Building Bitcoin Core with CMake

Use cases: from simple to advanced

Berlin, 8th April 2024

Preface

We are going to learn how to build Bitcoin Core binaries using a new CMake-based build system instead of the current Autotools-based one.

This workshop is focused on ready-to-use practices for everyday usage by developers.

Back to Basics: Know Your Tools. Make

We got used to get our binaries built using the Make (GNU Make) tool:

```
cd depends
make -j 8
```

To adjust the build process to our needs, we can use variables, for example:

```
make -j 8 NO_QT=1
```

It works for the **`depends`** subdirectory because it already contains the **`Makefile`** file that is processed by the Make tool.

That is not the case for the root and **`src`** directories.

Back to Basics: Know Your Tools. Autotools

Here enters Autotools, which is a set of tools including Autoconf, Automake and Libtool. The full list of commands is as follows:

```
./autogen.sh
./configure
make -j 8
```

To adjust the build process to our needs, we can use configure options, for example:

```
./configure --without-gui
```

Meet CMake

The CMake tracking issue is #28607.

The full-fledged (but unreviewed and outdated) branch is available in PR25797.

The reviewed version is available in https://github.com/hebasto/bitcoin/tree/cmake-staging. It produces the same binaries as the Autotools-based build system does.

Installing CMake

On many systems, you may find that CMake is already installed or is available for installation with the standard system's package manager.

For example, on Debian/Ubuntu, run:

```
sudo apt install cmake
```

On Fedora:

```
sudo dnf install cmake
```

Then check CMake version:

```
cmake --version
```

You can find precompiled CMake binaries for many common architectures on https://cmake.org/download.

Directory Structure

There are two main directories CMake uses when building a project: the *source* directory and the *binary* directory.

The source directory is where the source code for the project is located. For developers, it is a root of their local git repository.

The binary directory is also referred to as the *build* directory and is where CMake will put the resulting object files, libraries, and executables. CMake will not write any files to the source directory, only to the binary directory.

Out-of-source builds, where the source and binary directories are different, are strongly encouraged. In-source builds where the source and binary directories are the same are supported but should be avoided if possible. Out-of-source builds make it very easy to maintain a clean source tree and allow quick removal of all of the files generated by a build. Having the build tree differ from the source tree also makes it easy to support having multiple builds of a single source tree. This is useful when you want to have multiple builds with different options but just one copy of the source code.

Generating a Project Buildsystem

To generate a project buildsystem on Linux or macOS, run the following commands:

```
mkdir build
cd build
cmake -S ..
```

To adjust the configuration, we can use CMake variables, for example:

```
cmake -S .. -D WITH_BDB=OFF
```

Using an additional **-LH** option will effectively display current CMake settings, which can then be changed with **-D** option.

Building a project

To generate a project buildsystem on Linux or macOS, run the following command in the binary directory:

```
cmake --build . -j 8
```

A specific target might be specified with the **`--target`** / **`-t`** option:

```
cmake --build . -j 8 -t bitcoind
```

Running tests

To run unit tests, run the following command in the binary directory:

```
ctest -j 8
```

To run functional tests:

```
./test/functional/test_runner.py -j 8
```

Choosing compiler

The CMake's way to designate a non-default compiler:

```
cmake -S .. -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++
```

Some essential compiler flags can be specified as well:

```
cmake -S .. -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER='clang++;-stdlib=libc++'
```

CMake considers values of the **CC** and **CXX** environment variables as well.

Modifying compiler and linker flags

For simplicity, we'll consider C++ compiler flags only.

CMake has a separated set of flags, which depends on the chosen build configuration:

```
${CMAKE_CXX_FLAGS} ${CMAKE_CXX_FLAGS_DEBUG}
```

The latter unusually specifies optimization and debugging options.

When building Bitcoin Core, it is recommended to use environment variables:

```
env CXXFLAGS="-Wno-error=return-type" cmake -S ..
```

Cross-compiling

In cross-compiling scenarios, a generated toolchain file must be specified during generating a project buildsystem. For example, for macOS (arm64), run the commands as follows:

```
make -C depends -j 16 HOST=arm64-apple-darwin
mkdir build-macos
cd build-macos
cmake -S .. --toolchain ../depends/arm64-apple-darwin/toolchain.cmake
cmake --build . -j 8
cmake --build . --target deploy
```

Advanced CMake: Generators

By default, CMake chooses a generator, which is responsible for writing the input files for a native build system, based on the build platform. For example, on Unix-like systems, including macOS, the "Unix Makefiles" generator is used by default.

However, other generators are available for the user, who might override the default generator with the **`-G`** option:

```
mkdir build-ninja
cd build-ninja
cmake -S .. -G "Ninja"
```

The following build and test command remain the same:

```
cmake --build . -j 8
ctest -j 8
```

as CMake hides generator-specific details.

NOTE. The **`ninja-build`** package is required to be installed.

Advanced CMake: Multi-Configuration Generators

Configurations determine specifications for a certain type of build, such as "Release" or "Debug". For single configuration generators like "Unix Makefiles" and "Ninja", the configuration is specified at configure time by the **CMAKE_BUILD_TYPE** variable:

```
cmake -S .. -D CMAKE_BUILD_TYPE=Debug
```

For multi-configuration generators like "Visual Studio", "Xcode", and "Ninja Multi-Config", the configuration is chosen by the user at build time:

```
cmake -S .. -G "Ninja Multi-Config"
cmake --build . -j 8 --config Debug
```

It allows to have binaries built for different configurations in the binary tree simultaneously. During testing, the user have to specify configuration as well:

```
ctest -j 8 -C Debug
```

Advanced CMake: IDE Generators

The "Visual Studio" and "Xcode" generators were mentioned previously. They support IDE-specific project files that might be opened in an IDE. For example:

```
cmake -S .. -G "Xcode"
```

creates the "**Bitcoin Core.xcodeproj**" project file.

On Windows, the following command

```
cmake -S .. --preset vs2022
```

generates the "**Bitcoin Core.sln**" solution file respectively.

Takeaways

CMake is an open-source multi-platform build system generator for software projects that allows users to specify build parameters in a simple and portable way.

CMake adoption is successful (for instance, KDE, Qt) and expected to grow.

In Bitcoin Core project, Cmake does everything that Autotools do, but:

- on a wider range of systems
- with more options (for instance, (a) presets; (b) the integrated support for the **`clang-tidy`** and **`include-what-you-use`** tools; (c) interactive dialogs etc)
- opens gates for a range of improvements in the long run (for example, (a) C++20 modules; (b) Qt 6 etc)