Contents

Part I.

Users

1. Overview of the build system

1.1. Main objectives

The build system of ABINIT is here to fulfill the following objectives:

- take care of the makefiles;

1.2. Underlying concepts

Build directory support
Config-file support

2. The *configure* script

2.1. Running configure

Autoconf is a tool producing a shell script that automatically configures software source code to adapt to many kinds of environments. **The configuration script produced by Autoconf is independent of Autoconf when it is run, so that its users do not need to install Autoconf**. In other words: you do not need have Autoconf installed to build ABINIT. Moreover this configuration script requires no manual user intervention when run; it do not normally even need an argument specifying the system type. Instead, it individually tests for the presence of each feature that the software package it is tuned to might need. However it does not yet have paranormal powers, and in particular has no access to what you have in mind. You still have to explicitely interact with it for now, and the best way to do it is through the numerous options of this *configure* script.

One highly-recommended step is to create a build directory and go there before running configure:

```
mkdir tmp-build && cd tmp-build
```

In the following, we will assume that you have done so.

If you run it without arguments, the *configure* script will do its best to detect the components of your computer and development environment automatically. It is however possible to tune its behavior through the use of two classes of parameters:

- command-line options, composed of triggers (*enable/disable*) and specifiers (*with/without*), plus a few special options;
- environment variables, which influence the overall behaviour of the script.

A typical call looks like:

```
    ../configure [OPTION] ... [VAR=VALUE] ...
```

Here is what [OPTION] stands for:

Type ...	**if you want to ...**

`--enable-FEATURE[=ARG]`
`--disable-FEATURE`
	activate FEATURE [ARG=yes]
	do not activate FEATURE (same as --enable-FEATURE=no)

`--with-PACKAGE[=ARG]`
`--without-PACKAGE`
	use PACKAGE [ARG=yes]
	do not use PACKAGE (same as --with-PACKAGE=no)

To assign environment variables (e.g., CPP, FC, ...), you specify them as `VAR=VALUE` couples on the command line. Please note that there must be no spaces around the '=' sign. Moreover, `VALUE` must be quoted when it contains spaces. If some assignements are ignored by the configure script, just try the other way around:

```
VAR=VALUE ... ./configure [OPTION] ...
```

In this chapter, the defaults for the options are specified in square brackets. No brackets means that there is no default value.

2.2. Environment variables

In table 2.1, you will find short descriptions of the most useful variables recognized by the configure script of ABINIT. Use these variables to override the choices made by `configure` or to help it to find libraries and programs with nonstandard names/locations. Please note that they always have precedence over command-line options.

There are 2 environment variables of critical importance to the build system, though they cannot be managed by *configure*:

- *PATH*, which defines the order in which the compilers will be found, and the number of hits;
- *LD_LIBRARY_PATH*, which will greatly help the build system find usable external libraries, in particular MPI.

Improper settings of these 2 variables may cause a great confusion to the configure script in some cases, in particular when looking for MPI compilers and libraries. A typical issue encountered is the following crash:

```
checking for gcc... /home/pouillon/hpc/openmpi-1.2.4-gcc4.1/bin/mpicc
checking for C compiler default output file name... a.out
checking whether the C compiler works... configure: error: cannot run C compiled programs.
If you meant to cross compile, use `--host'.
See `config.log' for more details.
```

And a look at config.log shows:

```
...
configure:6613: checking whether the C compiler works
configure:6623: ./a.out ./a.out: error while loading shared libraries:
libmpi.so.0: cannot open shared object file: No such file or directory
configure:6626: $? = 127
configure:6635: error: cannot run C compiled programs.
...
```

Option	**Description**
AR	Archiver
ARFLAGS	Archiver flags
CPP	C preprocessor
CPPFLAGS	C/C++ preprocessor flags, e.g. `-I<include_dir>` if you have headers in a non-standard directory named *<include_dir>*
CC	C compiler command
CFLAGS	C compiler flags
CC_LDFLAGS	C link flags to prepend to the command line
CC_LIBS	Libraries to append when linking a C program
CXX	C++ compiler command
CXXFLAGS	C++ compiler flags
CXX_LDFLAGS	C++ link flags to prepend to the command line
CXX_LIBS	Libraries to append when linking a C++ program
FC	Fortran compiler command
FCFLAGS	Fortran compiler flags
FC_LDFLAGS	Fortran link flags to prepend to the command line
FC_LIBS	Libraries to append when linking a Fortran program

Table 2.1.: Influencial environment variables for the build system of ABINIT.

This kind of error results from a missing path in the *LD_LIBRARY_PATH* environment variable, and can be solved very easily, in the present case this way:

```
export LD_LIBRARY_PATH="/home/pouillon/hpc/openmpi-1.2.4-gcc4.1/lib:${LD_LIBRARY_PATH}"
```

in the case of a BASH shell, and by:

```
setenv LD_LIBRARY_PATH "/home/pouillon/hpc/openmpi-1.2.4-gcc4.1/lib:${LD_LIBRARY_PATH}"
```

for a C shell.

