Developer's Manual for Quantum ESPRESSO(v. 5.0.1)

Contents

1 Introduction

1.1 Who should read (and who should *write*) this guide

The intended audience of this guide is everybody who wants to:

- know how QUANTUM ESPRESSO works, including its internals;
- modify/customize/add/extend/improve/clean up QUANTUM ESPRESSO;
- know how to read data produced by QUANTUM ESPRESSO.

The same category of people should also *write* this guide, of course.

1.2 Who may read this guide but will not necessarily profit from it

People who want to know about the capabilities of QUANTUM ESPRESSO, or who want just to use it, should read the User Guide.

People who want to know about the methods or the physics behind QUANTUM ESPRESSO should read first the relevant literature (some pointers in the User Guide).

1.3 How to contribute to QUANTUM ESPRESSO

You can contribute to a better QUANTUM ESPRESSO, even if you are not a full-fledged developer, by

- answering other people's questions on the mailing list (correct answers are strongly preferred to wrong ones).
- porting to new/unsupported architectures or configurations: see section 2.1, "Installation mechanism". You should not need to add new preprocessing flags, but if you do, see section 4.1, "Preprocessing".

- pointing out bugs in the software and in the documentation (reports of real bugs are strongly preferred to reports of nonexistent bugs). See section 1.7, "Guidelines for reporting bugs".

- improving the documentation (generic complaints or suggestions that "there should be this and that" do not qualify as improvements).

- suggesting changes: note however that suggestions requiring a significant amount of work are welcome only if accompanied by implementation or by a promise of future implementation (fulfilled promises are strongly preferred to forgotten ones).

- adding new features to the code. If you like to have something added to QUANTUM ESPRESSO, contact the developers via the `q-e-developers` mailing list, hosted on `qe-forge.org`. Unless there are technical reasons not to include your changes, we will try to make you happy (no warranty that we will actually succeed).

1.4 How to make major contributions to QUANTUM ESPRESSO

If you want to get involved as a developer and contribute serious or nontrivial stuff, you should register for the QUANTUM ESPRESSO project on `qe-forge.org`. You may also consider the idea of opening your own project on `qe-forge.org`. There is a developers' mailing list, `q-e-developers`, to which you should subscribe, and a mailing list receiving commit message, `q-e-commits`. The ideal procedure to add extensive changes is as follows:

- download the current SVN version (see section 9, "Using SVN", and work on that version

- when you are happy with your modified version, make a copy of it, then update your copy with `svn update`

- if you get no conflicts and everything is still working, you have won. Send the modified files to the developers, or save your changes if you are one.

- if you get conflicts, or if the updated code doesn't work any longer, you haven't yet won. See section 9.2 for hints on how to remove conflicts and on how to figure out what went wrong.

1.5 About `qe-forge.org`

`qe-forge.org` is the portal for QUANTUM ESPRESSO developers, contributors, and for anybody else wanting to develop a project in the field of atomistic simulations. `qe-forge.org` provides a CVS or SVN repository, mailing lists, a wiki, upload space, a bug tracking facility, various other tools that are useful for developers. You can use either CVS or SVN but not both together. Note that the usage of the wiki provided by `qe-forge.org` is currently disabled for security reasons.

You can open your own project, retaining all rights on it (including the right not to release anything); or else, you can register as a developer in an existing project (or both).

Currently QUANTUM ESPRESSO uses the following development tools:

- SVN server

- Bug Tracking facility
- Upload space (with download counter)
- Mailing lists `q-e-commits` and `q-e-developers`.

Everybody is encouraged to explore other capabilities of `qe-forge.org`.

Once you are registered, you need to register your SSH keys in order to have read-write access the CVS or SVN repository (if you have been allowed by the project leader). The procedure is as follows:

- login to your `qe-forge.org` account
- click on My account (top right corner)
- click on Edit Keys, follow the instructions to add your keys
- go back to your home page, click on SCM, follow instruction

Specific to the QUANTUM ESPRESSO project:

- If you want to receive an e-mail at each commit, subscribe to `q-e-commits@qe-forge.org`. This will occasionally flood you with e-mails, because each changed directory generates an e-mail.
- If you want to be kept informed on what is going on, please subscribe to `q-e-developers@qe-forge` (low traffic)

1.6 Hints, Caveats, Do's and Dont's for developers

- Before doing anything, inquire whether it is already there, or under development.
- Before starting writing code, inquire whether you can reuse code that is already available in the distribution. Avoid redundancy: the only bug-free software line is the one that doesn't exist.
- When you make some changes:
  - Check that are not spoiling other people's work. In particular, search the distribution for codes using the routine or module you are modifying and change its usage or its calling arguments everywhere.
  - Do not forget that your changes must work on many different combinations of hardware and software, in both serial and parallel execution.
  - Do not forget that your changes must work for a wide variety of different system: if your code works only in some selected case, it must either stop or issue a warning in all other cases.
  - Do not forget that your changes must work on systems of wildly different computational size: solutions that may look appropriate in crystal silicon may gobble a disproportionate amount of time and/or memory in a 1000-atom cell.
- Document your changes:

  - If you modify what a code can do, or introduce incompatibilities with previous versions (e.g. old data file no longer readable, old input no longer valid), report it in file `Doc/release-notes`.
  - If you add/modify/remove input variables, document it in the appropriate file `INPUT_*.def`; if you remove an input variable, update tests and examples accordingly.
  - All newly introduced features or variables must be accompanied by an example or a test or both (either a new one or a modified existing test or example).

