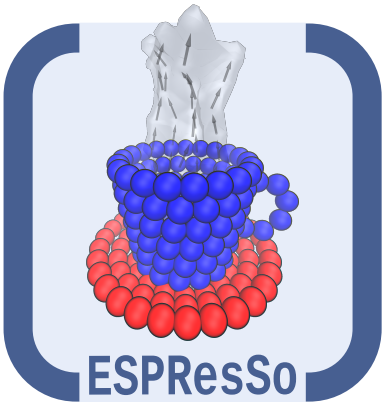

ESPResSo Developer's Guide

for version 3.4-dev-1009-g69a2b86-dirty

June 12, 2015

Contents

1 Getting into Contact

The first thing that you should do when you want to start to participate is to get into contact with the ESPResSo developers. To do that, subscribe to the developers' mailing list at `http://lists.nongnu.org/mailman/listinfo/espressomd-devel` and write a short email to the list address `espressomd-devel@nongnu.org` where you state your experience with simulations and programming and the things that you would like to do in ESPResSo. We do not bite, and we are happy about anybody who wants to participate!

2 Development Environment

2.1 Required Development Tools

If you want to participate in the development of ESPResSo, you will require the following tools depending on what tasks you want to perform:

- To be able to access the development version of ESPResSo, you will need the distributed versioning control system Git[1]. Section 2.3 on the following page contains documentation on how we employ git.
- To build ESPResSo from the development sources, you will need not too old versions of the "GNU autotools" (*i.e.* automake[2] and autoconf[3]) installed on your system. Section 2.2 contains information on how to build the development code, section 2.5 on the following page contains details on how the build system works.
- To be able to compile the User's Guide or the Developer's Guide, you will need a LaTeX-installation (all recent ones will do). For details, refer to chapter **??** on page ??.
- To compile the Doxygen code documentation, you will need to have the tool Doxygen[4].

All of these tools should be easy to install on most Unix operating systems.

2.2 Building the Development Code

- Use `bootstrap.sh` before building the code.
- Use configure with `--enable-maintainer-mode`.
- Possible targets of `make`
  - `check`
  - `dist`
  - `doc`

[1] `http://git-scm.com/`
[2] `http://www.gnu.org/software/automake/`
[3] `http://www.gnu.org/software/autoconf/autoconf.html`
[4] `http://www.doxygen.org/`

- `ug`
- `dg`
- `doxygen`
- `tutorials`
- `distcheck`
- `clean`, `distclean`

2.3 Git Repositories

- Development Repository: `https://github.com/espressomd/espresso`
- Code hosting at GNU Savannah `https://savannah.nongnu.org/projects/espressomd/`

2.4 Jenkins Build Server

`http://espressomd.org/jenkins`

- ESPResSo homepage `http://espressomd.org`
- Jenkins build server

2.5 Build System

The build system of ESPResSo makes use of the GNU autotools suite consisting of automake[5] and autoconf[6]. If you want to use the development source code of ESPResSo, you first need to install both of these tools.

The central source files of the build system are the following:

- `configure.ac`
- `config/*.m4`
- `Makefile.am`
- the different `Makefile.am` in the subdirectories

To change the behaviour of the build system, you have to modify any these files.

2.5.1 Running `bootstrap.sh`

The script `bootstrap.sh` in the top level source directory can be used to run `automake`, `autoconf` and the associated tools to generate the `configure` script and the `Makefile.in` used during compilation. Once `bootstrap.sh`, the ESPResSo development code can be build and used like the release code.

[5] `http://www.gnu.org/software/automake/`
[6] `http://www.gnu.org/software/autoconf/autoconf.html`

2.5.2 Creating a Distribution Package

As described in the User's Guide, to create a `.tar.gz` distribution file that contains all required files, you can simply run `make dist`. This will bundle all files and put them into an archive with the name `espresso-`*`version`*`.tar.gz`.

Even better, you can also run `make distcheck`. This will not only create the distribution file, but it will also thoroughly check the created distribution, *i.e.* it will try to

- unpack the distro into a new directory
- configure and build it (`configure`)
- run the testsuite (`make check`)
- install it into a new directory (`make install`)
- uninstall it (`make uninstall`)

Whenever something goes wrong in these checks, it will give an error message that describes the problem. When everything goes fine, you can be relatively sure that you have a useful ESPResSo distribution package.

In some cases, it might be necessary to pass some options to the run of `configure` done by `make distcheck¿`. To these ends, the environment variable `DISTCHECK_CONFIGURE_FLAGS` can be set to the required options.

