

Building the LOFAR software stack

LOFAR uses [CMake](#) as build tool. This page provides information that is specific for the use of CMake for the LOFAR software. It will focus on how to write CMakeLists.txt files for LOFAR, on LOFAR-specific CMake macros, etc.

For general information on CMake, please refer to the [CMake documentation pages](#).

For the old guys among us: See [Why we migrated from Autotools to CMake](#) for some information about the rationale of moving from Autotools to CMake in the past...

Prerequisites

You will need CMake 2.6 or later in order to build the LOFAR software. Most development was done using CMake 2.6.2. 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

The CMake build environment follows the same naming convention rules w.r.t. build directory names as the old Autotools-based build environment. However, the per-package configure-compile-install cycle has been replaced by one global cycle. This means that libraries and binaries will no longer be built in a subdirectory of each source package, but in a separate build-directory which will mirror the directory structure of the source tree. For those of you that do out-of-source-tree builds with rub this is nothing new.

Step 1

Make sure you have a working copy of (part of) the LOFAR software tree. Please refer to [The LOFAR Subversion Repository](#) page for more information on how to check out LOFAR software.

```
$ cd $HOME
$ 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 $HOME
$ svn checkout -N https://svn.astron.nl/LOFAR/trunk LOFAR
$ svn update LOFAR/CMake
```

Note On newer versions of svn you should probably use `–depth=immediates` instead of `-N`.

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 `$HOME/LOFAR`). You can give a list of packages to build using the `-DBUILD_PACKAGES` option:

```
$ cd build/gnu_debug
$ cmake -DBUILD_PACKAGES="Package1 Package2" $HOME/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> \
    $HOME/LOFAR
```

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
```

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:

```
$ make -k
```

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

```
$ make install
```

Note that the install option only works when the make 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.

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 by the [LofarOptions](#) macro. Furthermore, this macro calls `lofar_find_package` for each package that is marked to be used. It is a fatal error if that package cannot be found.

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.

Skeleton of a LOFAR Package

LOFAR packages come in two 'flavors': leaf-packages and meta-packages.

Leaf-packages

Leaf-packages consist of source code. Most of the time this is C++ code, but some leaf-packages consist purely of Python or Java. A leaf-package usually contains the following directories:

- `include`, containing the public header files (the interface) of the package;
- `src`, containing the source files (the implementation); and
- `test`, containing the test programs (source and/or scripts).

Each of these directories contains a `CMakeLists.txt` file, but each has a different structure.

The top-level directory

The `CMakeLists.txt` file in the top-level package directory defines the LOFAR package, lists its internal and external dependencies, and adds the subdirectories containing the source code.

Define the package

A leaf-package must be defined using the macro `lofar_package` (see [LofarPackage](#)). Calling this macro ensures that dependent packages are added to the build, and that the compiler's include path and the list of libraries to link against are setup correctly. The package name must be unique in the complete source tree.

For example:

```
lofar_package(BBSKernel 1.0 DEPENDS Blob Common ParmDB)
```

defines that the package `BBSKernel` has version 1.0, and depends on the packages `Blob`, `Common`, and `ParmDB`.

Define external dependencies

Dependencies on external packages must be defined using the macro `lofar_find_package` (see [LofarFindPackage](#)), which is a wrapper around the CMake command [find_package](#).

```
include(LofarFindPackage)
lofar_find_package(Boost REQUIRED)
lofar_find_package(Casacore REQUIRED COMPONENTS casa measures ms scimath
tables)
```

In this example we see that the Boost package is required, and that a number of Casacore components are required. It is the task of the FindBoost and FindCasacore modules to search for the required components and return variables that contain the include path, library path, and a list of libraries to be linked against.

Add subdirectories

Finally, we should add the subdirectories that contain the header files (include), the source files (src), and the test programs (test).

```
add_subdirectory(include/BBSKernel)
add_subdirectory(src)
add_subdirectory(test)
```

The 'include' directory

The skeleton of the CMakeLists.txt file in the include directory is roughly as follows.

Create symbolic link to include directory

CMake does not install LOFAR packages on a per-package basis as the old Autotools-based build-environment does. As a result, we must somehow tell other packages where they can find the header files of packages that they depend on. This is done by creating a symbolic link in the binary include directory to the current source directory.