- Please do not include files (any kind, including pseudopotential files) with DOS ^M characters or tabulators ^I.

When you modify the program sources, run the `install/makedeps.sh` script or type `make depend` to update files `make.depend` in the various subdirectories.

1.7 Guidelines for reporting bugs

- Before deciding that a problem is due to a bug in the codes, verify if it is reproducible on different machines/architectures/phases of the moon: erratic or irreproducible problems, especially in parallel execution, are often an indication of buggy compilers or libraries
- Bug reports should preferably be filed using the bug tracking facility at `qe-forge.org`: `http://qe-forge.org/tracker/?group_id=10`
- Bug reports should include enough information to be reproduced: typically, version number, hardware/software combination(s) for which the problem arises, whether it happens in serial or parallel execution or both, and, most important, an input and output exhibiting such behavior (fast to execute if possible). The error message alone is usually not a sufficient piece of information.
- If a bug is found in a stable (released) version of QUANTUM ESPRESSO, it must be reported in the `Doc/release-notes` file.

2 Structure of the distribution

2.0.1 Libraries

Subdirectory `flib/` contains libraries written in fortran77 (`*.f`) and in fortran-90 (`*.f90`). The latter should not depend on any module, except for modules `kinds` and `constants`.

Subdirectory `clib/` contains libraries written in C (`*.c`). Functions that are called by fortran should be preprocessed using the macros:

1. `F77_FUNC (func,FUNC)` for function `func`, not containing underscore(s) in name
2. `F77_FUNC_(f_nc,F_NC)` for function `f_nc`, containing underscore(s) in name

These macros are defined in file `include/c_defs.h`. This file must be included by all `*.c` files. The macros are automagically generated by `configure` and choose the correct case (lowercase or uppercase) and the correct number of final underscores. See file `include/defs.h.README` for more info.

2.1 Installation mechanism

The code contains C-style preprocessing directives. There are two ways to do preprocessing of fortran files:

- directly with the fortran compiler, if supported;
- by first pre-compiling with the C preprocessor `cpp`.

In the first case, one needs to specify in the `make.sys` file the fortran compiler option that tells the compiler to pre-process first. In the second case, one needs to specify the C precompiler and options (if needed) in `make.sys`. Normally, `configure` should take care of this.

2.1.1 How to edit the `configure` script

The `configure` script is generated from its source file `configure.ac` by the GNU `autoconf` utility (`http://www.gnu.org/software/autoconf/`). Don't edit `configure` directly: whenever it gets regenerated, your changes will be lost. Instead, go to the `install/` directory, edit `configure.ac`, then run `autoconf` to regenerate `configure`. If you want to keep the old `configure`, make a copy first.

GNU `autoconf` is installed by default on most Unix/Linux systems. If you don't have it on your system, you'll have to install it. You will need a recent version (e.g. v.2.65) of `autoconf`, because our `configure.ac`file uses recent syntax.

`configure.ac` is a regular Bourne shell script (i.e., "sh" – not csh!), except that:

- capitalized names starting with "AC_" are `autoconf` macros. Normally you shouldn't have to touch them.
- square brackets are normally removed by the macro processor. If you need a square bracket (that should be very rare), you'll have to write two.

You may refer to the GNU `autoconf` Manual for more info.

`make.sys.in` is the source file for `make.sys`, that `configure` generates: you might want to edit that file as well. The generation procedure is as follows: if `configure.ac` contains the macro "AC_SUBST(name)", then every occurrence of "@name@" in the source file will be substituted with the value of the shell variable "name" at the point where AC_SUBST was called.

Similarly, `configure.msg` is generated from `configure.msg.in`: this file is only used by `configure` to print its final report, and isn't needed for the compilation. We did it this way so that our `configure` may also be used by other projects, just by replacing the QUANTUM ESPRESSO-specific `configure.msg.in` by your own.

`configure` writes a detailed log of its operation to `config.log`. When any configuration step fails, you may look there for the relevant error messages. Note that it is normal for some checks to fail.

2.1.2 How to add support for a new architecture

In order to support a previously unsupported architecture, first you have to figure out which compilers, compilation flags, libraries etc. should be used on that architecture. In other words, you have to write a `make.sys` that works: you may use the manual configuration procedure for

that (see the User Guide). Then, you have to modify `configure` so that it can generate that `make.sys` automatically.

To do that, you have to add the case for your architecture in several places throughout `configure.ac`:

1. Detect architecture

   Look for these lines:

   ```
   if test "$arch" = ""
   then
           case $host in
                   ia64-*-linux-gnu )      arch=ia64   ;;
                   x86_64-*-linux-gnu )    arch=x86_64 ;;
                   *-pc-linux-gnu )        arch=ia32   ;;
                   etc.
   ```

   Here you must add an entry corresponding to your architecture and operating system. Run `config.guess` to obtain the string identifying your system. For instance on a PC it may be "i686-pc-linux-gnu", while on IBM SP4 "powerpc-ibm-aix5.1.0.0". It is convenient to put some asterisks to account for small variations of the string for different machines of the same family. For instance, it could be "aix4.3" instead of "aix5.1", or "athlon" instead of "i686"...

