Building LLVM with CMake

Introduction

CMake is a cross-platform build-generator tool. CMake does not build the project, it generates the files needed by your build tool (GNU make, Visual Studio, etc.) for building LLVM.

If **you are a new contributor**, please start with the Getting Started with the LLVM System page. This page is geared for existing contributors moving from the legacy configure/make system.

If you are really anxious about getting a functional LLVM build, go to the Quick start section. If you are a CMake novice, start with Basic CMake usage and then go back to the Quick start section once you know what you are doing. The Options and variables section is a reference for customizing your build. If you already have experience with CMake, this is the recommended starting point.

This page is geared towards users of the LLVM CMake build. If you're looking for information about modifying the LLVM CMake build system you may want to see the CMake Primer page. It has a basic overview of the CMake language.

Quick start

We use here the command-line, non-interactive CMake interface.

1. Download and install CMake. Version 3.4.3 is the minimum required.
2. Open a shell. Your development tools must be reachable from this shell through the PATH environment variable.
3. Create a build directory. Building LLVM in the source directory is not supported. cd to this directory:

```
$ mkdir mybuilddir
$ cd mybuilddir
```

4. Execute this command in the shell replacing *path/to/llvm/source/root* with the path to the root of your LLVM source tree:

```
$ cmake path/to/llvm/source/root
```

   CMake will detect your development environment, perform a series of tests, and generate the files required for building LLVM. CMake will use default values for all build parameters. See the Options and variables section for a list of build parameters that you can modify.

   This can fail if CMake can't detect your toolset, or if it thinks that the environment is not sane enough. In this case, make sure that the toolset that you intend to use is the only one reachable from the shell, and that the shell itself is the correct one for your development environment. CMake will refuse to build MinGW makefiles if you have a POSIX shell reachable through the PATH environment variable, for instance. You can force CMake to use a given build tool; for instructions, see the Usage section, below.

5. After CMake has finished running, proceed to use IDE project files, or start the build from the build directory:

```
$ cmake --build .
```

   The `--build` option tells `cmake` to invoke the underlying build tool (`make`, `ninja`, `xcodebuild`, `msbuild`, etc.)

   The underlying build tool can be invoked directly, of course, but the `--build` option is portable.

6. After LLVM has finished building, install it from the build directory:

```
$ cmake --build . --target install
```

   The `--target` option with `install` parameter in addition to the `--build` option tells `cmake` to build the `install` target.

   It is possible to set a different install prefix at installation time by invoking the `cmake_install.cmake` script generated in the build directory:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/tmp/llvm -P cmake_install.cmake
```

Basic CMake usage

This section explains basic aspects of CMake which you may need in your day-to-day usage.

CMake comes with extensive documentation, in the form of html files, and as online help accessible via the `cmake` executable itself. Execute `cmake --help` for further help options.

CMake allows you to specify a build tool (e.g., GNU make, Visual Studio, or Xcode). If not specified on the command line, CMake tries to guess which build tool to use, based on your environment. Once it has identified your build tool, CMake uses the corresponding *Generator* to create files for your build tool (e.g., Makefiles or Visual Studio or Xcode project files). You can explicitly specify the generator with the command line option `-G "Name of the generator"`. To see a list of the available generators on your system, execute

```
$ cmake --help
```

This will list the generator names at the end of the help text.

Generators' names are case-sensitive, and may contain spaces. For this reason, you should enter them exactly as they are listed in the `cmake --help` output, in quotes. For example, to generate project files specifically for Visual Studio 12, you can execute:

```
$ cmake -G "Visual Studio 12" path/to/llvm/source/root
```

For a given development platform there can be more than one adequate generator. If you use Visual Studio, "NMake Makefiles" is a generator you can use for building with NMake. By default, CMake chooses the most specific generator supported by your development environment. If you want an alternative generator, you must tell this to CMake with the `-G` option.

Options and variables

Variables customize how the build will be generated. Options are boolean variables, with possible values ON/OFF. Options and variables are defined on the CMake command line like this:

