *structured programming*. The recommendations in this section are explicitly written for F77 programmers since they are implicitly included in F90 if used properly.

1. All the structures for loops or switch structures which are not explicitly allowed by the *structured programming* model are forbidden.
2. The allowed constructs are: `While do`, `Repeat Until`, `Do For`.
3. It is forbidden to enter and exit from loops directly. Loops must be accessed only at its first line and left only at its end.
4. The simulations of such constructs in F77 has to follow the structures described in *Numerical Recipes* (Press, et al., 1986) and reported in a related appendix to this document.

8 Jumps

The use of `GOTO`s have to be avoided programming in structured languages such as: C, F90, or F95.

1. Whenever possible, programming F77 `GOTO`s should not to point more than 55 lines of text far from the starting line (the standard A4 page size).
2. Even in F77 the use of `GOTO`s to emulate structured programming functions have to be rigorous. Only one method for each structure emulation must be used (see the *Structures* rules for further details).

3. Computed jumps are forbidden. The equivalent of the C `switch` (PASCAL or F90: `CASE`) has to be simulated in F77 by a sequence of `IF … THEN …` `ELSEIF … THEN … ELSE …` constructs.

Declarations

Variables together with their types have to be explicitly declared.

1. The FORTRAN `IMPLICIT` assignment of data types is forbidden.
2. The compiler option for IMPLICIT assignment of undeclared variables must be excluded with the `IMPLICIT NONE` FORTRAN statement.
3. Rules (1) and (2) are mandatory for new programs and libraries and for upgraded old programs and libraries. It must be applied whenever possible even to old programs and libraries which are not planned to be upgraded.

Functions Typing

1. Both in F77 and F90 the type of functions must be explicitly declared.
2. The declaration must have the form

```
Type Function name(parameters-list)
```

As an example:

```
Real Function IGNU(X)
```

It has to be noted that both F77, F90 allows to define explicitly typized functions declaring their type in the body of the function itself.
Example:

```
Function IGNU(X)
Real IGNU
Real X
```

```
      ...
   Body of IGNU
      ...
IGNU = X+2.0*X
End
```

here `IGNU` should be implicitly regarded as `Integer`, but since the `Real IGNU` statement it is a `Real`. Despite all FORTRAN compilers recognize this syntax, some PC compilers do not implement it properly. When called `IGNU` will produce a `Real` number, but its output will be handled by the calling expression as an `Integer`, producing an erroneous result.

F77 and F90 Compliance

The conventions illustrated above work properly if the F77 and F90 compilers follow the FORTRAN standards.

1. F77 and F90 standard must be followed strictly.

As an example, many F77 compilers allows to write text of code beyond column 72 (ex.: the option of 132 columns code) so:

1.1 in order to take full advantage from these guidelines F77 programs must respect the limit of column 72, even if the compiler of choice allows to skip such restriction.

2. Exception to rule 1) is the use of variables and file names longer than 6 characters which is admitted since it improve the readability, while all the commonly used F77 compilers are able to manage properly such extension (see the section about *Functions Typing*)

Comments and Line Continuation

The Visual F90 compiler is very rigorous about F90 compliance. In particular, in porting CMB packages from UNIX to Windows/NT (PIR 99), all F90 programs of the original packages have had to be edited to correct comments and continuation characters, according to the F90 rules (Metcalf, Reid 1990). The modifications were:

1. Precede ‘C’ or ‘c’ or any comment character in the first column with ‘!’
2. Replace ‘&’ or any other continuation character in column 6th with a ‘space’ and:

   i) write all the statements on one line
   ii) if lines are too long to be readable, put the continuation character at the end of line

3. To assure that libraries written in FORTRAN using only F77 commands are compatible also with F90, two conventions about comments and line continuation are needed:

   i) comments MUST begin only with ! at first column
   ii) line continuations must have the following form:

```
   line.first              &
& line.second             &
& line.third              &
& …
& line.last
```

the & on the right side of the line must be located at the column 73 to assure it is not read by the F77 compiler.

