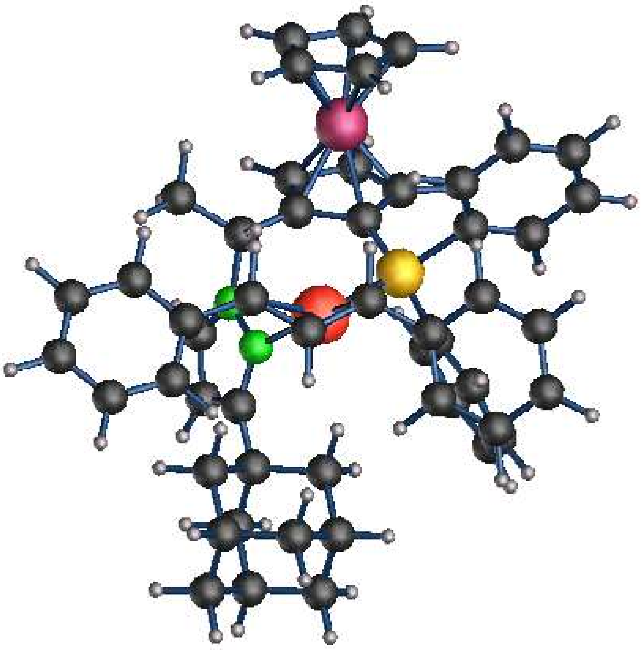
CP-PAW
Coding Style Guide



Peter E. Blöchl, Clausthal University of Technology

(April 1, 2014)

Preface

In this document I will try to develop the coding styles that teh CP-PAW code adheres to. It is meant to make the code more accessable and it should also lay down a basis for joint development of the code.

Peter Blöchl

Contents

Chapter 1

Coding design rules

1.1 General coding design rules

1.1.1 Visual structure

An example for a subroutine is this *“hello world”* routine.

```
!
!     ...1.........2.........3.........4.........5.........6.........7.........8
      SUBROUTINE HELLOWORLD(NFIL)
!     **************************************************************************
!     **  WRITES HELLO WORLD TO A FILE                                        **
!     **************************************************************************
      IMPLICIT NONE
      INTEGER(4)  ,INTENT(IN) :: NFIL     ! FORTRAN UNIT
      CHARACTER(8)            :: HELLOWORLD
!     **************************************************************************
!
!     ==========================================================================
!     ==  define                                                              ==
!     ==========================================================================
      HELLOWORLD='HELLO WORLD!'
!
!     ==========================================================================
!     ==  WRITE HELLOW WORLD                                                  ==
!     ==========================================================================
      WRITE(NFIL,FMT='(A)')HELLOWORLD
      RETURN
      END
```

- I am using lines with 80 characters wide, because this size is still easily printed without breaking lines. Therefore, I begin coding creating a ruler,

  ```
  12345678901234567890123456789012345678901234567890123456789012345678901234567890
  !     ...1.........2.........3.........4.........5.........6.........7.........8
  ```

  which also serves a header line for each subroutine. The first line with the numbers only serves to construct the actual header line below. Then it is deleted.

- All comment lines will have the same length so that the width is always available during coding.

- Subroutine documentation follows directly the calling sequence and have the form

  ```
  !     **************************************************************************
  !     **  this is a comment                                                   **
  !     **************************************************************************
  ```

The variable declaration is finished by another stared line.

- Comments are that describe a task are denoted as

```
!      ========================================================================
!      ==  this is a comment                                                 ==
!      ========================================================================
```

- One line comments that refer only to the next one or two lines can have the form

```
!      == this is a comment ===================================================
!      __this is a comment_____________________________________________________
```

Indentation

- We begin a code line always on column 7 as in fortran77 and try to stop whenever possible on or before column 80. The motivation is that code used for debugging can be inserted starting in the first line, so that corruptions of the code are immediately obvious.
- `DO` loops and and `IF` statements are indented by two characters for each level. This is a compromise between visibility on the one hand and on the other hand the space requirements for commands.
- Continuation lines are indicated by an `&` at the end of the continued line as required by the the fortran90 standard and by another `&` in column 6 of the continued line. In Fortran 90 the first & in the continuation line is not required, but it enhances clarity if the code structues.

1.1.2 Declarations

- All variables are declared explicitely, hence we introduce an `IMPLICIT NONE`.
- Input and output variables are explicitely declared with `INTENT(IN)`, `INTENT(OUT)`, and `INTENT(INOUT)`. `INTENT(INOUT)` is avoided whenever possible, and it is never used, when the more specific statements are used.
- The dimension is never declared with the statement `DIMENSION`, bcause this is too bulky. Thus we use

  ```
  REAL(8) :: x(10)
  ```

  instead of

  ```
  REAL(8),dimension(10) :: x
  ```

- With the exception of do-loop variables or closely related variables, each variable is specified on one line, and, whenever suitable, followed by a comment explaining it.

  ```
  REAL(8) :: x(10)  ! coordinate array
  ```

- currently the default sizes are `LOGICAL(4),INTEGER(4),REAL(8),COMPLEX(8)`. This is a compromize between being specific, and compatibility with 32-bit and 64-bit processors. (In future we may switch the `INTEGER(8)` or default integer, `INTEGER`, but this must be done consistently throughout the complete code.)