```
$ cmake -DVARIABLE=value path/to/llvm/source
```

You can set a variable after the initial CMake invocation to change its value. You can also undefine a variable:

```
$ cmake -UVARIABLE path/to/llvm/source
```

Variables are stored in the CMake cache. This is a file named `CMakeCache.txt` stored at the root of your build directory that is generated by `cmake`. Editing it yourself is not recommended.

Variables are listed in the CMake cache and later in this document with the variable name and type separated by a colon. You can also specify the variable and type on the CMake command line:

```
$ cmake -DVARIABLE:TYPE=value path/to/llvm/source
```

Frequently-used CMake variables

Here are some of the CMake variables that are used often, along with a brief explanation and LLVM-specific notes. For full documentation, consult the CMake manual, or execute `cmake --help-variable VARIABLE_NAME`.

CMAKE_BUILD_TYPE:STRING

Sets the build type for `make`-based generators. Possible values are Release, Debug, RelWithDebInfo and MinSizeRel. If you are using an IDE such as Visual Studio, you should use the IDE settings to set the build type. Be aware that Release and RelWithDebInfo use different optimization levels on most platforms.

CMAKE_INSTALL_PREFIX:PATH

Path where LLVM will be installed if "make install" is invoked or the "install" target is built.

LLVM_LIBDIR_SUFFIX:STRING

Extra suffix to append to the directory where libraries are to be installed. On a 64-bit architecture, one could use `-DLLVM_LIBDIR_SUFFIX=64` to install libraries to `/usr/lib64`.

CMAKE_C_FLAGS:STRING

Extra flags to use when compiling C source files.

CMAKE_CXX_FLAGS:STRING

Extra flags to use when compiling C++ source files.

LLVM-specific variables

LLVM_TARGETS_TO_BUILD:STRING

Semicolon-separated list of targets to build, or *all* for building all targets. Case-sensitive. Defaults to *all*. Example: `-DLLVM_TARGETS_TO_BUILD="X86;PowerPC"`.

LLVM_BUILD_TOOLS:BOOL

Build LLVM tools. Defaults to ON. Targets for building each tool are generated in any case. You can build a tool separately by invoking its target. For example, you can build *llvm-as* with a Makefile-based system by executing *make llvm-as* at the root of your build directory.

LLVM_INCLUDE_TOOLS:BOOL

Generate build targets for the LLVM tools. Defaults to ON. You can use this option to disable the generation of build targets for the LLVM tools.

LLVM_INSTALL_BINUTILS_SYMLINKS:BOOL

Install symlinks from the binutils tool names to the corresponding LLVM tools. For example, ar will be symlinked to llvm-ar.

LLVM_INSTALL_CCTOOLS_SYMLINKS:BOOL

Install symliks from the cctools tool names to the corresponding LLVM tools. For example, lipo will be symlinked to llvm-lipo.

LLVM_BUILD_EXAMPLES:BOOL

Build LLVM examples. Defaults to OFF. Targets for building each example are generated in any case. See documentation for *LLVM_BUILD_TOOLS* above for more details.

LLVM_INCLUDE_EXAMPLES:BOOL

Generate build targets for the LLVM examples. Defaults to ON. You can use this option to disable the generation of build targets for the LLVM examples.

LLVM_BUILD_TESTS:BOOL

Build LLVM unit tests. Defaults to OFF. Targets for building each unit test are generated in any case. You can build a specific unit test using the targets defined under *unittests*, such as ADTTests, IRTests, SupportTests, etc. (Search for `add_llvm_unittest` in the subdirectories of *unittests* for a complete list of unit tests.) It is possible to build all unit tests with the target *UnitTests*.

LLVM_INCLUDE_TESTS:BOOL

Generate build targets for the LLVM unit tests. Defaults to ON. You can use this option to disable the generation of build targets for the LLVM unit tests.

LLVM_BUILD_BENCHMARKS:BOOL

Adds benchmarks to the list of default targets. Defaults to OFF.

LLVM_INCLUDE_BENCHMARKS:BOOL

Generate build targets for the LLVM benchmarks. Defaults to ON.