2.3. The configuration process

Configuring ABINIT is a delicate step-by-step process, because each component is interacting permanently with most others. This is reflected in the output of *configure*, that we describe in this section.

The process starts with an overall startup, where the basic parameters required by Autoconf and Automake are set. During the second part of this step, the build system of ABINIT reads the options from the command line and from a config file, making sure that the environment variables will always have precedence over the command-line options, which in turn override the options read from the config file. It then reports about changes in the user interface of the build system, warning the user if they have used an obsolete option.

The next step is about ensuring the overall consistency of the options provided to configure. The build system takes the necessary decisions so that the code may be built safely. It then parses the options, and issues an error if the user has provided invalid options. The error messages give all the information needed to fix the problems.

Then comes the MPI startup stage, which the first half of the configuration of MPI support. This must happen \underline{before} any Autoconf compiler test, in order to give the build system the possibility to consistently select the MPI compilers that have been detected. This step is mandatory to avoid configuration issues later on, due to mismatches between the sequential and parallel compilers.

The next step is to find the various utilities that the build system may need along the rest of the configuration process. This runs usually very smoothly, since these tools are found on most of the platforms ABINIT runs on.

The preprocessing step is where serious things really start. The C preprocessor is searched for, which involves in turn the search for a working C compiler. At this point, all compilers must already have been selected. This is also typically where *configure* may crash if the MPI installation detected is broken or misconfigured (see "Environment variables" section within this chapter), because the C compiler will not be able to produce executables. This is why MPI support is disabled by default, and we are open to any suggestion.

The three next steps involve the search for suitable C, C++ and Fortran compilers, the detection of their type, and the application of tricks to have them work properly on the user's platform. These are also stages where the configuration may fail, in particular if no suitable Fortran compiler is found.

Then the build system configures the use of the archiver, to build the numerous libraries that are part of ABINIT.

The two next steps are about fine-tuning the compile flags so that the build will work fine if the architecture is 64-bit (work still in progress), and to set the adequate level of optimization according to the platform parameters identified so far.

Here comes the probably most critical step of the configuration: MPI support. If everything could be set during the MPI startup stage, no further test is performed, and the parallel code is marked for building. If not, the build system will try to detect whether the compilers are able to build MPI source code and set the MPI options accordingly.

Once all this is done, the build system can set the parameters for the linear algebra and FFT libraries (work still in progress), before turning to the connectors (see Fig. 2.1).

One last configuration step is dedicated to the nightly build support, which is now

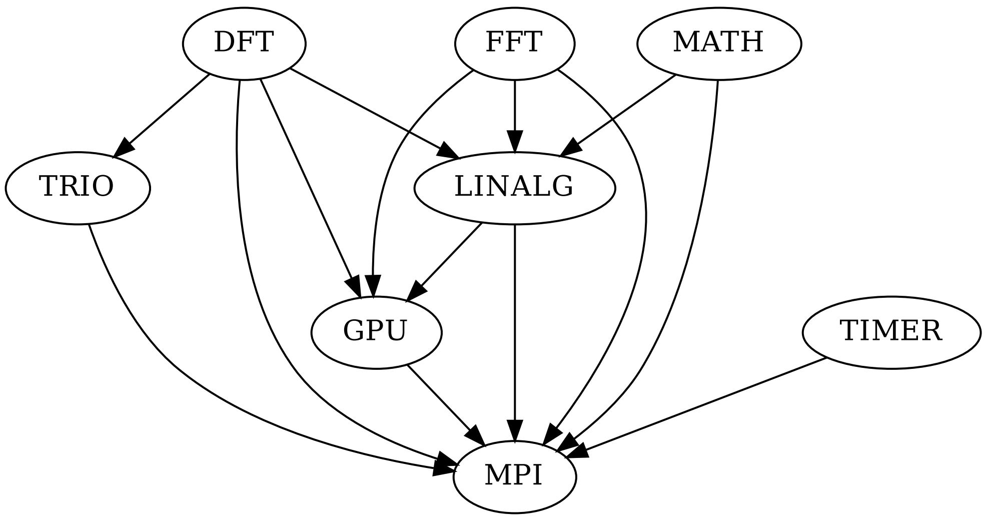

Figure 2.1.: Dependency diagram of the connectors.

working but still at an early stage of development.

The very last step is to output the configuration to the numerous makefiles, as well as to a few other important files. At the end, a warning is issued if the Fortran compiler in use is known to cause trouble.

2.4. Options provided by Autoconf

Every *configure* script generated by Autoconf provides a basic set of options, whatever the package and the environment. They either give information on the tunable parameters of the package or influence globally the build process. In most cases you will only need a few of them, if any.

Overall configuration:

Option		Description
-h,	--help	display all options and exit
	--help=short	display options specific to the ABINIT package
	--help=recursive	display the short help of all the included packages
-V,	--version	display version information and exit
-q,	--quiet, --silent	do not print 'checking...' messages
	--cache-file=FILE	cache test results in FILE [disabled]
-C,	--config-cache	alias for '--cache-file=config.cache'
-n,	--no-create	do not create output files
	--srcdir=DIR	find the sources in DIR [configure dir or '..']

