Coding standards for the CP-PAW code

Peter E. Blöchl

Institute of Theoretical Physics; Clausthal University of Technology;
D-38678 Clausthal Zellerfeld; Germany;
http://www.pt.tu-clausthal.de/atp/

Contents

Chapter 1

Coding Standards of the PAW project

Coding and maintainance of large code projects are made more efficient, if one is guided by recurring patterns. Therefore, a style guide for coding projects is adviseable.

The style guide is, as the name says, a guide and not a law. Thus it is permitted to deviate from it if there is good reason.

While such rules appear to the beginnner as administrative overhead, they pay off in larger code projects.

Some of the principles have been inspired by the GNU Coding Standards
http://www.gnu.org/prep/standards.

1.1 SHELL scripting

[**?**]

- specify the `SHELL` variable: start the script with `#!/bin/bash` as first line. You may use another shell instead of bash, but bash is preferred.
- Define a variable named `USAGE` describing the function of the script and its options. Example:

  ```
  export $USAGE="Usage of $0\n"
  USAGE="$USAGE description \n"
  ```

  (`\$0` expands into the name of the script.). Insert the description in place of the string description. Several lines can be added similar to the second one given here.
- pass arguments as variables to options. You may allow for one special argument that is provided behind the options. Analyze options with getopts. Example

  ```
  while getopts :h0b:p: OPT ; do
    case $OPT in
      x)   # executable
        EXCTBLE=$OPTARG
        shift
  ```

```
      ;;
    b)   # directory holding paw executables
      PAWXDIR=$OPTARG
      shift
      ;;
    p)   # project name
      PROJECT=$OPTARG
      echo argument projectname=${NAME}
      shift
      ;;
    0)   # dry run only
      DRYRUN=yes
      echo option dry-run=${DRYRUN}
      shift
      ;;
    h)   # help
      echo -e $USAGE
      exit 1
      ;;
    \?)   # unknown option (placed into OPTARG, if OPTSTRING starts with :)
      echo "error in $0" >&2
      echo "invalid option -$OPTARG" >&2
      echo "retrieve argument list with:" >&2
      echo "$0 -h" >&2
      exit 1
      ;;
    :)    # no argument passed to option requiring one
      echo "error in $0" >&2
      echo "option -$OPTARG requires an additional argument" >&2
      exit 1
      ;;  esac
  esac
done
shift $(($OPTIND - 1)) # shift so that following arguments are $1, $2, etc.
if [ -z $1 ] ; then echo ‘‘error in $0: missing argument’’ >&2
ARG=$1
```

- check if all mandatory arguments have been passed.
- for every error, that has been captured, exit with a non-zero return code, i.e. by `exit 1`, and issue an error message to “error out”=&2.

  ```
  echo "error in $0: message" >&2
  exit 1
  ```

- finish the script with `exit 0`

1.1.1 List of recommended option id's

The following list shall not be used, except with the meaning described here. Some of these choices are inspired by http://www.faqs.org/docs/artu/ch10s05.html.

a All.

c name of the control file

f input file (other than a control file)

l list

o output file

p root name of the paw project

e executable

b directory holding the executables (to select a specific paw distribution)

0 dry run

v verbose

V version

q quiet

h issue help message

1.1.2 Brief description of getopts

The bash command getopts process the argument list in a standardized manner and allows for automatized error handling.

The bash command

`getopts` $OPTSTRING OPT

processes an option string OPTSTRING, and returns true or false depending of whether it encountered a valid option in teh calling sequence of the calling bash script. It returns the id of the option as $OPT and it sets the variable OPTARG with the argument of the option. In case of an error OPTARG contains the name of the option, if OPTSTRING starts with a colon ":".

The option string is a string of option letters. An option with an argument is followed by a colon ":". An initial ":" switches getopts into the quiet mode, which also changes the error handling. Therefore capture all errors and work in quiet mode.