LLVM_APPEND_VC_REV:BOOL

Embed version control revision info (svn revision number or Git revision id). The version info is provided by the `LLVM_REVISION` macro in `llvm/include/llvm/Support/VCSRevision.h`. Developers using git who don't need revision info can disable this option to avoid re-linking most binaries after a branch switch. Defaults to ON.

LLVM_ENABLE_THREADS:BOOL

Build with threads support, if available. Defaults to ON.

LLVM_ENABLE_UNWIND_TABLES:BOOL

Enable unwind tables in the binary. Disabling unwind tables can reduce the size of the libraries. Defaults to ON.

LLVM_CXX_STD:STRING

Build with the specified C++ standard. Defaults to "c++11".

LLVM_ENABLE_ASSERTIONS:BOOL

Enables code assertions. Defaults to ON if and only if `CMAKE_BUILD_TYPE` is *Debug*.

LLVM_ENABLE_EH:BOOL

Build LLVM with exception-handling support. This is necessary if you wish to link against LLVM libraries and make use of C++ exceptions in your own code that need to propagate through LLVM code. Defaults to OFF.

LLVM_ENABLE_EXPENSIVE_CHECKS:BOOL

Enable additional time/memory expensive checking. Defaults to OFF.

LLVM_ENABLE_IDE:BOOL

Tell the build system that an IDE is being used. This in turn disables the creation of certain convenience build system targets, such as the various `install-*` and `check-*` targets, since IDEs don't always deal well with a large number of targets. This is usually autodetected, but it can be configured manually to explicitly control the generation of those targets. One scenario where a manual override may be desirable is when using Visual Studio 2017's CMake integration, which would not be detected as an IDE otherwise.

LLVM_ENABLE_PIC:BOOL

Add the `-fPIC` flag to the compiler command-line, if the compiler supports this flag. Some systems, like Windows, do not need this flag. Defaults to ON.

LLVM_ENABLE_RTTI:BOOL

Build LLVM with run-time type information. Defaults to OFF.

LLVM_ENABLE_WARNINGS:BOOL

Enable all compiler warnings. Defaults to ON.

LLVM_ENABLE_PEDANTIC:BOOL

Enable pedantic mode. This disables compiler-specific extensions, if possible. Defaults to ON.

LLVM_ENABLE_WERROR:BOOL

Stop and fail the build, if a compiler warning is triggered. Defaults to OFF.

LLVM_ABI_BREAKING_CHECKS:STRING

Used to decide if LLVM should be built with ABI breaking checks or not. Allowed values are *`WITH_ASSERTS`* (default), *`FORCE_ON`* and *`FORCE_OFF`*. *`WITH_ASSERTS`* turns on ABI breaking checks in an assertion enabled build. *`FORCE_ON`* (*`FORCE_OFF`*) turns them on (off) irrespective of whether normal (*`NDEBUG`*-based) assertions are enabled or not. A version of LLVM built with ABI breaking checks is not ABI compatible with a version built without it.

LLVM_BUILD_32_BITS:BOOL

Build 32-bit executables and libraries on 64-bit systems. This option is available only on some 64-bit Unix systems. Defaults to OFF.

LLVM_TARGET_ARCH:STRING

LLVM target to use for native code generation. This is required for JIT generation. It defaults to "host", meaning that it shall pick the architecture of the machine where LLVM is being built. If you are cross-compiling, set it to the target architecture name.

LLVM_TABLEGEN:STRING

Full path to a native TableGen executable (usually named `llvm-tblgen`). This is intended for cross-compiling: if the user sets this variable, no native TableGen will be created.

LLVM_LIT_ARGS:STRING

Arguments given to lit. `make check` and `make clang-test` are affected. By default, `'-sv --no-progress-bar'` on Visual C++ and Xcode, `'-sv'` on others.

LLVM_LIT_TOOLS_DIR:PATH

The path to GnuWin32 tools for tests. Valid on Windows host. Defaults to the empty string, in which case lit will look for tools needed for tests (e.g. `grep`, `sort`, etc.) in your %PATH%. If GnuWin32 is not in your %PATH%, then you can set this variable to the GnuWin32 directory so that lit can find tools needed for tests in that directory.