Installation directories:

Option	Description
`--prefix=PREFIX`	install architecture-independent files in PREFIX [/opt]
`--exec-prefix=EPREFIX`	install architecture-dependent files in EPREFIX [PREFIX]

By default, `make install` will install all the files in subdirectories of */opt/abinit/<version>*. You can specify an installation prefix other than */opt* using `--prefix`, for instance `--prefix=$HOME`.

For a finer-grained control, use the options below.

Fine tuning of the installation directories:

Option	Description
`--bindir=DIR`	user executables [EPREFIX/bin]
`--sbindir=DIR`	system admin executables [EPREFIX/sbin]
`--libexecdir=DIR`	program executables [EPREFIX/libexec]
`--datadir=DIR`	read-only architecture-independent data [PREFIX/share]
`--sysconfdir=DIR`	read-only single-machine data [PREFIX/etc]
`--sharedstatedir=DIR`	modifiable architecture-independent data [PREFIX/com]
`--localstatedir=DIR`	modifiable single-machine data [PREFIX/var]
`--libdir=DIR`	object code libraries [EPREFIX/lib]
`--includedir=DIR`	C header files [PREFIX/include]
`--oldincludedir=DIR`	C header files for non-gcc [/usr/include]
`--infodir=DIR`	info documentation [PREFIX/info]
`--mandir=DIR`	man documentation [PREFIX/man]

Program names:

Option	**Description**
`--program-prefix=PREFIX`	prepend PREFIX to installed program names
`--program-suffix=SUFFIX`	append SUFFIX to installed program names
`--program-transform-name=PROGRAM`	run sed PROGRAM on installed program names

System types:

Option	**Description**
`--build=BUILD`	configure for building on BUILD [guessed]
`--host=HOST`	cross-compile to build programs to run on HOST [BUILD]
`--target=TARGET`	configure for building compilers for TARGET [HOST]

Developer options:

Option	**Description**
`--enable-shared[=PKGS]`	build shared libraries [default=no]
`--enable-dependency-tracking`	speeds up one-time build
`--enable-dependency-tracking`	do not reject slow dependency extractors
`--with-gnu-ld`	assume the C compiler uses GNU ld [default=no]

The following sections describe Abinit-specific options.

2.5. Compiler options

2.5.1. Debugging

The build system of Abinit provides a comprehensive database of debugging flags, covering all supported compiler vendors, compiler versions and architectures. They can be accessed when using the *profile* debugging mode of the build system, which is enabled when the *--enable-debug* option is neither set to “yes” nor “no”. The profile mode offers 4 different optimization levels, controlled by the value of the *--enable-debug* option:

- *basic*;
- *enhanced*;
- *paranoid*;
- *naughty*.

The default level is *basic*, which corresponds to the setting of minimal compiler-dependent debugging flags, typically "*-g*". The *enhanced* level add checks for the most common bad programming practices, while the *paranoid* level covers a much broader range of mistakes. The *naughty* level should be used by developers only, since it tries to make the build fail. It is important to note that these debugging levels are strictly ordered and cumulative. This means e.g. that setting *--enable-debug=paranoid* will also apply the flags from the *basic* and *enhanced* levels.

When *--enable-debug=yes*, the build system expects debugging parameters from the command line. Compile flags may be specified through the *CFLAGS_DEBUG* (C programs), *CXXFLAGS_DEBUG* (C++ programs) and *FCFLAGS_DEBUG* (Fortran programs) environment variables. Link flags may be provided through *CC_LDFLAGS_DEBUG* (C programs), *CXX_LDFLAGS_DEBUG* (C++ programs) and *FC_LDFLAGS_DEBUG* (Fortran programs). Linking additional libraries should be done through *CC_LIBS_DEBUG* (C programs), *CXX_LIBS_DEBUG* (C++ programs) and *FC_LIBS_DEBUG* (Fortran programs).

When *--enable-debug=no*, the debugging features of the build system are completely silenced and the above environment variables are ignored.

2.5.2. Optimization

The build system of Abinit provides a comprehensive database of optimization flags, covering all supported compiler vendors, compiler versions and architectures. They can be accessed when using the *profile* optimization mode of the build system, which is enabled when the *--enable-optim* option is neither set to "yes" nor "no". The profile mode offers 3 different optimization levels, controlled by the value of the *--enable-optim* option:

- *safe*;
- *standard*;
- *aggressive.*

These names are self-explanatory, and the default is of course *standard*, which corresponds to the optimization database present in version 4 of Abinit, extended and updated. The *safe* level may be quite convenient when the build fails at the standard level, due to bugs in the compilers. It is obvious that the *aggressive* level should be used with extreme care, and systematically accompanied by a run of the whole test suite before any production calculation.

