

---

## Free Lossless Audio Codec (FLAC)

FLAC is open source software that can reduce the amount of storage space needed to store digital audio signals without needing to remove information in doing so.

The files read and produced by this software are called FLAC files. As these files (which follow the FLAC format) can be read from and written to by other software as well, this software is often referred to as the FLAC reference implementation.

FLAC has been developed by volunteers. If you want to help out, see CONTRIBUTING.md for more information.

### Components

FLAC is comprised of \* libFLAC, a library which implements reference encoders and decoders for native FLAC and Ogg FLAC, and a metadata interface \* libFLAC++, a C++ object wrapper library around libFLAC \* `flac`, a command-line program for encoding and decoding files \* `metaflac`, a command-line program for viewing and editing FLAC metadata \* user and API documentation

The libraries (libFLAC, libFLAC++) are licensed under Xiph.org's BSD-like license (see COPYING.Xiph). All other programs and plugins are licensed under the GNU General Public License (see COPYING.GPL). The documentation is licensed under the GNU Free Documentation License (see COPYING.FDL).

### Documentation

For documentation of the `flac` and `metaflac` command line tools, see the directory `man`, which contains the files `flac.md` and `metaflac.md`

The API documentation is in html and is generated by Doxygen. It can be found in the directory `doc/html/api`. It is included in a release tarball and must be build with Doxygen when the source is taken directly from git.

The directory `examples` contains example source code on using libFLAC and libFLAC++.

Documentation concerning the FLAC format itself (which can be used to create software reading and writing FLAC software independent from libFLAC) was included in previous releases, but can now be found on <https://datatracker.ietf.org/doc/draft-ietf-cellar-flac/> Additionally a set of files for conformance testing called the FLAC decoder testbench can be found at <https://github.com/ietf-wg-cellar/flac-test-files>

If you have questions about FLAC that this document does not answer, please submit them at the following tracker so this document can be improved:

---

<https://github.com/xiph/flac/issues>

## Building FLAC

All components of the FLAC project can be build with a variety of compilers (including GCC, Clang, Visual Studio, Intel C++ Compiler) on many architectures (including x86, x86\_64, ARMv7, ARMv8 and PowerPC) for many different operating systems.

To do this, FLAC provides two build systems: one using GNU's autotools and one with CMake. Both differ slightly in configuration options, but should be considered equivalent for most use cases.

FLAC used to provide files specifically for building with Visual Studio, but these have been removed in favor of using CMake.

## Building with CMake

CMake is a cross-platform build system. FLAC can be built on Windows, Linux, Mac OS X using CMake.

You can use either CMake's CLI or GUI. We recommend you to have a separate build folder outside the repository in order to not spoil it with generated files. It is possible however to do a so-called in-tree build, in that case `/path/to/flac-build` in the following examples is equal to `/path/to/flac-source`.

### CMake CLI

Go to your build folder and run something like this:

```
1 /path/to/flac-build$ cmake /path/to/flac-source
```

or e.g. in Windows shell

```
1 C:\path\to\flac-build> cmake \path\to\flac-source
```

(provided that cmake is in your %PATH% variable)

That will generate build scripts for the default build system (e.g. Makefiles for UNIX). After that you start build with a command like this:

```
1 /path/to/flac-build$ make
```

And afterwards you can run tests or install the built libraries and headers

---

```
1 /path/to/flac-build$ make test
2 /path/to/flac-build$ make install
```

If you want use a build system other than default add -G flag to cmake, e.g.:

```
1 /path/to/flac-build$ cmake /path/to/flac-source -GNinja
2 /path/to/flac-build$ ninja
```

or:

```
1 /path/to/flac-build$ cmake /path/to/flac-source -GXcode
```

Use cmake -help to see the list of available generators.