LLVM_ENABLE_FFI:BOOL

Indicates whether the LLVM Interpreter will be linked with the Foreign Function Interface library (libffi) in order to enable calling external functions. If the library or its headers are installed in a custom location, you can also set the variables FFI_INCLUDE_DIR and FFI_LIBRARY_DIR to the directories where ffi.h and libffi.so can be found, respectively. Defaults to OFF.

LLVM_EXTERNAL_{CLANG,LLD,POLLY}_SOURCE_DIR:PATH

These variables specify the path to the source directory for the external LLVM projects Clang, lld, and Polly, respectively, relative to the top-level source directory. If the in-tree subdirectory for an external project exists (e.g., llvm/tools/clang for Clang), then the corresponding variable will not be used. If the variable for an external project does not point to a valid path, then that project will not be built.

LLVM_ENABLE_PROJECTS:STRING

Semicolon-separated list of projects to build, or *all* for building all (clang, libcxx, libcxxabi, lldb, compiler-rt, lld, polly) projects. This flag assumes that projects are checked out side-by-side and not nested, i.e. clang needs to be in parallel of llvm instead of nested in *`llvm/tools`*. This feature allows to have one build for only LLVM and another for clang+llvm using the same source checkout.

LLVM_EXTERNAL_PROJECTS:STRING

Semicolon-separated list of additional external projects to build as part of llvm. For each project LLVM_EXTERNAL_<NAME>_SOURCE_DIR have to be specified with the path for the source code of the project. Example: `-DLLVM_EXTERNAL_PROJECTS="Foo;Bar" -DLLVM_EXTERNAL_FOO_SOURCE_DIR=/src/foo -DLLVM_EXTERNAL_BAR_SOURCE_DIR=/src/bar`.

LLVM_USE_OPROFILE:BOOL

Enable building OProfile JIT support. Defaults to OFF.

LLVM_PROFDATA_FILE:PATH

Path to a profdata file to pass into clang's -fprofile-instr-use flag. This can only be specified if you're building with clang.

LLVM_USE_INTEL_JITEVENTS:BOOL

Enable building support for Intel JIT Events API. Defaults to OFF.

LLVM_ENABLE_LIBPFM:BOOL

Enable building with libpfm to support hardware counter measurements in LLVM tools. Defaults to ON.

LLVM_USE_PERF:BOOL

Enable building support for Perf (linux profiling tool) JIT support. Defaults to OFF.

LLVM_ENABLE_ZLIB:BOOL

Enable building with zlib to support compression/uncompression in LLVM tools. Defaults to ON.

LLVM_ENABLE_DIA_SDK:BOOL

Enable building with MSVC DIA SDK for PDB debugging support. Available only with MSVC. Defaults to ON.

LLVM_USE_SANITIZER:STRING

Define the sanitizer used to build LLVM binaries and tests. Possible values are `Address`, `Memory`, `MemoryWithOrigins`, `Undefined`, `Thread`, and `Address;Undefined`. Defaults to empty string.

LLVM_ENABLE_LTO:STRING

Add `-flto` or `-flto=` flags to the compile and link command lines, enabling link-time optimization. Possible values are `Off`, `On`, `Thin` and `Full`. Defaults to OFF.

LLVM_USE_LINKER:STRING

Add `-fuse-ld={name}` to the link invocation. The possible value depend on your compiler, for clang the value can be an absolute path to your custom linker, otherwise clang will prefix the name with `ld.` and apply its usual search. For example to link LLVM with the Gold linker, cmake can be invoked with `-DLLVM_USE_LINKER=gold`.

LLVM_ENABLE_LIBCXX:BOOL

If the host compiler and linker supports the stdlib flag, -stdlib=libc++ is passed to invocations of both so that the project is built using libc++ instead of stdlibc++. Defaults to OFF.

LLVM_STATIC_LINK_CXX_STDLIB:BOOL

Statically link to the C++ standard library if possible. This uses the flag "-static-libstdc++", but a Clang host compiler will statically link to libc++ if used in conjuction with the **LLVM_ENABLE_LIBCXX** flag. Defaults to OFF.

LLVM_ENABLE_LLD:BOOL

This option is equivalent to *`-DLLVM_USE_LINKER=lld`*, except during a 2-stage build where a dependency is added from the first stage to the second ensuring that lld is built before stage2 begins.