When *--enable-optim=yes*, the build system expects optimization parameters from the command line. Compile flags may be specified through the *CFLAGS_OPTIM* (C programs), *CXXFLAGS_OPTIM* (C++ programs) and *FCFLAGS_OPTIM* (Fortran programs) environment variables. Link flags may be provided through *CC_LDFLAGS_OPTIM*

(C programs), *CXX_LDFLAGS_OPTIM* (C++ programs) and *FC_LDFLAGS_OPTIM* (Fortran programs). Linking additional libraries should be done through *CC_LIBS_OPTIM* (C programs), *CXX_LIBS_OPTIM* (C++ programs) and *FC_LIBS_OPTIM* (Fortran programs).

When *--enable-optim=no*, the optimization features of the build system are completely silenced and the above environment variables are ignored. Please note that debugging has precedence over optimization, and that the latter is disabled for all debugging levels above *basic*.

2.5.3. Hints

Abinit is a complex project, embedding the source code of other projects as well. In order to be successful, the builds require *hints* in addition to the other flags. The build system of Abinit thus provides a comprehensive database of optimization flags, covering all supported compiler vendors, compiler versions and architectures. This is the purpose of the *--enable-hints* option, which defaults to "yes". It is extremely recommend to keep it so.

However, there are cases where even the build system is not able to provide sufficient information for a successful build. It can thus be complemented by extra information from the command line. Compile flags may be specified through the *CFLAGS_EXTRA* (C programs), *CXXFLAGS_EXTRA* (C++ programs) and *FCFLAGS_EXTRA* (Fortran programs) environment variables. Link flags may be provided through *CC_LDFLAGS_EXTRA* (C programs), *CXX_LDFLAGS_EXTRA* (C++ programs) and *FC_LDFLAGS_EXTRA* (Fortran programs). Linking additional libraries should be done through *CC_LIBS_EXTRA* (C programs), *CXX_LIBS_EXTRA* (C++ programs) and *FC_LIBS_EXTRA* (Fortran programs).

2.5.4. 64-bit architectures

Most 64-bit architectures do not require specific flags for successful builds. However, in some situations, in particular on hybrid 32/64-bit systems, it is necessary to add such flags to select the proper binary object format. In these cases, you may want to use the *--enable-64bit-flags* option, in particular if you encounter link-time and/or run-time problems.

Please note that the support for 64-bit architectures is still incomplete and will be reworked during the next development cycle of ABINIT.

2.5.5. Overriding the build-system settings

In some extremely rare configurations, the build system is not able to set properly the build parameters. Should this happen, you can override all build-system settings by directly providing the flags from the command line. Compile flags may be specified through the *CFLAGS* (C programs), *CXXFLAGS* (C++ programs) and *FCFLAGS* (Fortran programs) environment variables. Link flags may be provided through *CC_LDFLAGS* (C programs), *CXX_LDFLAGS* (C++ programs) and *FC_LDFLAGS* (Fortran programs). Linking additional libraries should be done through *CC_LIBS* (C programs), *CXX_LIBS* (C++ programs) and *FC_LIBS* (Fortran programs).

If you had to that, please report a bug to https://bugs.launchpad.net/abinit/+filebug, providing as much information as you can on the architecture, including and the compiler, including how to get the version number of the compiler on the command line and the output of this command. We will add support for your case as soon as we can.

2.6. MPI options

In addition to serial optimization, ABINIT provides parallel binaries relying upon the MPI library. If you do not know what MPI stands for, then you really need the help of a computer scientist before reading this section. First let us make clear that we cannot provide you with any support to install MPI. If you need to do it, we advise you to ask help to your system and/or network administrators, because it will likely require special permissions and fair technical skills. In many cases you will already have a working installation of MPI at your disposal, and will at most need some information about its location.

Providing extended support for MPI is extremely delicate: there is no standard location for the package, there are concurrent implementations following different philosophies, and Fortran support is compiler-dependent. Moreover, there might be several versions of MPI installed on your system, and you have to choose one of them carefully. In particular, if you want to enable the build of parallel code in ABINIT — which you will likely do — you have to build ABINIT with the same Fortran compiler that has been used for MPI.

The MPI options provided by the build system are summarized in the following table.

Option	**Description**
`--enable-mpi`	Enable MPI support
`--enable-mpi-io`	Enable parallel I/O [default=no]
`--enable-mpi-trace`	Enable MPI time tracing [default=no]
`--with-mpi-prefix=PATH`	Prefix for the MPI installation
`--with-mpi-level=NUMBER`	MPI implementation level (1 or 2)
`--with-mpi-includes=FLAGS`	MPI include flags
`--with-mpi-libs=LIBS`	MPI libraries to append at the link stage
`--with-mpi-runner=PROG`	Full path to the MPI runner program

If --enable-mpi is left unset, there are 2 ways of enabling the build of parallel binaries: either by using the --with-mpi-prefix option, or by specifying MPI-capable compilers from the command line, through the CC (C compiler), CXX (C++ compiler) and FC (Fortran compiler) environment variables. Please note that you have to specify --with-mpi-runner manually in the latter case.