4. Comments starting with ! at the right of the line must be pushed after column 73 or better, be inserted in a new line.

```
& line.nth                &  ! Comment Here
```

because the following form

```
& line.mth ! Comment here    &
```

will prevent a correct continuation in the F90 compiler

Duplicated Files in Libraries

1. Avoid duplicated routines and objects in library files.
2. If two already existing libraries shares the same objects a third library has to be created containing the shared objects which will be removed from the original locations libraries.

As an example: `library1.for`, `library2.for` can not both contain `my_routine` if both libraries have to be linked to the same program. In that case a third library, `librarycom.for`, has to be created containing just `my_routine`.

It should be proposed to merge the two libraries may be merged in one library, but the solution proposed by rule 2 is more coherent in terms of *Object Oriented* design of programs. In this view libraries and modules may be seen as an equivalent of *classes*. Each class collects routines and functions to handle specific objects (i.e. data structure, file type, physical model, etc.). Programs are described in terms of a hierarchy of classes . An efficient design avoids overlapping of classes. If two classes (let us call them *A*, *B*) overlap (i.e. $A \cap B \neq \varnothing$) the conflict may be solved merging the two classes or splitting them. *Merging* (i.e. replacing (*A*, *B*) with $C = A \cup B$) may be useful if the two classes are related to the same object, but when translated in terms of source files the resulting file may reveal too long to be read and maintained efficiently. *Splitting* (i.e. replacing (*A*, *B*) with (*A'*, *B'*,*C*) : $A' = A - A \cap B$, $B' = B - A \cap B$, $C = A \cap B$) is better since, not only the resulting files are smaller, but the code reuse is favored, because if two classes overlap it means that there are common functions and definitions which, with a good probability, will be required by other future classes.

Appendix: Emulation of Structured Program in Fortran 77

F77 does not have standardized instructions to allow structured programming, for this reason these instructions must be emulated by standardized structures. Here the structures suggested by Press et al. (1986, chapter 1) are proposed as the reference structures.

The following control structures are forbidden:

1) Assigned `GOTO` and `ASSIGN` statements
2) Computed `GOTO` statement
3) Arithmetic `IF` statement

So if you have to choose between different possibilities use the `Switch` construct (not described in Press et all. (1996)) instead of the computed `GOTO`.

The description is base on a simple metalanguage:

1. Structures are defined by a set of fixed statements and user dependent statements.
2. Names of structures are in bold face.
3. The user dependent parts to be replaced by the programmer are listed between `(…) ‘ in the structure name and their names are italicized.
4. The “`:`” character marks the begin of the FORTRAN section.
5. Italicized symbols which are not declared between `(…) ‘ in the structure name are for internal use only, the programmer may changes their name or value time by time provided the name or value of choice does not interfere with other parts of the program.
6. The symbol *`body`* and symbols such as *`body1`*, *`body2`* , … are used to indicate a list of user defined instructions.
7. After the “`:`” character statements related to the management of a given structure share the same indentation level, while the statements of *`body`* (*`body1`*, *`body2`*, …) are placed at the next indentation level.

Single targets for different `GOTO` must be avoided, example:

```
        … GOTO label1

        … GOTO label1

   label1 CONTINUE
```

must be replaced by:

```
        … GOTO label1

        … GOTO label2

   label1 CONTINUE
   label2 CONTINUE
```

To skip the execution of a block when a given condition occurs use the `IF … THEN … ENDIF` construct instead of `GOTO`:

```
                IF (skip-condition) GOTO skip-label
                   …
                  body
                   …
skip-label      CONTINUE
```

has to be replaced by:

```
             IF (skip-condition) THEN
                …
               body
                …
             ENDIF ! skip-condition
```

Note the repeated commented condition to identify to which `IF` the `ENDIF` belong.

At the same manner it is forbidden to exit from a block or a loop with a direct `GOTO`, use the `IF (`*skip-condition*`) THEN … ENDIF` construct instead.

```
Do(index, lower, upper, body)EndDo:
          DO label index = lower, upper
                …
               body
                …
   label  CONTINUE

Do–While(condition, body):
   label IF (condition) THEN
                …
               body
                …
          GOTO label
          ENDIF

Do(body)Until(condition):
   label  CONTINUE
                …
               body
                …
          IF (condition) GOTO label

C/C++ For(index = lower, increment-expression,
          execution-condition):
          index = lower
   label  IF (execution-condition) THEN
                …
               body
                …
          increment-expression
          GOTO label
          ENDIF

Switch(index, value1, value2, …, valueN, body1, body2, …, bodyN):
          IF (index.eq.value1) THEN
```

```
          …                    ! Comment 1
       body1
          …
ELSEIF (index.eq.value2) THEN
          …                    ! Comment 2
       body2
          …
ELSEIF…
          …
ELSEIF (index.eq.valueN) THEN
          …                    ! Comment N
       bodyN
          …
ELSE
          …                    ! Comment END
       bodyEND
          …
ENDIF
```

