

HIRLAM Coding Styles

Gerard Cats

April 2002

Introduction

A distributed project like HIRLAM will see a better productivity of its researchers if they follow certain coding styles for their Fortran programs. In principle, HIRLAM agreed to follow the DOCTOR standard, over 15 years ago. In practice, this standard is not always followed: sometimes because a younger programmer does not know of the existence of this agreement; sometimes because the code was imported; sometimes because the programmer is not convinced of the usefulness of a standard or of the suitability of DOCTOR. In the last of these cases, the programmer would probably choose a different standard.

This note gives a number of considerations when writing or importing a program.

In general, when writing a new program, we prefer to use Fortran 90. For this language, the SRNWP initiative has accepted a number of coding standards (<http://www.knmi.nl/hirlam/SRNWP>). The following considerations are not in conflict with those, but they are primarily intended for use with Fortran 77, hence for legacy software. In as far as the suggested styles require recoding, the programmer should consider Fortran 90 to replace Fortran 77.

In practice, it is very difficult to strictly impose adherence to coding styles: when a researcher has done an interesting piece of work, which would lead to a marked forecast improvement, on the one hand no researcher is interested to modify this work to match HIRLAM coding styles, because it does not lead to new publications; on the other hand, the HIRLAM project does not employ persons with the task to rewrite code by others just to match our styles. In the end, we usually choose to accept the code as it is, because of its marked forecast improvement. However, there is one minimum requirement, without which code cannot be accepted: **proper documentation**.

General style rules

When writing or importing a program, follow the following style rules. Sometimes they will be conflicting. The listing is in the order of priority of the different rules, so it indicates how conflicts should be resolved:

1. **document**: whatever coding style is followed, it is important to **document** it. Others, working with the same code, will more likely follow your coding style if they know what it is.
2. **uniformity**: A piece of Fortran is much easier to read and modify if it is using the same, **documented**, style throughout. So if you are going to modify a program, try to follow the style that was used when the program was written.
3. **follow mathematical notation**: it will make equations more readable
4. **follow English**: it will make the program logics more readable

5. **case of Fortran names:** use the case of Fortran names to enhance the readability of programs
6. **DOCTOR:** a few rules of DOCTOR have proven benefits

Some of these are addressed below, and so is a number of conflicts. In any case, it is important to always apply the rules judiciously, and, it cannot be repeated enough, to **document** the style. Having mentioned **documentation** so often now, I will not elaborate on this any more, although it is the most important rule of all.

The considerations listed above are recommendations. There are a few coding rules that are obligatory. Violation of them will make the program unacceptable. They are listed in the last section of this note.

Follow mathematical notation

The use of the notation ``t'`, ``u'`, ``v'`, ``q'` for the primary model variables makes code immediately much easier to read than e.g. `t` for temperature. Constants as ``g'`, ``f'`, ``pi'`, should be coded like that.

Use the Latex notation underscore (``_`) to denote subscripts. For example: ``p_LCL` for p_{LCL} , or ``z_i` (from z_i) for inversion height. The mathematical notation for turbulent kinetic energy is less standardised. You may encounter `'TKE'`, `E`, `e`, or a Greek character. Choose judiciously and **document** your choice.

In equations, the Monin-Obukhov length is usually written as ``L`. In Fortran, this would not be a good choice, because it is too easily interpreted as an integer (or logical), perhaps even a loop index. But ``zoverL` is clear for z/L .

Follow English

Some Fortran statements are almost English. Use them, and avoid to work around them. This is true in particular for `'if then else'`. I am not strongly opposing the use of `'goto'`, in some occasions it can make the logics of a program clearer than the alternatives. Compilers nowadays produce more efficient code from `'if then else'` constructs than from the alternatives, but only if `'if'` clauses are not deeply nested. In this respect they are like humans. If you yourself cannot follow the logics anymore through the nesting of if statements, it is time to consider `'goto'` (in particular where performance is not critical) or `'switches'` (a switch is a real variable, taking the value 0 in the false, and 1 in the true case; the switch is used multiplicatively to `'mask'` one of two terms in an expression, one of the terms corresponding to the `'true'` case, the other to the `'false'` one).

Spaces are insignificant in Fortran. However, spaces within variable names and/or Fortran keywords must be banned. (hence ``goto` in stead of ``go to`).

The choice of case can help to make a program more readable. Fortran is case-insensitive, so you cannot use the case to distinguish variables (as opposed to the equations, where T is temperature and τ is time). But you can use the case for clarification, as e.g. in ``dTdt`. Of course, you should use the case consistently.

Quantities at the top of the atmosphere are usually indicated by the subscript `TOA`. But `TOA` would be closer to English.