Example

```
DISTCHECK_CONFIGURE_FLAGS="--without-mpi CPPFLAGS=\"-I /usr/include/tcl8.4\"" \
  make distcheck
```

2.5.3 Adding New Files

To add new files to ESPResSo (like C source files or header files) you need to do the following:

- Add the files to the `Makefile.am` in the same directory
- Run `bootstrap.sh` in the source directory
- Check the distribution by using `make distcheck`
- Add the files to the Git repository

2.6 Testsuite

- How to write tests?
- How they are called (`runtest.sh`)

3 Documentation

3.1 User's Guide

The User's Guide is written in LaTeX. The source files reside in the subdirectory `doc/ug/` of the ESPResSo sources. The master file is `ug.tex`, each chapter is contained in its own source file.

3.1.1 General issues

- Other than usual, in the UG's `.tex`-files, the underscore character "_" is a normal character in most cases, *i.e.* it can be used unquoted. Unfortunately, however, this makes it impossible to use "_" in LaTeX-labels (don't ask me why!).
- Headings should start with a capital letter and continue with lower-case letters ("First steps" and *not* "First Steps").
- Use the "-ing" form in headings, *i.e.* "Setting up particles" instead of "Particle setup" or the like.
- To see which parts of the User's guide need to be fixed or which documentation is missing, there is a `\todo`-command where one can put notes about what remains to be done. In the release version, the boxes are disabled so that they do not disturb the users. They can be turned on by commenting out the appropriate line in `ug.tex`:

```
%% For building the distribution docs, disable todo boxes.
%\usepackage[disable]{todonotes}
\usepackage{todonotes}
```

3.1.2 Building the User's Guide

- To build the User's Guide, you need to have a LaTeX-installation that includes BibTeX, PDFLaTeX and makeindex. All installations that I know of provide these tools.
- There are two methods to build the User's Guide:
  - Use `make ug` from the build directory to build it. This will automatically generate the ESPResSo quick reference and call latex, bibtex and makeindex as required.

– Use `perl latexmk` from the source directory. This will basically do the same, however, it will *not* automatically update the quick reference. The advantage of this method is, that you can use all of `latexmk`'s nice features, such as `-pvc`. You can always rebuild the quick reference manually by calling

```
awk -f assemble_quickref.awk > quickref.inp
```

3.1.3 Adding New Files

To add new LaTeX-files to the User's Guide, you need to modify

- `ug.tex`: add an appropriate include command near the end of the file
- `Makefile.am`: add the file to the variable `ug_TEXFILES`.

3.1.4 Additional Environments and Commands

To maintain a consistent layout, a number of environments and commands have been defined that should be used where applicable.

- For the description of ESPResSo's Tcl-commands, read 3.1.5.
- The name of ESPResSo should be set via the command `\es`.
- The strings "*i.e.*", "*e.g.*" and "*et al.*" should be set via `\ie`, `\eg` and `\etal`.
- For short pieces of code that can be displayed inline, use `\codebox{`*text*`}` or `\verb!`*text*`!`
- For longer code pieces or the syntax decription of non-Tcl commands, the environment `code` exists:

  ```
  \begin{code}
    ...
  \end{code}
  ```

  Note that this is *not* a verbatim environment, *i.e.* it will evaluate LaTeX-commands that are used inside. Therefore, the characters `\`, `{` and `}` need to be quoted with backslashes inside the environment! Also, the underscore character "_" needs to be quoted like `\_`. On the other hand, it is possible to use other layout commands (like `\textit`) inside.

- For pieces of Tcl-code that make extensive use of `{` and `}`, a verbatim environment `tclcode` exists:

  ```
  \begin{tclcode}
   ...
  \end{tclcode}
  ```

3.1.5 Documentation of Tcl commands

Formal Syntax Definition

All ESPResSo-commands have to be documented in the User's Guide.

The command `\newescommand[`*label*`]{`*command*`}` should be used at the beginning of a command description to generate a label `es:`*command* and create appropriate index entries. The optional argument *label* is only required, when *command* contains an underscore character "_". In that case, a label `es:`*label* is generated that should not contain an underscore character.