LLVM_PARALLEL_COMPILE_JOBS:STRING

Define the maximum number of concurrent compilation jobs.

LLVM_PARALLEL_LINK_JOBS:STRING

Define the maximum number of concurrent link jobs.

LLVM_BUILD_DOCS:BOOL

Adds all *enabled* documentation targets (i.e. Doxgyen and Sphinx targets) as dependencies of the default build targets. This results in all of the (enabled) documentation targets being as part of a normal build. If the `install` target is run then this also enables all built documentation targets to be installed. Defaults to OFF. To enable a particular documentation target, see see LLVM_ENABLE_SPHINX and LLVM_ENABLE_DOXYGEN.

LLVM_ENABLE_DOXYGEN:BOOL

Enables the generation of browsable HTML documentation using doxygen. Defaults to OFF.

LLVM_ENABLE_DOXYGEN_QT_HELP:BOOL

Enables the generation of a Qt Compressed Help file. Defaults to OFF. This affects the make target `doxygen-llvm`. When enabled, apart from the normal HTML output generated by doxygen, this will produce a QCH file named `org.llvm.qch`. You can then load this file into Qt Creator. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN=ON`; otherwise this has no effect.

LLVM_DOXYGEN_QCH_FILENAME:STRING

The filename of the Qt Compressed Help file that will be generated when `-DLLVM_ENABLE_DOXYGEN=ON` and `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON` are given. Defaults to `org.llvm.qch`. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise it has no effect.

LLVM_DOXYGEN_QHP_NAMESPACE:STRING

Namespace under which the intermediate Qt Help Project file lives. See Qt Help Project for more information. Defaults to "org.llvm". This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise it has no effect.

LLVM_DOXYGEN_QHP_CUST_FILTER_NAME:STRING

See Qt Help Project for more information. Defaults to the CMake variable `${PACKAGE_STRING}` which is a combination of the package name and version string. This filter can then be used in Qt Creator to select only documentation from LLVM when browsing through all the help files that you might have loaded. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise it has no effect.

LLVM_DOXYGEN_QHELPGENERATOR_PATH:STRING

The path to the `qhelpgenerator` executable. Defaults to whatever CMake's `find_program()` can find. This option is only useful in combination with `-DLLVM_ENABLE_DOXYGEN_QT_HELP=ON`; otherwise it has no effect.

LLVM_DOXYGEN_SVG:BOOL

Uses .svg files instead of .png files for graphs in the Doxygen output. Defaults to OFF.

LLVM_INSTALL_DOXYGEN_HTML_DIR:STRING

The path to install Doxygen-generated HTML documentation to. This path can either be absolute or relative to the CMAKE_INSTALL_PREFIX. Defaults to *`share/doc/llvm/doxygen-html`*.

LLVM_ENABLE_SPHINX:BOOL

If specified, CMake will search for the `sphinx-build` executable and will make the `SPHINX_OUTPUT_HTML` and `SPHINX_OUTPUT_MAN` CMake options available. Defaults to OFF.

SPHINX_EXECUTABLE:STRING

The path to the `sphinx-build` executable detected by CMake. For installation instructions, see http://www.sphinx-doc.org/en/latest/install.html

SPHINX_OUTPUT_HTML:BOOL

If enabled (and `LLVM_ENABLE_SPHINX` is enabled) then the targets for building the documentation as html are added (but not built by default unless `LLVM_BUILD_DOCS` is enabled). There is a target for each project in the source tree that uses sphinx (e.g. `docs-llvm-html`, `docs-clang-html` and `docs-lld-html`). Defaults to ON.

SPHINX_OUTPUT_MAN:BOOL

If enabled (and `LLVM_ENABLE_SPHINX` is enabled) the targets for building the man pages are added (but not built by default unless `LLVM_BUILD_DOCS` is enabled). Currently the only target added is `docs-llvm-man`. Defaults to ON.

SPHINX_WARNINGS_AS_ERRORS:BOOL

If enabled then sphinx documentation warnings will be treated as errors. Defaults to ON.

LLVM_INSTALL_SPHINX_HTML_DIR:STRING