If "--enable-mpi" is set to "yes", the parallel code will be built only if a usable MPI implementation can be detected. If the "--with-mpi-prefix" option is provided, *enable_mpi* is automatically set to "yes" and the build system tries to find a usable generic MPI installation at the specified location very early during the configuration. If this step is successful, the compilers and the runner provided by MPI are used in *lieu* of the user-specified ones, and no further test is performed. If "--with-mpi-prefix" is not present, the build of parallel code will be deactivated unless "--enable-mpi" is explicitely set to "yes".

If the first attempt fails, a second one is undertaken once the compilers have been configured. The build system then checks whether the compilers are able to build MPI source code natively, taking advantage of the user-specified parameters. If successful, a MPI runner will be looked for using the *PATH* environment variable. If something goes wrong, the build of parallel code will be automatically disabled. In such a case, and as a last resort, the user may force the build through the use of "--enable-mpi=manual".

Additional levels of parallelization may be activated, though they are still experimental and meant to be used by developers only:

- "--enable-mpi-io": parallel file I/O;
- "--enable-mpi-trace": parallel time tracing.

You will find a detailed description of all these options in the source package of ABINIT, within the MPI support section of the " abinit/doc/build/config-template.ac" template. We warmly recommend you to have a close look at this file and to use it as much as you wish.

2.7. External libraries

The *configure* script of ABINIT provides a unified way of dealing with external libraries, by means of a trigger (enable/disable) and two specifiers (for include and link flags) for each package. Below the surface, things are however much more complex: some libraries are required by ABINIT, others not; some are contained within the source package, others are too big to be included; a few of them depend on other external libraries, which may or may not be found within the package. The current situation is summarized in table 2.2, and the corresponding options are described in table 2.3.

When a library is required and cannot be found outside the source package, the build system systematically restores consistency by ignoring user requests and disabling the

Library	**Internal**	**Required**	**Depends**	**Note**
bigdft	yes	no	—	
etsf-io	yes	no	netcdf	
etsf-xc	yes	no	—	*Needs more testing*
fftw	no	no	—	
fox	yes	no	—	*Currently unsupported*
linalg	yes	yes	—	
netcdf	yes	no	—	
wannier90	yes	no	—	*Test library for the plug-in feature*

Table 2.2.: Specifications of the ABINIT libraries.

corresponding support.

Providing automatic external library detection lead to complicate the build system too much, and jeopardized its maintainability. Hence we decided to aim at maximum simplicity. This means that you need to provide the include and link flags yourself, just as you would do when directly calling the compiler, e.g.:

```
 ./configure \
 --enable-netcdf \
 --with-netcdf-includes=\textquotedbl{}-I/opt/etsf/include/g95\textquotedbl{}
\
 --with-netcdf-libs=\textquotedbl{}-L/opt/etsf/lib -lnetcdf\textquotedbl{}
```

2.8. Other options

The *configure* script provides a few more options. Though most of them will only be used in specific situations, they might prove very convenient in these cases. The full list of special options may be found in table 2.4.

The build system of ABINIT makes it possible to use config files to store your preferred build parameters. A fully documented template is provided in the source code under *abinit/doc/build/config-template.ac*, along with a few examples in *abinit/doc/build/config-examples/*. After editing this file to suit your needs, you may save it as *$HOME/.abinit/build/<hos* to keep these parameters as per-user defaults. Just replace *<hostname>* by the name of your machine, excluding the domain name. E.g.: if your machine is called *myhost.mydomain*, you will save this file as *$HOME/.abinit/build/myhost.ac*. You may put this file at the top level of an ABINIT source tree as well, in which case its definitions will apply to this particular tree only. Using config files is highly recommended, since it saves you from typing all the options on the command-line each time you build a new version of ABINIT.

Option	**Description**
`--enable-bigdft`	Enable support for the BigDFT wavelet library [default=yes]
`--with-bigdft-includes`	Include flags for the BigDFT library
`--with-bigdft-libs`	Library flags for the BigDFT library
`--enable-etsf-io`	Enable support for the ETSF I/O library [default=no]
`--with-etsf-io-includes`	Include flags for the ETSF I/O library
`--with-etsf-io-libs`	Library flags for the ETSF I/O library
`--enable-etsf-xc`	Enable support for the ETSF exchange-correlation library [default=no]
`--with-etsf-xc-includes`	Include flags for the XC library
`--with-etsf-xc-libs`	Library flags for the ETSF XC library
`--enable-fftw`	Enable support for the FFTW library [default=no]
`--enable-fftw-threads`	Enable support for the threaded FFTW library [default=no]
`--with-fftw-libs`	Library flags for the FFTW library
`--enable-fox`	Enable support for the FoX I/O library [default=no]
`--with-fox-includes`	Include flags for the FoX I/O library
`--with-fox-libs`	Library flags for the FoX I/O library
`--with-linalg-libs`	Library flags for the linalg library
`--enable-netcdf`	Enable support for the NetCDF I/O library [default=no]
`--with-netcdf-includes`	Include flags for the NetCDF library
`--with-netcdf-libs`	Library flags for the NetCDF library
`--enable-wannier90`	Enable support for the Wannier90 library [default=no]
`--with-wannier90-includes`	Include flags for the Wannier90 library
`--with-wannier90-libs`	Library flags for the Wannier90 library

Table 2.3.: External library options of ABINIT.