By default CMake will search for OGG. If CMake fails to find it you can help CMake by specifying the exact path:

```
1 /path/to/flac-build$ cmake /path/to/flac-source -DOGG_ROOT=/path/to/ogg
```

If you would like CMake to build OGG alongside FLAC, you can place the ogg sources directly in the flac source directory as a subdirectory with the name ogg, for example:

```
1 /path/to/flac-source/ogg
```

If you don't want to build flac with OGG support you can tell CMake not to look for OGG:

```
1 /path/to/flac-build$ cmake /path/to/flac-source -DWITH_OGG=OFF
```

Other FLAC's options (e.g. building C++ lib or docs) can also be put to cmake through -D flag. If you want to know what options are available, use -LH:

```
1 /path/to/flac-build$ cmake /path/to/flac-source -LH
```

## CMake GUI (for Visual Studio)

It is likely that you would prefer to use the CMake GUI if you use Visual Studio to build FLAC. It's in essence the same process as building using CLI.

Open cmake-gui. In the window select a source directory (the repository's root), a build directory (some other directory outside the repository). Then press button "Configure". CMake will ask you which build system you prefer. Choose that version of Visual Studio which you have on your system, choose whether you want to build for Win32 or x64. Press OK.

After CMake finishes you can change the configuration to your liking and if you change anything, run Configure again. With the "Generate" button, CMake creates Visual Studio files, which can be opened

---

from Visual Studio. With the button “Open Project” CMake will launch Visual Studio and open the generated solution. You can use the project files as usual but remember that they were generated by CMake. That means that your changes (e.g. some additional compile flags) will be lost when you run CMake next time.

CMake searches by default for OGG on your system and returns an error if it cannot find it. If you want to build OGG alongside FLAC, you can download the OGG sources and extract them in a subdirectory of the FLAC source directory with the name ogg (i.e. /path/to/flac-source/ogg) before running CMake. If you don’t want to build FLAC with OGG support, untick the box following WITH\_OGG flag in the list of variables in cmake-gui window and run “Configure” again.

If CMake fails to find MSVC compiler then running cmake-gui from MS Developer comand prompt should help.

## Building with GNU autotools

FLAC uses autoconf and libtool for configuring and building. To configure a build, open a command line/terminal and run `./configure` You can provide options to this command, which are listed by running `./configure --help`.

In case the configure script is not present (for example when building from git and not from a release tarball), it can be generated by running `./autogen.sh`. This may require a libtool development package though.

After configuration, build with `make`, verify the build with `make check` and install with `make install`. Installation might require administrator priviledged, i.e. `sudo make install`.

The ‘make check’ step is optional; omit it to skip all the tests, which can take about an hour to complete. Even though it will stop with an explicit message on any failure, it does print out a lot of stuff so you might want to capture the output to a file if you’re having a problem. Also, don’t run ‘make check’ as root because it confuses some of the tests.

Summarizing:

```
1 ./configure
2 make && make check
3 sudo make install
```

## Note to embedded developers

libFLAC has grown larger over time as more functionality has been included, but much of it may be unnecessary for a particular embedded implementation. Unused parts may be pruned by some simple

---

editing of `configure.ac` and `src/libFLAC/Makefile.am`; the following dependency graph shows which modules may be pruned without breaking things further down:

```
1 metadata.h
2     stream_decoder.h
3     format.h
4
5 stream_encoder.h
6     stream_decoder.h
7     format.h
8
9 stream_decoder.h
10    format.h
```

In other words, for pure decoding applications, both the stream encoder and metadata editing interfaces can be safely removed. Note that this is specific to building the libraries for embedded use. The command line tools do not provide such compartmentalization, and require a complete libFLAC build to function.

There is a section dedicated to embedded use in the libFLAC API HTML documentation (see [doc/html/api/index.html](http://doc/html/api/index.html)).

Also, there are several places in the libFLAC code with comments marked with “OPT:” where a `#define` can be changed to enable code that might be faster on a specific platform. Experimenting with these can yield faster binaries.