For the *formal syntax definition*, you have to use the environments `essyntax` or `essyntax*`. Both will generate the headings *Syntax* and *Description*. `essyntax` will furthermore copy the syntax definition to the quick reference guide. For an example, look at the documentation of the `part` command in the file `part.tex`. Inside the `essyntax` environment, you have to use the following commands for typesetting the definition:

- `\variant{`*number*`}` to typeset the label of a command variant
- `\var{`*name*`}` to typeset a variable argument. Note, that the argument is typeset in math mode. This means, that you can use "_" to denote a subscript.
- `\keyword{`*text*`}` or `\lit{`*text*`}` to typeset keywords or literals.
- `\opt{`*text*`}` to typeset optional arguments. Note that usually, the text inside the `opt` command will not be wrapped. If the optional argument is pretty long and needs to be wrapped, use `\optlong`.
- `\optlong{`*text*`}` to typeset long optional argument blocks.
- `\alt{`*alt1* `\asep` *alt2* `\asep` *alt3* `...}` to typeset alternatives.
- `\feature{`*feature*`}` to typeset when a *feature* is referred to.
- `\require{`*number*`}{`*text*`}` to typeset *text* to show that it requires certain features of ESPResSo. *number* denotes the marking that is shown next to *text*. When this command is used, you also have to use the `features`-environment (see below) at the end of the `essyntax` environment, where all of the *number*s used are explained.
- The environment `features` to typeset which features are required by the Tcl-command. Inside the environment, each feature should be declared via the command `\required[`*number*`]{`*feature*`}`, where the optional argument *number* is the number used above and *feature* is the feature in capital letters.

The formal syntax definition should be as simple and as readable as possible, as it will be what a user references to. Avoid very long definitions and constructs like nested alternatives and options. In those cases, prefer to split the syntax definition into several variants instead of writing it in a single, complicated definition.

Example

```
\begin{essyntax}
[clip]
\variant{5} constraint pore
  center \var{c_x} \var{c_y} \var{c_z}
  axis \var{n_x} \var{n_y} \var{n_z}
  radius \var{rad}
  length \var{length}
  type \var{id}

\require{1}{%
  \variant{6} constraint rod center \var{c_x} \var{c_y}
  lambda \var{lambda}
}

\require{1}{%
  \variant{7} constraint plate height \var{h}
  sigma \var{sigma}
}

\require{2,3}{%
  \variant{8} constraint ext_magn_field \var{f_x} \var{f_y} \var{f_z}
}

  \begin{features}
  \required{CONSTRAINTS}
  \required[1]{ELECTROSTATICS}
  \required[2]{ROTATION}
  \required[3]{DIPOLES}
  \end{features}
\end{essyntax}
```

Description

In the description, you should use all of the above typesetting commands when you refer to them in the text. In particular, every variable argument introduced via the `\var`-command in the definition has to be explained in detail:

- state explicitly the *type* of the argument (integer, float, string, Tcl-list)
- explain the meaning of the argument

If the command has a number of different options, *i.e.* independent, optional arguments, they can be described in the *arguments* environment:

```
\begin{arguments}
  \item[<arg1>] <description of arg1>
  \item[<arg2>] <description of arg2>
  ...
\end{arguments}
```

The environment will generate the subheading *Arguments* and nicely format the descriptions.

Example

```
\begin{arguments}
  \item[\opt{\alt{short \asep verbose}}] Specify, whether the output is
    in a human-readable, but somewhat longer format (\keyword{verbose}),
    or in a more compact form (\keyword{short}). The default is
    \keyword{verbose}.

  \item[\opt{\alt{folded \asep absolute}}] Specify whether the particle
    positions are written in absolute coordinates (\keyword{absolute})
    or folded into the central image of a periodic system
    (\keyword{folded}). The default is \keyword{absolute}.

  ...
\end{arguments}
```

3.2 Doxygen Code Documentation

The documentation of each function should contain a short description, if necessary a more detailed description and a description for the return value and parameters.

Look at the documentation of existing files and functions to get a feeling how it should be!

Doxygen is able to understand simple LaTeX and HTML commands as well as some special command in order to give the documentation a nice structure and to make it more readable. In the following list you find a short description of the most common commands we need:

- `\anchor` *name description*
  Create an anchor to which you can refer using the `\ref` command.

- `\ref` *name* `["`*text*`"]`
  Insert a link to another object in the documentation (*e.g.* an anchor).

- `<a href="http://www.your_url.html">title</a>`
  Link to an external HTML source.

- `\file` *name description*
  Special anchor for a file.

- `\image html` *image*
  Include a picture. The picture file should reside in the subdir `doc/doxygen/figs`. Do not use the HTML `<img>`-tag to include pictures, as doxygen will not copy the pictures into the documentation.

