

LOFAR build instructions

This page describes how to obtain and build LOFAR the software.



Please take note that ASTRON only thoroughly builds and tests the LOFAR software on their production systems that contain RHEL5 32-bit and Ubuntu 10.04 LTS/Ubuntu 12.04 LTS 64-bit Linux distros. For other distros and/or versions we cannot guarantee that a build will create a fully and correctly working package. Support for this is not provided.

Prerequisites

Mind that to be able to build LOFAR, first you need to have various other custom packages available on your system. below is a list of packages and versions that have been conformed to work (other versions may work, but we have not tested that). Note that this is not a complete list:

- boost 1.40 or larger
- cfitsio 3.23 or larger
- wcslib 4.8 or larger
- cmake 2.8 or larger
- hdf5 1.8.4 or larger
- casacore 2.0.3 or latest version and its dependencies (lapack, blas, fftw, etc.); see <https://github.com/casacore>
- casacore-python 2.0.1 or latest version (see <https://github.com/casacore>)
- casarest latest development version (see <https://github.com/pkgw/casarest> or its local mirror <https://svn.astron.nl/casarest/>)
- log4cplus 1.0.4 or larger
- AOFlagger 2.7.1 or larger (<http://sourceforge.net/projects/aoflagger/>)

Some Linux distributions may have ready to install packages available for some of these, for other distros you may need to build these from code or search hard for binary installations. You may require your system managers help to set all these up properly.

CMake

LOFAR uses [CMake](#) as build tool. For general information on CMake, please refer to the [CMake documentation pages](#).

You will need CMake 2.8 or later in order to build the LOFAR software. Most Linux distributions contain CMake as a binary package. If yours doesn't, or if it's too old, you can [download the CMake sources](#) and build CMake yourself.

Getting Started

Step 1

Make sure you have a working copy of (part of) the LOFAR software tree. To check out the whole tree:

```
$ cd <working directory>
$ svn checkout https://svn.astron.nl/LOFAR/trunk LOFAR
```

Alternatively, you can do a minimal checkout of the LOFAR tree and let the build system do a checkout of the parts that are needed for your specific build.

```
$ cd <working directory>
$ svn checkout -N https://svn.astron.nl/LOFAR/trunk LOFAR
$ svn update LOFAR/CMake
```

This requires an account on the SVN server. There is also a guest read-only account; mail Arno Schoenmakers or Marcel Loose for details if you need this.

Please refer to [The LOFAR Subversion Repository](#) page (password protected) for more information on how to check out LOFAR software.

Step 2

Create a build directory, preferably outside of the source tree. The name of the directory must adhere to the naming conventions described in section 3.6 of [LOFAR Build Environment](#). So, for example, when using the GNU compiler suite to build a debug version of the software, you'd have to create a build directory named `gnu_debug`.

```
$ mkdir -p build/gnu_debug
```

Step 3

Run `cmake` from the build directory. You must provide the (relative) path to the top-level `CMakeLists.txt` file (in this example `<working_dir>/LOFAR`). You can give a list of packages to build using the `-DBUILD_PACKAGES` option:

```
$ cd build/gnu_debug
$ cmake -DBUILD_PACKAGES="Package1 Package2" <working_dir>/LOFAR
```

If you plan to run `make install` to install the built software in a directory of your choice (instead of in the top level build directory), you will have to define `CMAKE_INSTALL_PREFIX` on the command-line:

```
$ cd build/gnu_debug
```

```
$ cmake -DBUILD_PACKAGES="Package1 Package2" \
        -DCMAKE_INSTALL_PREFIX:PATH=<installpath> \
        <working dir>/LOFAR
```

Most users probably need the Offline programs (NDP3, BBS, Imager) and/or the pipeline framework. To build all these, you can provide as package `Offline`.

A complete package list, useful for those who are familiar with the code organization, can be found in the file `CMake/LofarPackageList.cmake`.

Step 4

When CMake completes without errors, you can run `make` to actually build the software. You can use the curses-based `ccmake` (or use `make edit-cache`) to edit CMake's cache file to modify any of the cache variables (e.g., which LOFAR packages to build, paths to third-party libraries and/or include files, etc.).

```
$ make [-j n] [-k]
```

Where the `-j n` option denotes parallel build using `n` cores of your system (much faster).

If you want the build to continue even when encountering errors in the build process, you can add the `-k` flag to the `make` command. For instance:

If you want the build products to be installed as well, add `install` to the `make` command:

```
$ make install
```

Note that the `install` option only works when the previous `make` step has completed without errors.

Note

Most (but not all!) changes to CMake files (`*.cmake` or `CMakeLists.txt`) will be detected by CMake, and will trigger a (re)run of `cmake` whenever needed. So typing `make` is usually sufficient to get a correct (re)build of the software.

Build Options

Build options can be specified in two ways. The preferred, “static” way of doing this is through the different variants files. These settings can be overridden by the user, either by setting options on the command-line when invoking `cmake`, or by edit them by using the semi-graphical environment `ccmake`.

Packages

To find external packages (like casacore, aoflagger, etc) one can add directory pointers to the CMake command line like these:

```
-DCASACORE_ROOT_DIR=/opt/cep/casacore/current
-DAOFLAGGER_ROOT_DIR=/opt/cep/aoflagger/current
```

etc.

Available Options

The following options are currently available. This is neither an exhaustive, nor an authoritative list. It merely serves as an example to which global build options may be set.

Option	Description	Default value
BUILD_DOCUMENTATION	Build code documentation	OFF
BUILD_SHARED_LIBS	Build shared libraries	ON
BUILD_STATIC_EXECUTABLES	Build statically linked executables	OFF
BUILD_TESTING	Build test programs	ON
LOFAR_SVN_UPDATE	Always do an svn update	<undefined>
LOFAR_VERBOSE_CONFIGURE	Be verbose when configuring	ON
USE_BACKTRACE	Use backtraces in exceptions	ON
USE_LOG4CPLUS	Use the Log4Cplus logging package	ON
USE_LOG4CXX	Use the Log4Cxx logging package	OFF
USE_MPI	Compile with MPI support	OFF
USE_OPENMP	Compile with OpenMP support	OFF
USE_SHMEM	Use shared memory	ON
USE_SOCKETS	Use network sockets	ON
USE_THREADS	Use thread support	ON

Some options are mutually exclusive (e.g., USE_LOG4CPLUS, and LOG4CXX cannot be used simultaneously). These restrictions are checked.

LOFAR_SVN_UPDATE uses three states. When <undefined>, only files that are missing but needed are updated. When OFF, files are *never* updated (this is useful if you don't have access to the SVN server, or if you're working with an exported source tree). When ON, files are *always* updated.

From:
<https://www.astron.nl/lofarwiki/> - **LOFAR Wiki**

Permanent link:
https://www.astron.nl/lofarwiki/doku.php?id=public:user_software:lofar&rev=1453370261

Last update: **2016-01-21 09:57**