2. Select compilers

   Look for these lines:

   ```
   # candidate compilers and flags based on architecture
   case $arch in
   ia64 | x86_64 )
         ...
   ia32 )
         ...
   aix )
         ...
   etc.
   ```

   Add an entry for your value of $arch, and set there the appropriate values for several variables, if needed (all variables are assigned some reasonable default value, defined before the "case" block):

   - "try_f90" should contain the list of candidate Fortran 90 compilers, in order of decreasing preference (i.e. configure will use the first it finds). If your system has parallel compilers, you should list them in "try_mpif90".

   - "try_ar", "try_arflags": for these, the values "ar" and "ruv" should be always fine, unless some special flag is required (e.g., -X64 With sp4).

   - you should define "try_dflags" if there is any "#ifdef" specific to your machine: for instance, on IBM machines, "try_dflags=-D__AIX" . A list of such flags can be found in file `include/defs.h.README`.

You shouldn't need to define the following: - "try_iflags" should be set to the appropriate "-I" option(s) needed by the preprocessor or by the compiler to locate *.h files to be included; try_iflags="-I../include" should be good for most cases

For example, here's the entry for IBM machines running AIX:

```
 aix )
      try_mpif90="mpxlf90_r mpxlf90"
      try_f90="xlf90_r xlf90 $try_f90"
      try_arflags="-X64 ruv"
      try_arflags_dynamic="-X64 ruv"
      try_dflags="-D__AIX -D__XLF"
      ;;
```

The following step is to look for both serial and parallel fortran compilers:

```
# check serial Fortran 90 compiler...
...
AC_PROG_F77($f90)
...
      # check parallel Fortran 90 compiler
...
      AC_PROG_F77($mpif90)
...
echo setting F90... $f90
echo setting MPIF90... $mpif90
```

A few compilers require some extra work here: for instance, if the Intel Fortran compiler was selected, you need to know which version because different versions need different flags.

At the end of the test,

- $mpif90 is the parallel compiler, if any; if no parallel compiler is found or if `--disable-parallel` was specified, $mpif90 is the serial compiler

- $f90 is the serial compiler

Next step: the choice of (serial) C and Fortran 77 compilers. Look for these lines:

```
# candidate C and f77 compilers good for all cases
try_cc="cc gcc"
try_f77="$f90"

case "$arch:$f90" in
*:f90 )
      ....
etc.
```

Here you have to add an entry for your architecture, and since the correct choice of C and f77 compilers may depend on the fortran-90 compiler, you may need to specify the f90 compiler as well. Again, specify the compilers in try_cc and try_f77 in order of decreasing preference. At the end of the test,

- $cc is the C compiler

- $f77 is the Fortran 77 compiler, used to compile *.f files (may coincide with $f90)

3. Specify compilation flags.

   Look for these lines:

```
# check Fortran compiler flags
...
case "$arch:$f90" in
ia64:ifort* | x86_64:ifort* )
      ...
ia64:ifc* )
      ...
etc.
```

   Add an entry for your case and define:

   - "try_fflags": flags for Fortran 77 compiler.

   - "try_f90flags": flags for Fortran 90 compiler. In most cases they will be the same as in Fortran 77 plus some others. In that case, define them as "$(FFLAGS) -something_else".

   - "try_fflags_noopt": flags for Fortran 77 with all optimizations turned off: this is usually "-O0". These flags must be used for compiling flib/dlamch.f (part of our version of Lapack): it won't work properly with optimization.

   - "try_ldflags": flags for the linking phase (not including the list of libraries: this is decided later).

   - "try_ldflags_static": additional flags to select static compilation (i.e., don't use shared libraries).

   - "try_dflags": must be defined if there is in the code any #ifdef specific to your compiler (for instance, -D__INTEL for Intel compilers). Define it as "$try_dflags -D..." so that pre-existing flags, if any, are preserved.

   - if the Fortran 90 compiler is not able to invoke the C preprocessor automatically before compiling, set "have_cpp=0" (the opposite case is the default). The appropriate compilation rules will be generated accordingly. If the compiler requires that any flags be specified in order to invoke the preprocessor (for example, "-fpp " – note the space), specify them in "pre_fdflags".

   For example, here's the entry for ifort on Linux PC:

```
ia32:ifort* )
        try_fflags="-O2 -tpp6 -assume byterecl"
        try_f90flags="\$(FFLAGS) -nomodule"
        try_fflags_noopt="-O0 -assume byterecl"
```

```
        try_ldflags=""
        try_ldflags_static="-static"
        try_dflags="$try_dflags -D__INTEL"
        pre_fdflags="-fpp "
        ;;
```

Next step: flags for the C compiler. Look for these lines:

```
case "$arch:$cc" in
*:icc )
      ...
*:pgcc )
      ...
etc.
```

Add an entry for your case and define:

- "try_cflags": flags for C compiler.

- "c_ldflags": flags for linking, when using the C compiler as linker. This is needed to check for libraries written in C, such as FFTW.

- if you need a different preprocessor from the standard one ($CC -E), define it in "try_cpp".

For example for XLC on AIX:

```
aix:mpcc* | aix:xlc* | aix:cc )
        try_cflags="-q64 -O2"
        c_ldflags="-q64"
        ;;
```

Finally, if you have to use a nonstandard preprocessor, look for these lines:

```
echo $ECHO_N "setting CPPFLAGS... $ECHO_C"
case $cpp in
      cpp) try_cppflags="-P -traditional" ;;
      fpp) try_cppflags="-P"              ;;
      ...
```

and set "try_cppflags" as appropriate.