- A double dash "–" signifies the end of the options
- Options may be grouped such as `-abc` which is identical to `-a -b -c`.
- Options may only be single letters or numerals.
- OPTIND is another variable used by getopts and it identifies the number of items in the argument list that has been processed by getopts. With the command `shift $(($OPTIND - 1))` the first item following the option list is `$1`.

1.1.3 Default environment

- the current PAW directory can be obtained via

```
export PAWXDIR=$(which paw_fast.x); PAWXDIR=${PAWXDIR%paw_fast.x}
```

- The current directory is captured with

```
THISDIR=$(pwd)   # current directory
```

1.1.4 Other shell rules

- Make temporary files: Check first if the environment variable TMPDIR is set and use this one. If it is not set, use /tmp. Create temporary file using mktemp. Check error code of mktemp. Delete the file when done.
- do not use non-printing characters nor include them in the files. Use tabs only, when the syntax requires it, such as in make.
- indent with two spaces for each level.

1.2 Fortran codes

- I start with code at column 7 similar to the ancient fixed file format of Fortran. The first few spaces are for code that is written for debugging only and that should be removed in the final code.
- Keep lines within 80 characters. *Rationale: Longer lines usually break when they are printed, which makes the printout ugly.*
- The general style for subroutines is as follows. (the line with the numbers does not belong to the style.)

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
!
!     ...1.........2.........3.........4.........5.........6.........7.........8
      Subroutine doing(n,b,c)
!     ************************************************************************
!     **  description of doing here                                         **
!     ************************************************************************
      implicite none
      integer(4),intent(in) :: n
      real(8)   ,intent(in) :: a(n) !comment on variable a here
      real(8)   ,intent(out):: b(n)
!     ************************************************************************
!
!     ========================================================================
!     == section comment                                                    ==
```

```
!     ========================================================================
!
!     ==minor structuring comment============================================

!     __explaining comment/remark___________________________________________

print*,'a ',a   ! statement used for testing

      return
      end
```

- modules are not indented to the seventh column but are typed starting at the first column.
- Always use `implicit none`. Try to start the names of integers with letters, i,j,k,l,m,n, and the name of logical variables with t. *Rationale: with "implicit none", typos become apparent quickly.*
- For all variables in the argument list, declare the intent as `intent(in)`, `intent(out)` or `intent(inout)`.
- declare one variable per line. Do note use "DIMENSION", but use

```
real(8)              :: a(n)    ! good
real(8),dimension(n) :: a       ! bad
```

  Rationale: Dimensions are declared more efficiently together with the variable. The explicit dimension statement takes space away that can be used better for comments.

- Specify the used module variables explicitly using the `only` statement. Preferably break the line after each variable in the `only`-list, so that a descriptive comment can be added.

```
 use xxx_module, only : variable1 & !(n1,n2) first variable
&                      ,variable2   ! second variable
```

- Keep the following sequence in the declaration section of the code.
  1. `use` statements
  2. `implicit none`
  3. Derived type definitions
  4. declarations of variables in the argument list
  5. declare parameters
  6. declare other variables
  7. keep loop variables towards the end of the declaration section.
- variable and subroutine names shall be as descriptive as possible.

- avoid functions in favor of subroutines. *Rationale: Functions calls can easily be mistaken as variables. Subroutines offer the same functionality as functions, but are more explicit. Functions offer two competing ways of returning variables, which seems to be a design flaw.*
- define parameters such as as tolerances, maximum iteration numbers with the parameter statement in the declaration section of a subroutine, and NOT in the code section itself.
- do not introduce size limitations, if they can change. Instead allocate arrays dynamically.
- if an error has been detected, let the code stop. Give it a sensible error message if possible. If the crash is due to a user error, provide info on how to correct it.
- Continuation lines: use a & both at the end of the continued line and at the beginning (column 6) of the continuing line.
- Place a connecting symbol such as an operator "+","*", etc. or separator ",", "//" on the continued line instead of the continuing line. *Rationale: This symbol has only meaning with the following element and it looses its meaning when the continuing element is removed. Thus it belongs conceptually to the latter.*
- write any comments in english.
- write the code so that it can be made uppercase without changing its meaning. Thus the value string variable must be uppercase, or it must be made lowercase for example using the string object of the PAW library. Case conversion can be done with the operators "+" and "-" defined in the paw_strings object of the paw code. *Rationale: Fortran is case insensitive. To be consistent, case conversion shall not change the meaning of the code. Some programmers prefer to type uppercase and others lowercase. Case conversion shall therefore be without consequence.*
- use descriptive filenames
- add a comment to each "enddo" or "end if", if the loop or if statement has been long, such as