Option	**Description**
`--enable-config-file`	Read options from config files [default=yes]
`--with-config-file=FILE`	Specify config file to read options from
`--enable-cclock`	Use C clock for timings [default=no]
`--enable-stdin`	Read file lists from standard input [default=yes]

Table 2.4.: Special options of ABINIT.

Part II.

Developers

3. Preprocessing macros

3.1. Propagating information to the source code

While many arguments of the configure script control the way ABINIT is built, some of them --- in addition to the results of the tests performed at configure-time --- greatly influence what will be built. In the latter case, the information has to be propagated up to the source code, which is done by means of preprocessing macros. They are created by the `AC_DEFINE` macro of Autoconf, or specified by the user on the command line.

Macros are *(name,value)* pairs allowing the mapping of a sequence to another. Names are usually single words, while values usually range from simple numbers to very complex sequences of instructions. During compilation, *name* is replaced by *value* every time it is encountered, this process being called *macro expansion*. Special lines, starting with the '#' character in C, allow for more operations on macros, like setting, unsetting or tests. Last but not least, the concept of macro is not limited to any programming language, and macros are indeed ubiquitous in the programming world.

The build of ABINIT leads to the creation of many preprocessing macros (73 in ABINIT 5.5), which are stored in *config.h*. Besides command-line options, this file is the only link between the build system and the source code of ABINIT, and this is the reason why all of them must include it at their very beginning.

3.2. Naming conventions

As far as preprocessing directive names are concerned, ABINIT abides strictly by the GNU Coding Standards. This means in particular that:

- all user-defined compiler directives must be upper case;
- all names must start with a letter;
- names may contain capital letters, digits and underscores only;

Directives related to features that may or may not be present depending on the configuration must begin by the keyword *HAVE_ **, e.g. *HAVE_CONFIG_H*, *HAVE_NETCDF*, *HAVE_ETSF_XC*, etc.

3.3. If statements

If statements should all begin with '`#if`'. We kindly ask you not to use '`#ifdef`', but '`#if defined`' instead. A line ending an *if* statement must contain the '`#endif`' keyword only. The same holds for '`#else`'.

Here is a typical example:

```
 \#if defined HAVECONFIGH \#include \textquotedbl{}config.h\textquotedbl{}
\#endif
```

We thank you in advance for following these simple rules, as it will greatly simplify the automatic checks and fixes of the source code.

3.4. Preprocessing macros of ABINIT 5

3.4.1. Generic macros

CONTRACT	Design-by-contract code
HAVE_CONFIG_H	Mandatory: use config.h if present

3.4.2. Architecture-related macros

OS_IRIX	IRIX operating system
OS_LINUX	Linux operating system
OS_MACOSX	Mac OS X operating system
OS_WINDOWS	DOS/Windows operating system
VMS	VAX/VMS architecture

3.4.3. Optional library macros

HAVE_COMPAQ_FFT	HP/Compaq/DEC FFT library
HAVE_FFTW	FFTW library
HAVE_FFTWTHREADS	FFTW library (threaded version)
HAVE_FFTW3	FFTW3 library
HAVE_FFTW3THREADS	FFTW3 library (threaded version)
HAVE_HP_MLIB	HP mathematical library
HAVE_SCALAPACK	SCALAPACK linear algebra library
HAVE_SGI_MATH	SGI mathematical library
HAVE_IBM_ESSL	IBM mathematical library
HAVE_IBM_ESSL_OLD	IBM mathematical library (old version)
HAVE_NEC_ASL	NEC mathematical library
HAVE_NETCDF	NetCDF file I/O library
HAVE_BIGDFT	BigDFT wavelet library
HAVE_ETSF_IO	ETSF file I/O library
HAVE_FOX	Fortran XML I/O library
HAVE_LIBXC	Octopus exchange-correlation library

3.4.4. MPI macros

MPI macros may not be included in the *config.h* file, as it would preclude the build of sequential code. They should be specified within the compiler command line. The following table gives the full list of permitted MPI macros and the way they are managed. Manual handling is done through the *--with-mpi-cppflags* option of configure.