4. Search for libraries

   To instruct `configure` to search for libraries, you must tell it two things: the names of libraries it should search for, and where it should search.

   The following libraries are searched for:

   - BLAS or equivalent. Some vendor replacements for BLAS that are supported by QUANTUM ESPRESSO are:

MKL on Linux, 32- and 64-bit Intel CPUs
ACML on Linux, 64-bit AMD CPUs
essl on AIX
SCSL on sgi altix
SUNperf on sparc

Moreover, ATLAS is used over BLAS if available.

- LAPACK or equivalent. Some vendor replacements for LAPACK that are supported by Quantum ESPRESSO are:

mkl on linux SUNperf on sparc

- FFTW (version 3) or another supported FFT library. The latter include:

essl on aix ACML on Linux, 64-bit AMD CPUs SUNperf on sparc

- the MASS vector math library on aix

- an MPI library. This is often automatically linked by the compiler

If you have another replacement for the above libraries, you'll have to insert a new entry in the appropriate place.

This is unfortunately a little bit too complex to explain. Basic info:
"AC_SEARCH_LIBS(function, name, ...)" looks for symbol "function" in library "libname.a". If that is found, "-lname" is appended to the LIBS environment variable (initially empty). The real thing is more complicated than just that because the "-Ldirectory" option must be added to search in a nonstandard directory, and because a given library may require other libraries as prerequisites (for example, Lapack requires BLAS).

3 Algorithms

3.1 Gamma tricks

In calculations using only the Γ point (k=0), the Kohn-Sham orbitals can be chosen to be real functions in real space, so that $\psi(G) = \psi^*(-G)$. This allows us to store only half of the Fourier components. Moreover, two real FFTs can be performed as a single complex FFT. The auxiliary complex function Φ is introduced: $\Phi(r) = \psi_j(r) + i\psi_{j+1}(r)$ whose Fourier transform $\Phi(G)$ yields
$\psi_j(G) = \frac{\Phi(G)+\Phi^*(G)}{2}, \psi_{j+1}(G) = \frac{\Phi(G)-\Phi^*(G)}{2i}$.

A side effect on parallelization is that G and $-G$ must reside on the same processor. As a consequence, pairs of columns with $G_{n'_1,n'_2,n'_3}$ and $G_{-n'_1,-n'_2,n'_3}$ (with the exception of the case $n'_1 = n'_2 = 0$), must be assigned to the same processor.

4 Structure of the code

4.1 Preprocessing

The code contains C-style preprocessing directives. Most fortran compilers directly support them; some don't, and preprocessing is "hand-made" by the makefile using the C preprocessor

`cpp`. The C preprocessor may:

- assign a value to a given expression. For instance, command `#define THIS that`, or the option in the command line: `-DTHIS=that`, will replace all occurrence of `THIS` with `that`.
- include file (command `#include`)
- expand macros (command `#define`)
- execute conditional expressions such as

```
#ifdef __expression
  ...code A...
#else
  ...code B...
#endif
```

  If "expression" is defined (with a `#define` command or from the command line with option `D__expression`), then `...code A...` is sent to output; otherwise `...code B...` is sent to output.

The file `include/defs.h.README` contains a list of definitions that are used in the code. In order to make preprocessing options easy to see, preprocessing variables should start with two underscores, as `__expression` in the above example. Traditionally "preprocessed" variables are also written in uppercase.

5 Format of arrays containing charge density, potential, etc.

The index of arrays used to store functions defined on 3D meshes is actually a shorthand for three indices, following the FORTRAN convention ("leftmost index runs faster"). An example will explain this better. Suppose you have a 3D array `psi(nr1x,nr2x,nr3x)`. FORTRAN compilers store this array sequentially in the computer RAM in the following way:

```
        psi(   1,   1,   1)
        psi(   2,   1,   1)
        ...
        psi(nr1x,   1,   1)
        psi(   1,   2,   1)
        psi(   2,   2,   1)
        ...
        psi(nr1x,   2,   1)
        ...
        ...
        psi(nr1x,nr2x,   1)
        ...
        psi(nr1x,nr2x,nr3x)
etc
```

Let `ind` be the position of the `(i,j,k)` element in the above list: the following relation

```
ind = i + (j - 1) * nr1x + (k - 1) *  nr2x * nr1x
```

holds. This should clarify the relation between 1D and 3D indexing. In real space, the `(i,j,k)` point of the FFT grid with dimensions `nr1` ($\leq$`nr1x`), `nr2` ($\leq$`nr2x`), , `nr3` ($\leq$`nr3x`), is

$$r_{ijk} = \frac{i-1}{nr1}\tau_1 + \frac{j-1}{nr2}\tau_2 + \frac{k-1}{nr3}\tau_3$$

where the τ_i are the basis vectors of the Bravais lattice. The latter are stored row-wise in the `at` array: τ_1 = `at(:, 1)`, τ_2 = `at(:, 2)`, τ_3 = `at(:, 3)`.

The distinction between the dimensions of the FFT grid, `(nr1,nr2,nr3)` and the physical dimensions of the array, `(nr1x,nr2x,nr3x)` is done only because it is computationally convenient in some cases that the two sets are not the same. In particular, it is often convenient to have `nrx1`=`nr1`+1 to reduce memory conflicts.

6 Parallelization

In parallel execution (MPI only), N independent processes are started (do not start more than one per processor!) that communicate via calls to MPI libraries. Each process has its own set of variables and knows nothing about other processes' variables. Variables that take little memory are replicated, those that take a lot of memory (wavefunctions, G-vectors, R-space grid) are distributed.