```
enddo   ! end of loop over atoms (iat)
```

- Never use special characters or blanks in a file name. *Rationale: These characters cannot be easily identified with any editor. Blanks act as separator for shell arguments. A blank must escaped with a backslash. Thus files in blanks are a common source of error.*
- statements that are inserted for debugging purposes start in the first instead of the 7-th column. *Rationale: Statements starting in he first column are easily identified and removed, when they become obsolete.*

1.2.1 Object oriented coding

An object is a fortran module holding a set of data and a set of subroutines operating on them. (It could be a class in ++)

The main feature of an object is that it exposes only a well defined and minimal interface to the outside and hides data and internal technicalities from the "user" of the object.

Data will be exchanged almost exclusively during function calls or using explicit interfaces such as

```
subroutine objnm$setr8a(id,len,val)
use objnm_module, only : this,that
implicit none
character(*),intent(in) :: id
real(8)     ,intent(in) :: val(len)
if(id.eq.'this') then
  thisdata=val
else if(id.eq.'that') then
  thatdata=val
else
  call error$msg('id not recognized')
  call error$chval('id',id)
  call error$r8val('val',val)
  call error$stop('objnm$setr8')
end
return
end
```

- For scalar variables the letter "a" is dropped from the name and the variable "len".
- "R8" stands for real(8). Other possibilities are "I4" (integer(4)), "L4" logical(4), "C8" complex(8).
- in addition to the set routines there are analogous get routines.

The object modul MUST NEVER be used outside the object!

Each object has a name such as `objnm`. The corresponding object module has the name `objnm_module`. The subrouitines that may be used externally, wave names `objnm\$subname`, where the dollar sign separates the object name from the function name. The dollar sign will be replaced by a double underscore during compilation. Subroutines that are to be used only from subroutines of the object, will be named as `objnm\_subnm`, where the function part is separated by an underscore from the main part.

Chapter 2

Code analysis using Doxygen

Doxygen is a free multi-platform code analysis and documentation tool. Doxygen can be downloaded from www.doxygen.org.

A description on how to use it can be found in http://www.msg.chem.iastate.edu/gamess/DoxygenRules

Lines starting with "!>" are interpreted as Doxygen comments

Doxygen knows a couple of keywords:

- @brief: One-sentence description of the subroutine
- @details: Detailed information about the subroutine
- @note: special notes for the user
- @see
- @warning
- @todo
- @bug
- @param *[dir] argument-name description*: in and output variables. Dir can be "in" or "out".
- @author
- @date *Month,Year, Author's name*: used for Bugfixes
- `\cite` *label*: refers to latex bibliography identifiers specified in `CITE_BIB_FILES`

2.0.1 Markdown language

2.0.2 Ideas to automatically convert code into doxygen comments

- the PAW comments should follow the rules of the markdown language
- A comment following an variable specification carrying the intent(in),intent(out) or intent(inout) is the variable description and is tagged by the @param keyword.
- A subroutine comment is characterized by `!  **` and follows the subroutine statement

- the first sentence goes into @brief
- the rest, not identified by some other keyword goes into @details
- note,see,todo,bug is parsed and transformed into Doxygen statements
- @author is grabbed from the last line of the subroutine comment

Bibliography