The path to install Sphinx-generated HTML documentation to. This path can either be absolute or relative to the CMAKE_INSTALL_PREFIX. Defaults to *`share/doc/llvm/html`*.

LLVM_INSTALL_OCAMLDOC_HTML_DIR:STRING

The path to install OCamldoc-generated HTML documentation to. This path can either be absolute or relative to the CMAKE_INSTALL_PREFIX. Defaults to *`share/doc/llvm/ocaml-html`*.

LLVM_CREATE_XCODE_TOOLCHAIN:BOOL

macOS Only: If enabled CMake will generate a target named 'install-xcode-toolchain'. This target will create a directory at $CMAKE_INSTALL_PREFIX/Toolchains containing an xctoolchain directory which can be used to override the default system tools.

LLVM_BUILD_LLVM_DYLIB:BOOL

If enabled, the target for building the libLLVM shared library is added. This library contains all of LLVM's components in a single shared library. Defaults to OFF. This cannot be used in conjunction with BUILD_SHARED_LIBS. Tools will only be linked to the libLLVM shared library if LLVM_LINK_LLVM_DYLIB is also ON. The components in the library can be customised by setting LLVM_DYLIB_COMPONENTS to a list of the desired components.

LLVM_LINK_LLVM_DYLIB:BOOL

If enabled, tools will be linked with the libLLVM shared library. Defaults to OFF. Setting LLVM_LINK_LLVM_DYLIB to ON also sets LLVM_BUILD_LLVM_DYLIB to ON.

BUILD_SHARED_LIBS:BOOL

Flag indicating if each LLVM component (e.g. Support) is built as a shared library (ON) or as a static library (OFF). Its default value is OFF. On Windows, shared libraries may be used when building with MinGW, including mingw-w64, but not when building with the Microsoft toolchain.

> **Note**
>
> BUILD_SHARED_LIBS is only recommended for use by LLVM developers. If you want to build LLVM as a shared library, you should use the `LLVM_BUILD_LLVM_DYLIB` option.

LLVM_OPTIMIZED_TABLEGEN:BOOL

If enabled and building a debug or asserts build the CMake build system will generate a Release build tree to build a fully optimized tablegen for use during the build. Enabling this option can significantly speed up build times especially when building LLVM in Debug configurations.

LLVM_REVERSE_ITERATION:BOOL

If enabled, all supported unordered llvm containers would be iterated in reverse order. This is useful for uncovering non-determinism caused by iteration of unordered containers.

LLVM_BUILD_INSTRUMENTED_COVERAGE:BOOL

If enabled, source-based code coverage instrumentation is enabled while building llvm.

LLVM_CCACHE_BUILD:BOOL

If enabled and the `ccache` program is available, then LLVM will be built using `ccache` to speed up rebuilds of LLVM and its components. Defaults to OFF. The size and location of the cache main-

tained by `ccache` can be adjusted via the LLVM_CCACHE_MAXSIZE and LLVM_CCACHE_DIR options, which are passed to the CCACHE_MAXSIZE and CCACHE_DIR environment variables, respectively.

LLVM_FORCE_USE_OLD_TOOLCHAIN:BOOL

If enabled, the compiler and standard library versions won't be checked. LLVM may not compile at all, or might fail at runtime due to known bugs in these toolchains.

LLVM_TEMPORARILY_ALLOW_OLD_TOOLCHAIN:BOOL

If enabled, the compiler version check will only warn when using a toolchain which is about to be deprecated, instead of emitting an error.

LLVM_USE_NEWPM:BOOL

If enabled, use the experimental new pass manager.

LLVM_ENABLE_BINDINGS:BOOL

If disabled, do not try to build the OCaml and go bindings.

LLVM_ENABLE_Z3_SOLVER:BOOL

If enabled, the Z3 constraint solver is activated for the Clang static analyzer. A recent version of the z3 library needs to be available on the system.

CMake Caches

Recently LLVM and Clang have been adding some more complicated build system features. Utilizing these new features often involves a complicated chain of CMake variables passed on the command line. Clang provides a collection of CMake cache scripts to make these features more approachable.

CMake cache files are utilized using CMake's -C flag:

```
$ cmake -C <path to cache file> <path to sources>
```

CMake cache scripts are processed in an isolated scope, only cached variables remain set when the main configuration runs. CMake cached variables do not reset variables that are already set unless the FORCE option is specified.

A few notes about CMake Caches:

- Order of command line arguments is important
  - -D arguments specified before -C are set before the cache is processed and can be read inside the cache file
  - -D arguments specified after -C are set after the cache is processed and are unset inside the cache file
- All -D arguments will override cache file settings
- CMAKE_TOOLCHAIN_FILE is evaluated after both the cache file and the command line arguments
- It is recommended that all -D options should be specified *before* -C

For more information about some of the advanced build configurations supported via Cache files see Advanced Build Configurations.

Executing the Tests

Testing is performed when the *check-all* target is built. For instance, if you are using Makefiles, execute this command in the root of your build directory:

```
$ make check-all
```

On Visual Studio, you may run tests by building the project “check-all”. For more information about testing, see the LLVM Testing Infrastructure Guide.

Cross compiling

See this wiki page for generic instructions on how to cross-compile with CMake. It goes into detailed explanations and may seem daunting, but it is not. On the wiki page there are several examples including toolchain files. Go directly to this section for a quick solution.

Also see the LLVM-specific variables section for variables used when cross-compiling.

Embedding LLVM in your project

From LLVM 3.5 onwards the CMake build system exports LLVM libraries as importable CMake targets. This means that clients of LLVM can now reliably use CMake to develop their own LLVM-based projects against an installed version of LLVM regardless of how it was built.

Here is a simple example of a CMakeLists.txt file that imports the LLVM libraries and uses them to build a simple application `simple-tool`.