6.1 Tricks and pitfalls

- Replicated calculations may either be performed independently on each processor, or performed on one processor and broadcast to all others. The first approach requires less programming, but it is unsafe: in principle all processors should yield exactly the same results, if they work on the same data, but sometimes they don't (depending on the machine, compiler, and libraries). Even a tiny difference in the last significant digit can eventually cause serious trouble if allowed to build up, especially when a replicated check is performed (in which case the code may "hang" if the check yields different results on different processors). Never assume that the value of a variable produced by replicated calculations is exactly the same on all processors: when in doubt, broadcast the value calculated on a specific processor (the "root" processor) to all others.

- Routine `errore` should be called in parallel by all processors, or else it will hang

- I/O operations: file opening, closing, and so on, are as a rule performed only on processor `ionode`. The correct way to check for errors is the following:

```
IF ( ionode ) THEN
   OPEN ( ..., IOSTAT=ierr )
   ...
END IF
CALL mp_bcast( ierr, ... )
CALL errore( 'routine','error', ierr )
```

The same applies to all operations performed on a single processor, or a subgroup of processors: any error code must be broadcast before the check.

6.2 Data distribution

Quantum ESPRESSO employ arrays whose memory requirements fall into three categories.

- *Fully Scalable*: Arrays that are distributed across processors of a pool. Fully scalable arrays are typically large to very large and contain one of the following dimensions:
  - number of plane waves, npw (or max number, npwx)
  - number of Gvectors, ngm
  - number of grid points in the R space, dfft%nnr

  Their size decreases linearly with the number of processors in a pool.

- *Partially Scalable*: Arrays that are distributed across processors of the ortho or diag group. Typically they are much smaller than fully scalable array, and small in absolute terms for moderate-size system. Their size however increases quadratically with the number of atoms in the system, so they have to be distributed for large systems (hundreds to thousands atoms). Partially scalable arrays contain none of the dimensions listed above, two of the following dimensions:
  - number of states, nbnd
  - number of atomic states, natomwfc
  - number of projectors, nkb

  Their size decreases linearly with the number of processors in a ortho or diag group.

- *Nonscalable*: All the remaining arrays, that are not distributed across processors. These are typically small arrays, having dimensions like for instance:
  - number of atoms, nat
  - number of species of atoms, nsp

  The size of these arrays is independent on the number of processors.

7 File Formats

7.1 Data file(s)

Quantum ESPRESSO restart file specifications: Paolo Giannozzi scripsit AD 2005-11-11, Last modified by Andrea Ferretti 2006-10-29

7.1.1 Rationale

Requirements: the data file should be

- efficient (quick to read and write)
- easy to read, parse and write without special libraries
- easy to understand (self-documented)
- portable across different software packages
- portable across different computer architectures

Solutions:

- use binary I/O for large records
- exploit the file system for organizing data
- use XML
- use a small specialized library (iotk) to read, parse, write
- ensure the possibility to convert to a portable formatted file

Integration with other packages:

- provide a self-standing (code-independent) library to read/write this format
- the use of this library is intended to be at high level, hiding low-level details

7.1.2 General structure

Format name: QEXML
Format version: 1.4.0

The "restart file" is actually a "restart directory", containing several files and sub-directories. For CP/FPMD, the restart directory is created as "$prefix_$ndw/", where $prefix is the value of the variable "prefix". $ndw the value of variable ndw, both read in input; it is read from "$prefix_$ndr/", where $ndr the value of variable ndr, read from input. For PWscf, both input and output directories are called "$prefix.save/".

The content of the restart directory is as follows:

```
data-file.xml          which contains:
                       - general information that doesn't require large data set:
                         atomic structure, lattice, k-points, symmetries,
                         parameters of the run, ...
                       - pointers to other files or directories containing bulkier
                         data: grids, wavefunctions, charge density, potentials, ...

charge_density.dat     contains the charge density
spin_polarization.dat  contains the spin polarization (rhoup-rhodw) (LSDA case)
```

```
magnetization.x.dat
magnetization.y.dat    contain the spin polarization along x,y,z
magnetization.z.dat    (noncollinear calculations)
lambda.dat             contains occupations (Car-Parrinello dynamics only)
mat_z.1                contains occupations (ensemble-dynamics only)

<pseudopotentials>     A copy of all pseudopotential files given in input

<k-point dirs>         Subdirectories K00001/, K00002/, etc, one per k-point.
```

Each k-point directory contains:

```
    evc.dat                wavefunctions for spin-unpolarized calculations, OR
    evc1.dat
    evc2.dat               spin-up and spin-down wavefunctions, respectively,
                           for spin polarized (LSDA) calculations;
    gkvectors.dat          the details of specific k+G grid;
    eigenval.xml           eigenvalues for the corresponding k-point
                           for spin-unpolarized calculations, OR
    eigenval1.xml          spin-up and spin-down eigenvalues,
    eigenval2.xml          for spin-polarized calculations;
```

in a molecular dynamics run, also wavefunctions at the preceding time step:

```
    evcm.dat               for spin-unpolarized calculations OR
    evcm1.dat
    evcm2.dat              for spin polarized calculations;
```

- All files "*.xml" are XML-compliant, formatted file;
- Files "mat_z.1", "lambda.dat" are unformatted files, containing a single record;
- All other files "*.dat", are XML-compliant files, but they contain an unformatted record.