- `<ul> <li>List entry 1</li> <li>List entry 2</li></ul>`
  Creates a list in the documentation.

- `\param` *name description*
  Document the parameter of a function.

- `\return` *decription*
  Document the return value of a function.

4 Programmer's Guide

This chapter provides some hints on how to extend ESPResSo. It is not exhaustive, so for major changes the best documentation are the other developers.

4.1 Adding Global Variables

Global variables are the simplest way to communicate values between the Tcl script and the C simulation code. To make a C variable available to Tcl, declare the variable `extern` in a header file and include in `global.c`. Then add a new line to the definition of the constant data structure `fields` at the beginning of the file `global.c`. For details on the entries, see the definition of `Datafield` in `global.h`). Basically you have to declare *where* the variable is stored, *which type* (INT or DOUBLE) it has and *how many* elements. A callback procedure can be provided which checks if the given value is valid and stores it. It is also responsible for dispatching the new value to the other compute nodes, if necessary. The easiest way to do that is by using `mpi_bcast_parameter`, which will transfer the value to the other nodes. A simple example is `box_l` with the callback procedure `boxl_callback`. For `mpi_bcast_parameter` to work, it is necessary that they occur in the list of constant definitions at the beginning of `global.h`. So please keep this list in sync!

4.2 Adding New Bonded Interactions

Every interaction resides in its own source file. A simple example for a bonded interaction is the FENE bond in `fene.h`. The data structures, however, reside in `interaction_data.h`. The bonded interactions are all stored in a union, `Bonded_ia_parameters`. For a new interaction, just add another struct. Each bonded interaction is assigned a type number, which has the form `BONDED_IA_*`, *e.g.* `BONDED_IA_FENE`. The new interaction also has to have such a *unique* number.

After the setup of the necessary data structures in `interaction_data.h`, write the source file, something like `new_interaction.h`. You may want to use `fene.h` as a template file. Typically, you will have to define the following procedures:

- `int *_set_params(int bond_type, ...)`

  This function is used to define the parameters of a bonded interaction. `bond_type` is the bond type number from the inter command, and not one of the `BONDED_*`. It is rather an index to the `bonded_ia_params` array. `make_bond_type_exist` makes sure that all bond types up the given type exist and are preinitialized with

`BONDED_IA_NONE`, *i.e.* are empty bond types. Therefore fill `bonded_ia_params[bond_type]` with the parameters for your interaction type.

- ```
  int calc_*_force(Particle *p1, Particle *p2,...,
                   Bonded_ia_parameters *iaparams,
                   double dx[3], double force[3], ...)
  ```

  This routine calculate the force between the particles. `ia_params` represents the parameters to use for this bond, `dx` represents the vector pointing from particle 2 to particle 1. The force on particle 1 is placed in the force vector (and *not* added to it). The force on particle 2 is obtained from Newton's law. For many body interactions, just add more particles in the beginning, and return the forces on particles 1 to N-1. Again the force on particle N is obtained from Newton's law. The procedure should return 0 except when the bond is broken, in which case 1 is returned.

- ```
  int *_energy(Particle *p1, Particle *p2, ...,
               Bonded_ia_parameters *iaparams,
               double dx[3], double *_energy)
  ```

  This calculates the energy originating from this bond. The result is placed in the location `_energy` points to, `ia_params` and `dx` are the same as for the force calculation, and the return value is also the flag for a broken bond.

After the preparation of the header file, the bonded interaction has to be linked with the rest of the code. In `interaction_data.c`, most of the work has to be done:

1. Add a name for the interaction to `get_name_of_bonded_ia`.
2. In `calc_maximal_cutoff`, add a case for the new interaction which makes sure that `max_cut` is larger than the interaction range of the new interaction, typically the bond length. This value is always used as calculated by `calc_maximal_cutoff`, therefore it is not strictly necessary that the maximal interaction range is stored explicitly.
3. Add a print block for the new interaction to `tclcommand_inter_print_bonded`. The print format should be such that the output can be used as input to inter, and defines the same bond type.
4. In `tclcommand_inter_parse_bonded`, add a parser for the parameters. See the section on parsing below.
5. Besides this, you have enter the force respectively the energy calculation routines in `add_bonded_force`, `add_bonded_energy`, `add_bonded_virials` and `pressure_calc`. The pressure occurs twice, once for the parallelized isotropic pressure and once for the tensorial pressure calculation. For pair forces, the pressure is calculated using the virials, for many body interactions currently no pressure is calculated.