Option	**Description**	**Management**
MPI	MPI statements follow	Build system
MPI1	MPI version 1	Manual
MPI2	MPI version 2	Manual
MPI3	MPI version 3	Manual
MPI_EXT	MPI HTOR routines (?)	Manual
MPI_FFT	Parallel FFT	Build system
MPI_IO	Parallel I/O	Build system
MPI_TRACE	Timing within parallel routines	Build system

3.4.5. Compiler macros

FC_ABSOFT	ABSoft Fortran compiler
FC_COMPAQ	HP/Compaq/DEC Fortran compiler
FC_FUJITSU	Fujitsu Fortran compiler
FC_GNU	GNU Fortran compiler (gfortran)
FC_G95	G95 Fortran compiler (g95)
FC_HITACHI	Hitachi Fortran compiler
FC_HP	HP Fortran compiler
FC_IBM	IBM XL Fortran compiler
FC_INTEL	Intel Fortran compiler
FC_MIPSPRO	SGI MipsPro Fortran compiler
FC_NAG	NAGWare Fortran compiler
FC_NEC	NEC Fortran compiler
FC_PGI	PGI Fortran compiler
FC_SUN	Sun Fortran compiler

The same holds for C and C++ compilers.

3.4.6. Fortran-specific macros

HAVE_FORTRAN_EXIT	The Fortran compiler accepts exit()
USE_CCLOCK	Use C clock for timings

3.4.7. Renamed macros

Option	**Replaced by**	**Version**
__IFC	FC_INTEL	5.1
ibm	FC_IBM	5.1
NAGf95	FC_NAG	5.1
mpi	MPI	5.1
MPIEXT	MPI_EXT	5.1
TRACE	MPI_TRACE	5.1
FFTW	HAVE_FFTW	5.1
FFTWTHREADS	HAVE_FFTWTHREADS	5.1
bim	HAVE_IBM_ESSL	5.1
bmi	HAVE_IBM_ESSL	5.1
cen	HAVE_NEC_ASL, FC_NEC	5.1
dec_alpha	FC_COMPAQ	5.1
hp	HAVE_HP_MLIB, FC_HP	5.1
hpux	HAVE_HP_MLIB	5.1
nec	HAVE_NEC_ASL, FC_NEC	5.1
nolib	HAVE_COMPAQ_FFT	5.1
sgi	HAVE_SGI_MATH, FC_MIPSPRO	5.1
sr8k	FC_HITACHI	5.1
vpp	FC_FUJITSU	5.1
__VMS	VMS	5.1
P6	i386	5.1
macosx	OS_MACOSX	5.1
CHGSTDIO	READ_FROM_FILE	5.1

3.4.8. Unmaintained macros

OPENMP	OpenMP parallelism
T3E	Cray T3E architecture
TEST_AIM	Optional checks for AIM

3.4.9. Removed macros

The following preprocessing macros have been removed from the ABINIT source code.

Option	**Last version**	**Comments**
OLD_INIT	4.6	Was used in *src/04wfs/wfconv.F90* to initialize the wavefunctions, and has been replaced for a long time by a more efficient method.
PGIWin	4.6	The PGI Fortran compiler is no longer used to build Abinit under Windows, since it is too buggy.
ultrix	4.6	Ultrix was an operating system based on a 4.2BSD Unix with some features from System V. It was first released in 1984. Its purpose was to provide a DEC-supported native Unix for VAX. The last major release of Ultrix was version 4.5 in 1995, which supported DECstations and VAXen. There were some subsequent Y2K patches. There has been no ABINIT user on Ultrix quite some time.

4. Adding external libraries / plug-ins

4.1. Overall procedure

For all the tasks to perform, just use the existing libraries as examples and tutorials as soon as you have a doubt. All paths are given from the top source directory. Please note that this procedure has been elaborated and complexified progressively and is now being reworked in order to greatly simplify it.

1. Create a new directory in *plugins/*, with a short and explicit name.
2. Copy the tarball to the new directory and go there. Its name should be: *<package_name>.tar.gz*. The package name may of course include a version number.
3. Create a RoboDOC header briefly describing the library. The file should have the same name as the directory, plus leading and trailing underscores. Suggestion: copy the one from *lib/netcdf/* and start from it.
4. Create a makefile with the same name as the directory, plus a ".mk" extension. It will tell the build system how to perform the various steps of the library build: uncompress, configure, build, install. Suggestion: copy the one from *lib/netcdf/* and use it as a starting point.
5. Create a *abinit.amf* file containing a list of additional files to clean. It will basically consists in the libraries, binaries, headers and Fortran modules used by ABINIT. Suggestion: use *lib/netcdf/abinit.amf* to see which is the format to follow.
6. Edit *config/specs/extlibs.cf*: add one line for your library following the specified format. Put the most important module only in the second column if your library has several C/C++ headers or Fortran modules. The name of the library should be the same as for the directory.
7. Edit *config/specs/libraries.cf*:

    a. in `abinit_libs`, add the library **after** the others it could depend on and **before** the libraries depending on it;

   b. in `abilibs_specs`, copy the "netcdf" line, changing only its name and removing `|ABI_LIB_INC` if your library has no C/C++ header and no Fortran module; here the order is external/internal, then alphabetical, so you should add your library before the "defs" line;

   c. describe the dependencies in `abilibs_deps`.

   The name of the library should be the same as for the directory.