7.1.3 Structure of file "data-file.xml"

```
XML Header: whatever is needed to have a well-formed XML file

Body: introduced by <Root>, terminated by </Root>. Contains first-level tags
      only. These contain only other tags, not values. XML syntax applies.

First-level tags: contain either
     second-level tags, OR
     data tags:   tags containing data (values for a given variable), OR
     file tags:   tags pointing to a file
```

data tags syntax ([...] = optional) :

```
      <TAG type="vartype" size="n" [UNIT="units"] [LEN="k"]>
      values (in appropriate units) for variable corresponding to TAG:
      n elements of type vartype (if character, of lenght k)
      </TAG>
```

where TAG describes the variable into which data must be read;
"vartype" may be "integer", "real", "character", "logical";
if type="logical", LEN=k" must be used to specify the length of the variable character; size="n" is the dimension.
Acceptable values for "units" depend on the specific tag.

Short syntax, used only in a few cases:

```
<TAG attribute="something"/> .
```

For instance:

```
<FFT_GRID nr1="NR1" nr2="NR2" nr3="NR3"/>
```

defines the value of the FFT grid parameters nr1, nr2, nr3 for the charge density

7.1.4 Sample

Header:

```
<?xml version="1.0"?>
<?iotk version="1.0.0test"?>
<?iotk file_version="1.0"?>
<?iotk binary="F"?>
```

These are meant to be used only by iotk (actually they aren't)