After the new bonded interaction works properly, it would be a good idea to add a testcase to the testsuite, so that changes breaking your interaction can be detected early.

4.3 Adding New Nonbonded Interactions

Writing nonbonded interactions is similar to writing nonbonded interactions. Again we start with `interaction_data.h`, where the parameter structure has to be set up. Just add your parameters *with reasonable names* to `IA_parameters`. Note that there must be a setting for the parameters which disables the interaction.

Now write the header file for the interaction. This time `ljcos.h` may be a good example. The needed routines are

- `int print*IAToResult(Tcl_Interp *interp, int i, int j)`

  writes out the interaction parameters between particles of type `i` and `j` to the interpreters result such that the result can be fed into the `inter` command again to obtain the same interaction. The `IA_parameters` pointer can be obtained conveniently via `get_ia_param(i,j)`.

- ```
  int *_parser(Tcl_Interp * interp, int part_type_a, int part_type_b,
               int argc, char ** argv)
  ```

  parses the command line given by `argc` and `argv` for the parameters needed for the interaction, and writes them to the `IA_parameters` for types `part_type_a` and `part_type_b`. For details on writing the parser, see below. The routine returns 0 on errors and otherwise the number of parameters that were read from the command line.

- ```
  void add_*_pair_force(Particle *p1, Particle *p2,
                        IA_parameters *ia_params,
                        double d[3], double dist2, double dist,
                        double force[3])
  double *_pair_energy(Particle *p1, Particle *p2,
                       IA_parameters *ia_params,
                       double d[3], double dist2, double dist)
  ```

  are the routines to compute the force respectively the energy. `ia_params` gives the interaction parameters for the particle types of particles `p1` and `p2`, `d` gives the vector from particle 2 to particle 1, `dist` its length and `dist2` its squared length. The last three parameters can be chosen on demand. Note that unlike in the bonded case, the force routine is called `add_*`, *i.e.* the force has to be *added* to force. The `*_pair_energy` routine simply returns the energy directly instead of the pointer approach of the bonded interactions.

Change `interaction_data.c` as follows (most changes are pretty much the same for all potentials):

1. modify `initialize_ia_params` and `copy_ia_params` to take care of the additional parameters needed for your potential.

2. `checkIfParticlesInteract` has to be modified to also check for the no interaction condition for the new interaction (typically zero cutoff).

3. `calc_maximal_cutoff` has to modified such that `max_cut` is larger than the maximal cutoff your interaction needs. Again, the code always uses the result from this function, therefore the cutoff does not have to be stored explicitly in the interaction parameters.

4. add your `print*IAToResult` routine to `tclprint_to_result_NonbondedIA`.

5. add the `*_parser` routine to `tclcommand_inter_parse_bonded`.

After this, add the force calculation to `add_non_bonded_pair_force`, `add_non_bonded_pair_virials` and `pressure_calc`, and the energy calculation to `add_non_bonded_pair_energy`.

After the new non-bonded interaction works properly, it would be a good idea to add a testcase to the testsuite, so that changes breaking your interaction can be detected early.

4.4 Tcl I/O - Parsing and Printing

- `ARG_0_IS`
- `Tcl_GetDouble/Int ...`
- `Tcl_PrintDouble/Int` (take care of number of arguments)
- `TCL_INTEGER_SPACE` ...

4.5 Particle Data Organization

The particle data organization is described in the Tcl command cellsystem, its implementation is briefly described in `cells.h` and `ghosts.h`. Here only some details on how to access the data is assembled. Writing a new cellsystem almost always requires deep interactions with the most low level parts of the code and cannot be explained in detail here.

Typically, one has to access all real particles stored on this node, or all ghosts. This is done via a loop similar to the following:

```
Cell *cell;
int c,i,np,cnt=0;
Particle *part;

for (c = 0; c < local_cells.n; c++) {
  cell = local_cells.cell[c];
  part = cell->part;
  np   = cell->n;
  for(i=0 ; i < np; i++) {
     do_something_with_particle(part[i]);
  }
```

```
}
```

To access the ghosts instead of the real particles, use `ghost_cells` instead of `local_cells`.

Another way to access particle data is via `local_particles`. This array has as index the particle identity, so that `local_particles[25]` will give you an pointer to the particle with identity 25, or `NULL`, if the particle is not stored on this node, neither as ghost nor as real particle. Note that the `local_particle` array does not discriminate between ghosts and real particles. Its primary use is for the calculation of the bonded interactions, where it is used to efficiently determine the addresses of the bonding partner(s).