8. Edit *config/specs/binaries.cf*: add the library to the dependencies of every binary that may use it; the line should be put **before** the libraries it depends on and **after** the libraries that depend on it. The name of the library should be the same as for the directory.

9. Edit *config/specs/options.cf*: add the `--enable-*` and `--with-*` options for your library, with short and precise info strings. Use *netcdf* as a typical example.

10. Edit *config.mk.in*: add the build flags of the library at the end of the file. You may copy/paste from another external library, yet be careful to change **ALL** the references.

11. Edit *config/m4/tricks.m4*: add a "tricky" macro at the end of the file. You may leave it empty, just as many of them already are.

12. Edit *configure.ac*:

    a. at the beginning, where external packages are declared;

    b. at the end, where the external library macros are called.

    Add the relevant information using what is there as examples.

13. Run *config/scripts/makemake*, and watch carefully any possible error message.

14. Run *configure*, and watch carefully any possible error message.

15. Run *make*, and watch carefully any possible error message.

4.2. The library makefile

The build system expects a few things from the makefile *<lib_name>.mk* managing the package stored in *<package_name>.tar.gz*:

- it should include *config.mk*, in order to transmit the build parameters to the package's own build system;

- it should uncompress in *lib/<lib_name>/<package_name>*, and thus move the uncompressed directory afterwards if it not the case (see *lib/fox/fox.mk* for an example);

- it should install in *lib/<lib_ name>/tmp*, so that the build system of ABINIT may import all required data by itself if the package is managed by the Autotools.

Please read all the library makefiles contained within ABINIT before writing yours, this might help you a lot.

4.3. Fine-tuning *abinit.amf*

Once you manage to build your library properly, run a *make clean* from within and add all remaining files that should have been swept off to the list contained in *abinit.amf*.

Part III.

Maintainers

5. Extending the build system

5.1. Prerequisites

In order to efficiently tweak the build system, you will need to have a good experience of some basic Unix utilities: *cat*, *grep*, *sed*, *awk*, *cut*, *tr*, *tee*, *wc*. A long familiarity with ABINIT and an active participation to the developments occuring within the last six months, though mandatory, will not suffice. You should already be fluent in the following areas as well:

- Bourne-type shell scripting
  (`http://en.wikipedia.org/wiki/Bourne_shell`);
- Perl scripting
  (`http://en.wikipedia.org/wiki/Perl`);
- Python scripting
  (`http://en.wikipedia.org/wiki/Python_%28programming_language%29`);
- M4 scripting
  (`http://en.wikipedia.org/wiki/M4_%28computer_language%29`);
- Makefile writing
  (http://en.wikipedia.org/wiki/Makefile);
- Link editing
  (http://en.wikipedia.org/wiki/Linker);
- Regular expression designing
  (http://en.wikipedia.org/wiki/Regular_expression).

Just as when developing for ABINIT, you will need a fully working installation of the GNU Autotools. And here is what distinguishes the maintainer from the developer: you will need to know how they work and understand their principles. Their respective documentations may be found at the following addresses:

- Autoconf ⟶ http://www.gnu.org/software/autoconf/manual/
- Automake ⟶ http://sources.redhat.com/automake/automake.html
- Libtool ⟶ http://www.gnu.org/software/libtool/manual.html

We strongly urge you to read them if you want to know what you are doing.

Last but not least, you will need to have Bazaar (http://bazaar-vcs.org/) installed on your development machine, since the delicate character of your contributions will require real-time interactions with other maintainers and/or developers, be it for bug fixing or testing.

5.2. Adding scripts

If your extension influences exclusively the pre-build stage of ABINIT, i.e. it prepares the way for the Autotools, you may add it in the form of a script in *~abinit/config/scripts/*. Please follow the conventions already adopted for the other scripts. When done, do not forget to add a call to your script in *~abinit/config/scripts/makemake*, and remember that *makemake* expects to be called from the top-level directory of the source tree.

If your script is producing M4 macros, the names of the files containing them must be prefixed by "auto-".

5.3. Adding M4 macros

When you want to propagate information up to the Makefiles of ABINIT, the recommended way to extend the build system is by writing M4 macros. The best practice is to create a new file in *~abinit/config/m4/*, following the conventions adopted for the other files. If at a later time your contribution is approved, it may be redispatched into other files.

5.4. Editing *configure.ac*

The *configure.ac* file is the spinal cord of the build system. Every single character of this file plays a well-defined role, and is present for a carefully-thought logical reason. In particular, the order of the lines is of critical importance to the proper functioning of the whole build system. That is why this file should only be edited with **extreme care** by persons having a good knowledge of shell-scripts, M4, Autoconf, Automake, Libtool and the ABINIT build system. Messing-up with the instructions present there without a sufficient experience in these matters will **for sure** lead to catastrophic consequences, and may even result in massive loss of data. To summarize, **YOU EDIT THIS FILE AT YOUR OWN RISKS**. Believing you are more clever than the designers of the ABINIT build system will not save you.

The *configure* script is generated from *configure.ac* by Autoconf. As such, *configure* should **NEVER EVER** been edited.