Chapter 2

Object-oriented programming

The main idea of object-oriented programming is to structure the code in terms of rather independent agents, also called objects. An agent consists of a set of functions (operations) and the memory needed to perform these functions. Thus a higher level language is constructed, which essentially calls these agents and controls the communication between them. On the higher level the operations can be combined again into agents creating an even higher level language. This approach is scalable in complexity, and allows to efficiently develop and maintain large codes.

The agents need to be prepared as logical entities. What this means is rarely clear and a good code is characterized by a good choice of agents. Some design rules are important:

- The same data should not be dublicated in different agents. This is not always possible, and if it is not one should at least identify one agent who is responsible for a certain variable. Otherwise the value of that data may become unsynchronized.
- The amount of external information required by an agent to perform its function, should be minimized.
- The agents and functions should be intuitive.

2.1 Encapsulation

The design of an object distinguishes clearly between what is accessable towards the outside of the object and the internal operations of the object. The latter should be made invisible to the user. This concept is called *encapsulation* and is one of the main concepts of object oriented programming.

An object contains two sets of subroutines: those that are available to the user of the object and those that are required used only by other routines of the object. To make this evident I introduced the following naming convention.

- *objectname*`$`*functionname* is a subroutine that can be accessed from the outside
- *objectname_functionname* is a subroutine that can be only from inside the object.
- *objectname*`$_module` or *objectname*`$_interface` is the module of the object. The module contains the permanent data of the object, declarations of common data structures and the like. At the moment there is no clear distinction between internal and external module files. Type declarations, definition of overloaded operations may need to be communicated to the outer world, while the internal data should be accessable only by calling subroutines of the object.

2.2 Design of a general object

An Object consists of a module holding general data and a set of functions (subroutines). The module holding general data is named `OBJECT_MODULE`, where `OBJECT` stands is a place holder of the object name.

The functions of teh object can be public and private. The public functions can be accessed from everywhere, which the private functions must only be called from within the object. The public functions are named

OBJECT$FUNCTION and the private functions are named OBJECT_FUNCTION, where FUNCTION is a place holder for the function name.

Data will be exchanged with the object via functions

```
OBJECT$GETCH(ID,VAL)       ! STRINGS
OBJECT$GETL4(ID,VAL)       ! LOGICAL SCALARS
OBJECT$GETI4(ID,VAL)       ! INTEGER SCALARS
OBJECT$GETR8(ID,VAL)       ! REAL SCALARS
OBJECT$GETC8(ID,VAL)       ! COMPLEX SCALARS
OBJECT$GETI4A(ID,LEN,VAL) ! INTEGER ARRAYS
OBJECT$GETR8A(ID,LEN,VAL) ! REAL ARRAYS
OBJECT$GETC8A(ID,LEN,VAL) ! COMPLEX ARRAYS
```

There is a similar set of functions where GET is replaced by SET. ID is an character identifier, which is checked and LEN is the size of the array. The length is explicitely checked for the OBJECT$GET.. functions. For the set functions the length may be used to set a variable determining the size, but it is tested if this size variable has not been set otherwise.

It is sometimes neccesary to make function interfaces publically available, for example for generic subroutine names. In this case a second module with name OBJECT$INTERFACE is created.

2.3 Design of a dynamical object

The basic iteration is done by three functions

```
OBJECT$ETOT()
OBJECT$PROPAGATE()
OBJECT$SWITCH()
```

Usually the object has to set itself up before the iteration can begin. This is done by a public function

```
OBJECT$INITIALIZE()
```

Such a function is provided if internal data need to be made available externally. Preferred is an internal function OBJECT_INITIALIZE(), that is called whenever needed as decided by an internal flag TINI of type logical.

For reading and writing the data to a restart file the functions

```
OBJECT$READ(NFIL,NFILO,TCHK)
OBJECT$WRITE(NFIL,NFILO,TCHK)
```

are provided.

The setting of the object is reported by the function

```
OBJECT$REPORT(NFIL)
```

The object has a number of parameters that allow to control its functionality.

- STOP; logical; sets initial velocities to zero.
- DT; real; timestep
- FRICTION; real; friction parameter $\alpha\Delta/2$
- ekin; kinetic energy
- epot; potential energy
- force; generalized force, $-\frac{dE}{dx}$
- xp