To quit a loop before its termination Press et al. (1986) suggest the `break–iteration` construct:

```
Press et al. Break-Iteration(break-condition, body1,
                             body2):
     label1 CONTINUE
               …
            body1
               …
          IF (break-condition) GOTO label2
               …
            body2
               …
          GOTO label1
     label2 CONTINUE
```

this is definitely a bad construct! When the task is to stop the execution of a do-loop before its natural termination, it is better to replace it with the equivalent of the C/C++ `For` construct or to use the `Repeat-Until` or the `Do-While` constructs. If there are parts of the loop which has to be skipped after the irregular termination use the `IF`

(*skip-condition*) `THEN` … `ENDIF` construct. Here are examples of such applications.

```
Break-Iteration(break-condition, index, limit1, limit2,
                body):
          index = limit1
    label IF ((index.le.limit2).and.not.(break-condition)) THEN
                …
               body
                …
          index = index + 1
          GOTO label
          ENDIF
```

Often the *break-condition* takes the form of (*break-flag.eq.true.*) where the *break-flag* is a `logical` variable which is set `.true.` inside the *body* of the loop when the loop has to be terminated. In this case use:

```
Break-Iteration(break-flag, index, limit1, limit2, body):
          break-flag = .false.
          index = limit1
    label IF ((index.le.limit2).and.not.(break-flag)) THEN
                …
               body
                …
          index = index + 1
          GOTO label
          ENDIF
```

A similar structure may be used when instead of a logical flag, an integer flag is preferred.

If there is a part of *body* which has not to be executed when the break condition occurs (like in the **`Break-Iteration`** of Press et al. (1996)) then use the `IF` … `THEN` … `ENDIF` construct.

```
Break-Iteration(break-condition, index, limit1, limit2,
                body1, body2):
          index = limit1
    label IF ((index.le.limit2).and.not.(break-condition)) THEN
                  …
                body1
                  …
                IF (not.break-condition) THEN
                       …
                    body2
                      …
                ELSEIF
          index = index + 1
          GOTO label
          ENDIF
```

Or analogously:

```
Break-Iteration(break-flag, index, limit1, limit2, body1,
                body2):
          break-flag = .false.
          index = limit1
    label IF ((index.le.limit2).and.not.(break-flag)) THEN
                  …
                body1
                  …
                IF (not.break-flag) THEN
                       …
                    body2
                      …
                ELSEIF
          index = index + 1
          GOTO label
          ENDIF
```

In a **Do-Loop** if a *step* different from +1 is required then use:

```
Do (index, limit1, step, limit2, body) EndDo:
           index = limit1
    label  IF (((step.ge.0).and.(index.le.limit2)).or.
          &    ((step.lt.0).and.(index.ge.limit2))     ) THEN
                  …
```

```
    body
      …
index = index + step
GOTO label
ENDIF
```

here it is assumed that `step > 0` if `limit1 < limit2` and `step < 0` if `limit1 > limit2`.

References

- Metcalf M., Reid J., 1990, *Fortran 90 Explained*, Oxford University Press, New York, USA, ISBN 0-19-853772-7
- Press W. H., Flannery B. P., Teukolsky S. A., Vetterling W. T., 1986, *Numerical Recipes*, Cambridge University Press, Cambridge, UK
- O'Mullane William, Hazell Adam, Barfoed Morten, 1998, *Software Repository User Requirements Document*, IDIS-URD-2
- Barfoed Morten, 1998, *CVS for IDIS (Initial recommendations)*, planck004.w70
- PIR 99 – Maris M., Pasian F., Smareglia R., Maino D., Burigana C., 1999, *Porting of CMB Packages from UNIX to Windows/NT Parallel Machines* (report in preparation)