First-level tags:

```
 - <HEADER>          (global information about fmt version)
 - <CONTROL>         (miscellanea of internal information)
 - <STATUS>          (information about the status of the CP simulation)
 - <CELL>            (lattice vector, unit cell, etc)
 - <IONS>            (type and positions of atoms in the unit cell etc)
 - <SYMMETRIES>      (symmetry operations)
 - <ELECTRIC_FIELD> (details for an eventual applied electric field)
 - <PLANE_WAVES>     (basis set, cutoffs etc)
 - <SPIN>            (info on spin polarizaztion)
 - <MAGNETIZATION_INIT>     (info about starting or constrained magnetization)
 - <EXCHANGE_CORRELATION>
 - <OCCUPATIONS>     (occupancy of the states)
 - <BRILLOUIN_ZONE> (k-points etc)
 - <PHONON>          (info for phonon calculations)
 - <PARALLELISM>     (specialized info for parallel runs)
 - <CHARGE-DENSITY>
 - <TIMESTEPS>       (positions, velocities, nose' thermostats)
 - <BAND_STRUCTURE_INFO>    (dimensions and basic data about band structure)
 - <EIGENVALUES>     (eigenvalues and related data)
 - <EIGENVECTORS>    (eigenvectors and related data)


* Tag description
```

```
<HEADER>
   <FORMAT>    (name and version of the format)
   <CREATOR>   (name and version of the code generating the file)
</HEADER>

<CONTROL>
   <PP_CHECK_FLAG>    (whether the file can be used for post-processing)
   <LKPOINT_DIR>      (whether kpt-data are written in sub-directories)
   <Q_REAL_SPACE>     (whether augmentation terms are used in real space)
</CONTROL>

<STATUS>  (optional)
   <STEP>   (number $n of steps performed, i.e. we are at step $n)
   <TIME>   (total simulation time)
   <TITLE>  (a job descriptor)
   <ekin>   (kinetic energy)
   <eht>    (hartree energy)
   <esr>    (Ewald term, real-space contribution)
   <eself>  (self-interaction of the Gaussians)
   <epseu>  (pseudopotential energy, local)
   <enl>    (pseudopotential energy, nonlocal)
   <exc>    (exchange-correlation energy)
   <vave>   (average of the potential)
   <enthal> (enthalpy: E+PV)
</STATUS>

<CELL>
   <BRAVAIS_LATTICE>
   <LATTICE_PARAMETER>
   <CELL_DIMENSIONS>  (cell parameters)
   <DIRECT_LATTICE_VECTORS>
      <UNITS_FOR_DIRECT_LATTICE_VECTORS>
      <a1>
      <a2>
      <a3>
   <RECIPROCAL_LATTICE_VECTORS>
      <UNITS_FOR_RECIPROCAL_LATTICE_VECTORS>
      <b1>
      <b2>
      <b3>
</CELL>

<IONS>
   <NUMBER_OF_ATOMS>
   <NUMBER_OF_SPECIES>
   <UNITS_FOR_ATOMIC_MASSES>
```

```
   For each $n-th species $X:
      <SPECIE.$n>
         <ATOM_TYPE>
         <MASS>
         <PSEUDO>
      </SPECIE.$n>
   <PSEUDO_DIR>
   <UNITS_FOR_ATOMIC_POSITIONS>
   For each atom $n of species $X:
      <ATOM.$n SPECIES="$X">
</IONS>

<SYMMETRIES>
   <NUMBER_OF_SYMMETRIES>
   <INVERSION_SYMMETRY>
   <NUMBER_OF_ATOMS>
   <UNITS_FOR_SYMMETRIES>
   For each symmetry $n:
      <SYMM.$n>
         <INFO>
         <ROTATION>
         <FRACTIONAL_TRANSLATION>
         <EQUIVALENT_IONS>
      </SYMM.$n>
</SYMMETRIES>

<ELECTRIC_FIELD>  (optional)
   <HAS_ELECTRIC_FIELD>
   <HAS_DIPOLE_CORRECTION>
   <FIELD_DIRECTION>
   <MAXIMUM_POSITION>
   <INVERSE_REGION>
   <FIELD_AMPLITUDE>
</ELECTRIC_FIELD>

<PLANE_WAVES>
   <UNITS_FOR_CUTOFF>
   <WFC_CUTOFF>
   <RHO_CUTOFF>
   <MAX_NUMBER_OF_GK-VECTORS>
   <GAMMA_ONLY>
   <FFT_GRID>
   <GVECT_NUMBER>
   <SMOOTH_FFT_GRID>
   <SMOOTH_GVECT_NUMBER>
   <G-VECTORS_FILE>       link to file "gvectors.dat"
   <SMALLBOX_FFT_GRID>
```

```
</PLANE_WAVES>

<SPIN>
   <LSDA>
   <NON-COLINEAR_CALCULATION>
   <SPIN-ORBIT_CALCULATION>
   <SPIN-ORBIT_DOMAG>
</SPIN>

<EXCHANGE_CORRELATION>
   <DFT>
   <LDA_PLUS_U_CALCULATION>
   if LDA_PLUS_U_CALCULATION
      <NUMBER_OF_SPECIES>
      <HUBBARD_LMAX>
      <HUBBARD_L>
      <HUBBARD_U>
      <HUBBARD_ALPHA>
   endif
</EXCHANGE_CORRELATION>

if hybrid functional
    <EXACT_EXCHANGE>
      <x_gamma_extrapolation>
      <nqx1>
      <nqx2>
      <nqx3>
      <exxdiv_treatment>
      <yukawa>
      <ecutvcut>
      <exx_fraction>
      <screening_parameter>
    </EXACT_EXCHANGE>
endif

<OCCUPATIONS>
   <SMEARING_METHOD>
   if gaussian smearing
      <SMEARING_TYPE>
      <SMEARING_PARAMETER>
   endif
   <TETRAHEDRON_METHOD>
   if use tetrahedra
      <NUMBER_OF_TETRAHEDRA>
      for each tetrahedron $t
         <TETRAHEDRON.$t>
   endif
```

```
   <FIXED_OCCUPATIONS>
   if using fixed occupations
      <INFO>
      <INPUT_OCC_UP>
      if lsda
         <INPUT_OCC_DOWN>
      endif
   endif
</OCCUPATIONS>

<BRILLOUIN_ZONE>
   <NUMBER_OF_K-POINTS>
   <UNITS_FOR_K-POINTS>
   <MONKHORST_PACK_GRID>
   <MONKHORST_PACK_OFFSET>
   For each k-point $n:
      <K-POINT.$n>
</BRILLOUIN_ZONE>

<PHONON>
   <NUMBER_OF_MODES>
   <UNITS_FOR_Q-POINT>
   <Q-POINT>
</PHONON>

<PARALLELISM>
   <GRANULARITY_OF_K-POINTS_DISTRIBUTION>
</PARALLELISM>

<CHARGE-DENSITY>
    link to file "charge_density.rho"
</CHARGE-DENSITY>

<TIMESTEPS>  (optional)
   For each time step $n=0,M
     <STEP$n>
        <ACCUMULATORS>
        <IONS_POSITIONS>
           <stau>
           <svel>
           <taui>
           <cdmi>
           <force>
        <IONS_NOSE>
           <nhpcl>
           <nhpdim>
           <xnhp>
```

```
            <vnhp>
          <ekincm>
          <ELECTRONS_NOSE>
             <xnhe>
             <vnhe>
          <CELL_PARAMETERS>
             <ht>
             <htve>
             <gvel>
          <CELL_NOSE>
             <xnhh>
             <vnhh>
          </CELL_NOSE>
</TIMESTEPS>

<BAND_STRUCTURE_INFO>
   <NUMBER_OF_BANDS>
   <NUMBER_OF_K-POINTS>
   <NUMBER_OF_SPIN_COMPONENTS>
   <NON-COLINEAR_CALCULATION>
   <NUMBER_OF_ATOMIC_WFC>
   <NUMBER_OF_ELECTRONS>
   <UNITS_FOR_K-POINTS>
   <UNITS_FOR_ENERGIES>
   <FERMI_ENERGY>
</BAND_STRUCTURE_INFO>

<EIGENVALUES>
   For all kpoint $n:
       <K-POINT.$n>
           <K-POINT_COORDS>
           <WEIGHT>
           <DATAFILE>                  link to file "./K$n/eigenval.xml"
       </K-POINT.$n>
</EIGENVALUES>

<EIGENVECTORS>
   <MAX_NUMBER_OF_GK-VECTORS>
   For all kpoint $n:
       <K-POINT.$n>
           <NUMBER_OF_GK-VECTORS>
           <GK-VECTORS>                link to file "./K$n/gkvectors.dat"
           for all spin $s
              <WFC.$s>                 link to file "./K$n/evc.dat"
              <WFCM.$s>                link to file "./K$n/evcm.dat" (optional)
                                       containing wavefunctions at preceding step
       </K-POINT.$n>
```

```
</EIGENVECTORS>
```

7.2 Restart files

8 Modifying/adding/extending QUANTUM ESPRESSO

8.1 Programming style (or lack of it)

Guidelines for developers:

- preprocessing options should be capitalized and start with two underscores. Examples: __AIX, __LINUX, ...
- fortran commands should be capitalized: CALL something( )
- variable names should be lowercase: `foo = bar/2`
- indent DO's and IF's with three white spaces (editors like emacs will do this automatically for you)
- do not write crammed code: leave spaces, insert empty separation lines
- comments (introduced by a !) should be used to explain what is not obvious from the code, not to repeat what is already evident. Obscure comments serve no purpose.
- do not use machine-dependent extensions or sloppy syntax. Standard f90 requires that a & is needed both at end of line AND at the beginning of continuation line if there is a character variable (inside ' ' or " ") spanning two lines. Some compilers do not complain if the latter & is missing, others do.
- use "dp" (defined in module "kinds") to define the type of real and complex variables
- all constants should be defined to be of kind "dp". Preferred syntax: 0.0_dp.
- use "generic" intrinsic functions: SIN, COS, etc.
- conversions should be explicitely indicated. For conversions to real, use DBLE, or else REAL(...,KIND=dp). For conversions to complex, use CMPLX(...,...,KIND=dp). For complex conjugate, use CONJG. For imaginary part, use AIMAG. IMPORTANT: Do not use REAL or CMPLX without KIND=dp, or else you will lose precision (except when you take the real part of a double precision complex number).
- Do not use automatic arrays (e.g. `REAL(dp) ::  A(N)` in a subroutine) except if you are sure that the array is small in all cases: you may easily exceed the stack size if the arrays are large.
- Do not use pointers unless you have a good reason to: pointers may hinder optimization, allocatable arrays should be used instead.
- If you use pointers, nullify them before performing tests on their status.

- Beware fancy constructs like structures and pointers: they look great on paper, but they have also an unlimited potential to make a code unreadable.
- Do not pass unallocated arrays as arguments, even in those cases where they are not actually used inside the subroutine.
- Do not use any construct that is susceptible to be flagged as out-of-bounds error, even if no actual out-of-bound error takes place.

8.2 Adding or modifying input variables

New input variables should be added to "Modules/input_parameters.f90", then copied to the code internal variables in the "input.f90" subroutine. The namelists and cards parsers are in "Modules/read_namelists.f90" and "Modules/read_cards.f90". Files "input_parameters.f90", "read_namelists.f90", "read_cards.f90" are shared by all codes, while each code has its own version of "input.f90" used to copy input values into internal variables

EXAMPLE: suppose you need to add a new input variable called "pippo" to the namelist control, then:

1. add pippo to the input_parameters.f90 file containing the namelist control

```
INTEGER :: pippo = 0
NAMELIST / control / ....., pippo
```

   Remember: always set an initial value!

2. add pippo to the control_default subroutine (contained in module read_namelists.f90)

```
 subroutine control_default( prog )
...
IF( prog == 'PW' ) pippo = 10
...
end subroutine
```

   This routine sets the default value for pippo (can be different in different codes)

3. add pippo to the control_bcast subroutine (contained in module read_namelists.f90)

```
  subroutine control_bcast( )
  ...
  call mp_bcast( pippo )
  ...
  end subroutine
```

9 Using SVN

The package is available read-only using anonymous access to the Subversion (SVN) repository. Developers can have read-write access when needed. Note that the latest (development) version may not work properly, and sometimes not even compile properly. Use at your own risk.

Subversion, also known as SVN, is a software that allows many developers to work and maintain a single copy of a software in a central location (repository). It is installed by default on many Unix machines, or otherwise it can be very easily installed. For the end user, SVN is rather similar to CVS: if no advanced features are used, the basic commands are the same. More information on SVN can be found here: `http://subversion.apache.org/`.

Follow the instructions in the SCM page in `qe-forge.org` for the QUANTUM ESPRESSO project: `http://qe-forge.org/scm/?group_id=10`, to check out (i.e. download) the SVN repository in either read-write or anonymous mode. The distribution will appear in directory `trunk/espresso/`. Branches (i.e. sub-versions) will appear as separate directories.

9.1 SVN operations

To update the code to the current version:

```
svn update
```

in the directory containing the distribution. To see the difference between the current version and your modified copy:

```
svn diff
```

To save your modified version into the repository: (read-write access only):

```
svn commit
```

If you also want to add a new file, before commiting give command

```
svn add
```

9.2 Removing conflicts

Look into the conflicting section: in most cases conflicts are trivial (format changes, white spaces) or easily solved (the part of the code you were modifying has been moved to another place, for instance). Sometimes, somebody else has done changes that are incompatible with yours during the same period. Use the web-svn interface:

```
http://qe-forge.org/scm/browser.php?group\_id=10
```

You can select the repository at a given date or with a given "tag": this may be useful if if you suspect that the incompatibile change happened after that date.

10 bibliography

Fortran books:

- M. Metcalf, J. Reid, Fortran 95/2003 Explained, Oxford University Press (2004)
- S. J. Chapman, Fortran 95/2003 for Scientists and Engineers, McGraw Hill (2007)
- J. C. Adams, W. S. Brainerd, R. A. Hendrickson, R. E. Maine, J. T. Martin, B. T. Smith, The Fortran 2003 Handbook, Springer (2009)
- W. S. Brainerd, Guide to Fortran 2003 Programming, Springer (2009)

On-line tutorials:

- Fortran: http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html
- Make: http://en.wikipedia.org/wiki/Make_(software)
- Configure script: http://en.wikipedia.org/wiki/Configure_script

(info courtesy of Goranka Bilalbegovic)