```
execute_process(COMMAND ${CMAKE_COMMAND} -E create_symlink
  ${CMAKE_CURRENT_SOURCE_DIR}
  ${CMAKE_BINARY_DIR}/include/${PACKAGE_NAME})
```

Header files to install

Header files that must be installed, because they are part of the package's interface, should be listed

```
install(FILES
  Evaluator.h
  Equator.h
  Exceptions.h
  ...
  DESTINATION include/${PACKAGE_NAME})
```

The 'src' directory

The skeleton of the CMakeLists.txt file in the src directory is roughly as follows.

LofarPackageVersion

[LofarPackageVersion](#) is responsible for generating and/or updating the files `Package__Version.h`, `Package__Version.cc`, and `version<pkg>` (where `<pkg>` is the name of the current package).

```
include(LofarPackageVersion)
```

This line should be the first, or at least, it should appear before any reference is made to the files that are generated by this macro. In practice, this means that this line should appear before a `lofar_add_library`, or `lofar_add_bin_program` command.

Add a library

Add a library to the current project, using the specified source files.


```
lofar_add_library(bbskernel
  Package__Version.cc
  Evaluator.cc
  ...
)
```

The library will be installed in `<prefix>/<libdir>`, where `<libdir>` can either be `lib`, or `lib64`, depending on processor architecture and Linux distribution.

Add a binary

Add a binary program to the current project, using the specified source files. The first source file must contain the `main()` function. The executable program will be installed in `<prefix>/bin`.

```
lofar_add_bin_program(versionbbskernel versionbbskernel.cc)
```

 If you want to create a non-installable binary, you can use the `lofar_add_executable` macro. This macro will take care of linking in all the dependent libraries, but will not mark the binary for install.

The 'test' directory

LofarCTest

The first line in the `CMakeLists.txt` file should be

```
include(LofarCTest)
```

This macro will add the current source directory to the compiler's include path, and generate a

wrapper script around the `runtest.sh` script to define some required environment variables.

Add tests

Next tests can be added, using the `lofar_add_test` macro

```
lofar_add_test(tFillRow tFillRow.cc)
lofar_add_test(tJonesCMul3 tJonesCMul3.cc utils.cc)
```

In the first case the test `tFillRow` is passed to `lofar_add_test()` and is a linux shell-script. For the test to be run it is required to compile `tFileRow.cc` into an application called `tFileRow`. In the second case the test `tJonesCMul3` requires `tJonesCMul3.cc` and `utils.cc` to be compiled first.

Hence, at minimum the following files would exist in the 'test' directory for C/C++ tests:

- CMakeLists.txt
- [test_name].sh
- [unit_test].cc

For Python tests an extra `.run`-file is necessary, so that a test directory would at least contain:

- CMakeLists.txt
- [test_name].sh
- [test_name].run
- [unit_test].py

For both languages the `.sh`-file will call `runtest` with a given test, e.g for the `RATaskSpecifiedService`:

```
#!/bin/sh

./runtest.sh tRATaskSpecified
```

For Python this will not call the `.py`-file containing the test immediately, but will call the `.run`-file.

The `.run`-file will then contain commands related to the Python test facilities (and the actual call to the file containing the Python test):

```
#!/bin/bash

# Run the unit test
source python-coverage.sh
python_coverage_test "rataskspecifiedservice*" tRATaskSpecified.py
```


The LOFAR tree uses CMake for both building and testing. In order to test the whole LOFAR tree, issue the following command from the build directory (`build/gnu_debug`):

```
make test
```

In order to run the test for a specific sub-system, navigate to its sub-directory (not its 'test'-directory) and issue the same command. The Test results will be written to files in the sub-directory

'./Testing/Temporary/':

- CTestCostData.txt
- LastTest.log
- LastTestsFailed.log

 You may find it convenient (during development) to just call the Python test script immediately (from the source directory, not the build directory), instead of calling 'make test' in order to see the detailed test output immediately.

Meta-packages

A meta-package is a package that consists of one or more (meta-)packages that are subdirectories of the current directory. For example, the `BBSKernel` package is part of the meta-package `Calibration`, which in turn is part of the meta-package `CEP`.

The structure of the `CMakeLists.txt` file of a meta-package is much simpler than that of an ordinary package.

Adding a package

Packages are added to the build with the CMake macro `lofar_add_package` (see [LofarPackage](#)).

For example, the `CMakeLists.txt` file for the `Calibration` meta-package contains the following two lines:

```
lofar_add_package(BBSKernel)
lofar_add_package(BBSControl)
```

This instructs CMake to add the packages `BBSKernel`, and `BBSControl` to the build, unless any of them was explicitly excluded from the build (e.g., when `BUILD_BBSKernel` was set to `OFF`).



You should *not* use `lofar_package` to define a meta-package. Only leaf-packages must be defined.

Note

It is not an error if the source directory of the added package does not exist (e.g., `BBSControl` has not been checked out). CMake will then simply set the option `BUILD_BBSControl` to `OFF`, so that it will be excluded from the build.

The top-level LOFAR package

The top-level LOFAR package is a somewhat different meta-package. Its `CMakeLists.txt` file

contains initialization and finalization sections; code that only needs to be executed once.

Initialization

The first line in the `CMakeLists.txt` file sets the minimum required version of CMake.

```
cmake_minimum_required(VERSION 2.6)
```

LofarInit

Next we include the [LofarInit](#) macro file. This file *must* be included *before* the `project` command, because it will define the compilers to use (by reading the variants files). Note that at this stage the CMake variable `CMAKE_MODULE_PATH` is not yet set, so we need to specify the full path and filename to `LofarInit`.

```
include(CMake/LofarInit.cmake)
```

Project

Next we define the project. This causes CMake to check for the presence of working C and C++ compilers.

```
project(LOFAR)
```

LofarGeneral

Then we include the [LofarGeneral](#) macro file, which mainly performs a number of checks for the presence of system header files. This file *must* be included immediately *after* the `project` command.

```
include(LofarGeneral)
```

Main part

In the main part, the different meta-packages are added to the build. Note that the user has some control over which packages will be added to the build by defining `BUILD_PACKAGES` on the command-line.

LofarPackage

First, we need to include the file [LofarPackage](#), which defines two important macros: `lofar_package` and `lofar_add_package`.

```
include(LofarPackage)
```

Then, if the user did *not* define `BUILD_PACKAGES`, all existing meta-packages will be added to the build.

```
if(NOT DEFINED BUILD_PACKAGES)
  lofar_add_package(LCS)
  lofar_add_package(CEP)
  lofar_add_package(RTCP)
  lofar_add_package(SAS)
  lofar_add_package(MAC)
  lofar_add_package(LCU)
  lofar_add_package(SubSystems)
```

Remember that these packages do not have to be present, but that they will be added to the build, if they are present.

If the user *did* define `BUILD_PACKAGES`, then each of the packages listed will be added to the build.

```
else(NOT DEFINED BUILD_PACKAGES)
  separate_arguments(BUILD_PACKAGES)
  foreach(pkg ${BUILD_PACKAGES})
    lofar_add_package(${pkg} REQUIRED)
  endforeach(pkg ${BUILD_PACKAGES})
endif(NOT DEFINED BUILD_PACKAGES)
```

If any of these packages depends on other packages, then these will automatically be added to the build too. Packages specified in `BUILD_PACKAGES` are considered required. It is an error if one or more required packages are missing.

Finalization

When all packages have been processed by CMake, enough information has been gathered to generate a configuration file.

LofarConfig

The `LofarConfig` macro will generate the file `lofar_config.h` from the file `lofar_config.h.cmake` and place it in the binary `include` directory.

```
include(LofarConfig)
```

The header file `lofar_config.h` *must* be included in *each* compilation unit *before* any other header file.

Links

- [LOFAR CMake Macros](#)
- [CMake Equivalent of Autotools Commands and Options](#)
- [The LOFAR Subversion Repository](#)

From:

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

Permanent link:

https://www.astron.nl/lofarwiki/doku.php?id=public:user_software:documentation:lofar-cmake&rev=1483959265

Last update: **2017-01-09 10:54**