The master node can add and remove particles via `place_particle` and `remove_particle`, or change properties via `set_particle_v` etc. This is the preferred way to handle particles, since it is multiprocessor save.

However, some algorithms, especially new cellsystems, may force you to operate locally on the particle data and shift them around manually. Since the particle organization is pretty complex, there are additional routines to move around particles between particle lists. The routines exist in two versions, one indexed, and one unindexed. The indexed version take care of the `local_particles` array, which for each particle index tells where to find the particle on this node (or `NULL` if the particle is not stored on this node), while the unindexed versions require you to take care of that yourself (for example by calling `update_local_particles`). The second way is much faster if you do a lot of particle shifting. To move particles locally from one cell to another, use `move_indexed_particle` or `move_unindexed_particle`, never try to change something directly in the lists, you will create a mess! Inserting particles locally is done via `append_indexed_particle` or `append_unindexed_particle`.

Besides the `local_particles array`, which has to be up to date at any time, there is a second array `particle_node`, which is available on the master node only outside of the integrator, *i.e.* in the Tcl script evaluation phases. If `particle_node` is `NULL`, you have to call `build_particle_node` to rebuild it. For each particle identity it contains the node that the particle is currently located on.

The proper cell for a particle is obtained via `CellStructure::position_to_node`, which calculates for a given position the node it belongs to, and `CellStructure::position_to_cell`, which calculates the cell it belongs to on this node, or `NULL`, if the cell is from a different node. However, you should normally not be bothered with this information, as long as you stick to `place_particle` and the other routines to modify particle data.

Writing a new cellsystem basically requires only to create the functions listed in `CellStructure`. The `init` function has to also setup the communicators, which is the most complex part of writing a new cellsystem and contains all the communication details. `prepare_comm` is a small wrapper for the most common operations. Otherwise just grep for `CELL_STRUCTURE_DOMDEC`, and add some appropriate code for your cell system. Note, however, that each cell system has its specific part of the code, where only this cellsystem does something strange and unique, so here you are completely on your own. Good luck.

4.6 Errorhandling for Developers

Developers should use the errorhandling mechanism whenever it is possible to recover from an error such that continuing the simulation is possible once the source of the error is removed, i. e. the bond is removed or a parameter changed. For example, if due to excessive forces, particles have been far out of their current node, ESPResSo puts them into one of the local cells. Since the position is unphysical anyways, it is of no importance anymore, but now the user can place the particles anew and perhaps decrease the time step such that the simulation can continue without error. However, most often the recovery requires no special action.

To issue a background error, call

```
errtxt=runtime_error(length)
```

where length should be the maximal length of the error message (you can use `TCL_DOUBLE_SPACE` rsp. `TCL_INTEGER_SPACE` to obtain space for a double rsp. integer). The function returns a pointer to the current end of the string in `error_msg`. After doing so, you should use the `ERROR_SPRINTF`-macro, which substitutes to a simple `sprintf`, so that your errormessage will automatically be added to the "runtime-errors resolved"-page. Please make sure that you give each of your errors an unique 3-digit errorcode (for already used errorcodes have a look at the "runtime-errors resolved"-page), have the curled braces around your message and the space at the end, otherwise the final error message will look awful and will propably not automatically be added to our error-page. Typically, this looks like this:

```
if (some_error_code != OK) {
  char *errtxt = runtime_error(TCL_INTEGER_SPACE + 128);
  ERROR_SPRINTF(errtxt, "{error occured %d} ", some_error_code);
  recovery;
}
```

If you have long loops during which runtime errors can occur, such as the integrator loop, you should call `check_runtime_errors` from time to time and exit the loop on errors. Note that this function requires all nodes to call it synchronously.

In all cases, all Tcl commands should call `mpi_gather_runtime_errors` before exiting. You simply handover the result you were just about to return. If the result was `TCL_ERROR`, then `mpi_gather_runtime_errors` will keep the Tcl error message and eventually append the background errors. If the result was `TCL_OK`, *i.e.* your function did not find an error, the result will be reset (since ESPResSo is in an undefined state, the result is meaningless), and only the background errors are returned. Whenever a Tcl command returns, instead of `return TCL_OK/TCL_ERROR` you should use

```
return mpi_gather_runtime_errors(interp, TCL_OK/TCL_ERROR);
```