DOCTOR

The DOCTOR standard prescribes in fair detail the format of in-line documentation for a Fortran module. This aspect of the standard is more or less obsolete, because it originates from the times that it was not possible to use lower case text, and nobody had heard of more modern documentation techniques, like hypertext. Yet, the contents of the documentation, as prescribed by DOCTOR, should still be present in the in-line documentation (subroutine purpose and interface, author, record of changes, sections). Furthermore, DOCTOR prescribes the initial characters of Fortran variable names. The standard has a comprehensive scheme, but the main ones are:

- local variables: z, i, j (loop index)
- dummy arguments: k, l (logical), p

On the one hand, it is extremely useful to immediately recognise a variable to be local, or in a common block, *etc.* On the other hand, it is confusing that a variable changes its name in a program (e.g. in the namelist `NAMPRC`, there is a variable `IACDG`; this is a variable local to `HLPROG`. This variable is passed into the forecast model as actual argument to subroutines; in those subroutines the variable now is called `KACDG`). So this rule of the DOCTOR standard is disputed. Yet, you are requested to at least follow the two bulleted rules.

Conflicts

Here I just mention a few possible conflicts between the coding style rules. It is up to you to choose a proper strategy, and **document** it. It is recommended to write the code such that it becomes suitable for automatic code extraction, but the exact way to do this should be subject of a more thorough review of existing methods, in the overhaul context.

Case conflicts: t (time), T (temperature)

Math/English/DOCTOR conflicts: T(i,j,k,t) (math), T(ix, iy, il, it) (DOCTOR), T(i, j, k, it) (math, practical)

`z_i`, (`zz_i` ?), L for inversion height, Monin-Obukhov length, *resp.*

Requirements

`STOP` statements should only be used in the main program. Because HIRLAM does not have main programs, `STOP` is banned. If you want to stop the execution of a program, either `call abort` (obligatorily preceded by an error message to `stderr`) or return to the invoking routine.

Limit line lengths to 72 characters. Longer lines may lead to different results on different machines.

Do not use tabs (except for tabulation, implying in comments or strings). Because tabs are not always expanded to 8 characters, tabs are banned, if alone to allow to check line lengths.

Avoid trailing white space. It is confusing, and inhibits proper checking for line lengths.

Always use `implicit none`. This may require some significant coding, but the benefits of this construct compensate easily for the effort.

Indent if blocks and loop bodies (by 2 or 3 spaces).

Use generic names (`log`, not `alog` or `dlog`; `min`, not `amin1` *etc.*, beware that `cos` is not generic for `acos` ;-). This facilitates porting to machines that support several word lengths.

Subroutines must have one entry point (so `'entry'` is banned). They should have one exit point but `'return'` may appear after a fatal error.

Recommendations

Do not deviate from the ANSI standard.

After an error, return to the invoking subroutine with an error indicator set. Write an appropriate error message to `stderr` (better still, to both `stderr` and `stdout`, in this order). Interesting user support facilities are offered by the possibility to write HTML in the messages. (HIRLAM displays text in a browser as pre-formatted text. If your message is better displayed without pre-formatting, surround it by `</pre` and `<pre` tags, in this order). With this, you can *e.g.* add a link to a web document with more details on the error.

Invoking routines should always check the error indicator after each subroutine invocation.

Use `c` or `C` as initial comment character, but gradually change to the preferred `!`. However, leave empty lines empty. Never use `*`.

When writing code, use tools to automatically check uninitialized variables, argument lists, *etc.*

Use spaces to improve readability. I suggest, for example:

```
array( i+1, j+nlon ) = funtion( a * (i+3), j+nlon )
```

Use `'goto'`, but very cautiously.

Avoid common blocks but use them for constants (`g`, `pi`). Do not use constants like that except directly from the common block. So no program unit should specify `g`, except one single initialization module.

Namelists

Researchers are tempted to use namelists, to be able to experiment with tuning constants *etc.*, without having to recompile codes. This may be nice in the reasearch stage, but for operational implementations it is often confusing. Limit the use of namelists to those parameters that *are* subject to change. Do not use namelists for tuning parameters (except during the development phase, if you wish so). The default values of namelist parameters must be the reference configuration; so reference namelists should be empty.

The recommendation that the reference namelists should be empty implies that whenever there is a change of the reference values, this change is implemented by a change of the default value in the Fortran sources, not by a change in the scripts that provide the namelists.

To allow the Fortran program to distinguish between default settings and overrides by the namelist input, a namelist variable should have a separate value to identify the fact that the user wishes to use the default value. Often, there is a natural value for this default marker. For example, for the time step the value 0 could be used. So if the value after reading the namelist is 0, the Fortran program knows to substitute a reasonable value (depending on *e.g.* the resolution, and the advection and time stepping schemes). A consequence of this strategy is that binary valued namelist parameters ('logicals') must have a third value, to indicate the default; this implies they must be integers. For example, the parameter `nphys`, a *logical* to

indicate whether the physics package must be invoked, should be set dependent on the value of the integer `nphys`, which is a namelist parameter. It takes the following values:

- 0: do not invoke physics (*i.e.* set `nphys` to `.false.` after the namelist has been read)
- 1: invoke physics (*i.e.* set `nphys` to `.true.` after the namelist has been read)
- -1: use the default (*i.e.* 0 during the backward DFI phase, and 1 during the forward DFI and forecast).

With this strategy for namelist parameters, all consistency checks can be done at the Fortran level. Novice users are unlikely to construct inconsistent experiments (because they would use defaults, guaranteed to lead to consistent values), but more experienced users can always override the defaults with their own preferences.

Acknowledgments

I thank José-Antonio Garcia-Moya for his contributions.