```
cmake_minimum_required(VERSION 3.4.3)
project(SimpleProject)

find_package(LLVM REQUIRED CONFIG)

message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")

# Set your project compile flags.
# E.g. if using the C++ header files
# you will need to enable C++11 support
# for your compiler.

include_directories(${LLVM_INCLUDE_DIRS})
add_definitions(${LLVM_DEFINITIONS})

# Now build our tools
add_executable(simple-tool tool.cpp)

# Find the libraries that correspond to the LLVM components
# that we wish to use
llvm_map_components_to_libnames(llvm_libs support core irreader)

# Link against LLVM libraries
target_link_libraries(simple-tool ${llvm_libs})
```

The `find_package(...)` directive when used in CONFIG mode (as in the above example) will look for the `LLVMConfig.cmake` file in various locations (see cmake manual for details). It creates a `LLVM_DIR` cache entry to save the directory where `LLVMConfig.cmake` is found or allows the user to specify the directory (e.g. by passing `-DLLVM_DIR=/usr/lib/cmake/llvm` to the `cmake` command or by setting it directly in `ccmake` or `cmake-gui`).

This file is available in two different locations.

- `<INSTALL_PREFIX>/lib/cmake/llvm/LLVMConfig.cmake` where `<INSTALL_PREFIX>` is the install prefix of an installed version of LLVM. On Linux typically this is `/usr/lib/cmake/llvm/LLVMConfig.cmake`.
- `<LLVM_BUILD_ROOT>/lib/cmake/llvm/LLVMConfig.cmake` where `<LLVM_BUILD_ROOT>` is the root of the LLVM build tree. **Note: this is only available when building LLVM with CMake.**

If LLVM is installed in your operating system's normal installation prefix (e.g. on Linux this is usually `/usr/`) `find_package(LLVM ...)` will automatically find LLVM if it is installed correctly. If LLVM is not installed or you wish to build directly against the LLVM build tree you can use `LLVM_DIR` as previously mentioned.

The `LLVMConfig.cmake` file sets various useful variables. Notable variables include

`LLVM_CMAKE_DIR`
: The path to the LLVM CMake directory (i.e. the directory containing LLVMConfig.cmake).

`LLVM_DEFINITIONS`
: A list of preprocessor defines that should be used when building against LLVM.

`LLVM_ENABLE_ASSERTIONS`
: This is set to ON if LLVM was built with assertions, otherwise OFF.

`LLVM_ENABLE_EH`
: This is set to ON if LLVM was built with exception handling (EH) enabled, otherwise OFF.

`LLVM_ENABLE_RTTI`
: This is set to ON if LLVM was built with run time type information (RTTI), otherwise OFF.

`LLVM_INCLUDE_DIRS`
: A list of include paths to directories containing LLVM header files.

`LLVM_PACKAGE_VERSION`
: The LLVM version. This string can be used with CMake conditionals, e.g., `if (${LLVM_PACKAGE_VERSION} VERSION_LESS "3.5")`.

`LLVM_TOOLS_BINARY_DIR`
: The path to the directory containing the LLVM tools (e.g. `llvm-as`).

Notice that in the above example we link `simple-tool` against several LLVM libraries. The list of libraries is determined by using the `llvm_map_components_to_libnames()` CMake function. For a list of available components look at the output of running `llvm-config --components`.

Note that for LLVM < 3.5 `llvm_map_components_to_libraries()` was used instead of `llvm_map_components_to_libnames()`. This is now deprecated and will be removed in a future version of LLVM.

Developing LLVM passes out of source

It is possible to develop LLVM passes out of LLVM's source tree (i.e. against an installed or built LLVM). An example of a project layout is provided below.

```
<project dir>/
    |
    CMakeLists.txt
    <pass name>/
        |
        CMakeLists.txt
        Pass.cpp
        ...
```

Contents of `<project dir>/CMakeLists.txt`:

```
find_package(LLVM REQUIRED CONFIG)
```

```
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})

add_subdirectory(<pass name>)
```

Contents of `<project dir>/<pass name>/CMakeLists.txt`:

```
add_library(LLVMPassname MODULE Pass.cpp)
```

Note if you intend for this pass to be merged into the LLVM source tree at some point in the future it might make more sense to use LLVM's internal `add_llvm_library` function with the MODULE argument instead by...

Adding the following to `<project dir>/CMakeLists.txt` (after `find_package(LLVM ...)`)

```
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(AddLLVM)
```

And then changing `<project dir>/<pass name>/CMakeLists.txt` to

```
add_llvm_library(LLVMPassname MODULE
  Pass.cpp
  )
```

When you are done developing your pass, you may wish to integrate it into the LLVM source tree. You can achieve it in two easy steps:

1. Copying `<pass name>` folder into `<LLVM root>/lib/Transform` directory.
2. Adding `add_subdirectory(<pass name>)` line into `<LLVM root>/lib/Transform/CMakeLists.txt`.

Compiler/Platform-specific topics

Notes for specific compilers and/or platforms.

Microsoft Visual C++

LLVM_COMPILER_JOBS:STRING

Specifies the maximum number of parallel compiler jobs to use per project when building with msbuild or Visual Studio. Only supported for the Visual Studio 2010 CMake generator. 0 means use all processors. Default is 0.