How To Cross-Compile Clang/LLVM using Clang/LLVM

Introduction ¶

This document contains information about building LLVM and Clang on host machine, targeting another platform.

For more information on how to use Clang as a cross-compiler, please check http://clang.llvm.org/docs/CrossCompilation.html.

TODO: Add MIPS and other platforms to this document.

Cross-Compiling from x86_64 to ARM

In this use case, we'll be using CMake and Ninja, on a Debian-based Linux system, cross-compiling from an x86_64 host (most Intel and AMD chips nowadays) to a hard-float ARM target (most ARM targets nowadays).

The packages you'll need are:

- cmake
- ninja-build (from backports in Ubuntu)
- gcc-4.7-arm-linux-gnueabihf
- gcc-4.7-multilib-arm-linux-gnueabihf
- binutils-arm-linux-gnueabihf
- libgcc1-armhf-cross
- libsfgcc1-armhf-cross
- libstdc++6-armhf-cross
- libstdc++6-4.7-dev-armhf-cross

Configuring CMake

For more information on how to configure CMake for LLVM/Clang, see *Building LLVM with CMake*.

The CMake options you need to add are:

- -DCMAKE_CROSSCOMPILING=True
- -DCMAKE_INSTALL_PREFIX=<install-dir>
- -DLLVM_TABLEGEN=<path-to-host-bin>/llvm-tblgen
- -DCLANG_TABLEGEN=<path-to-host-bin>/clang-tblgen
- -DLLVM_DEFAULT_TARGET_TRIPLE=arm-linux-gnueabihf
- -DLLVM_TARGET_ARCH=ARM
- -DLLVM_TARGETS_TO_BUILD=ARM

If you're compiling with GCC, you can use architecture options for your target, and the compiler driver will detect everything that it needs:

- -DCMAKE_CXX_FLAGS='-march=armv7-a -mcpu=cortex-a9 -mfloat-abi=hard'

However, if you're using Clang, the driver might not be up-to-date with your specific Linux distribution, version or GCC layout, so you'll need to fudge.

In addition to the ones above, you'll also need:

- '-target arm-linux-gnueabihf' or whatever is the triple of your cross GCC.
- '--sysroot=/usr/arm-linux-gnueabihf', '--sysroot=/opt/gcc/arm-linux-gnueabihf' or whatever is the location of your GCC's sysroot (where /lib, /bin etc are).
- Appropriate use of -I and -L, depending on how the cross GCC is installed, and where are the libraries and headers.

The TableGen options are required to compile it with the host compiler, so you'll need to compile LLVM (or at least llvm-tblgen) to your host platform before you start. The CXX flags define the target, cpu (which in this case defaults to fpu=VFP3 with NEON), and forcing the hard-float ABI. If you're using Clang as a cross-compiler, you will *also* have to set --sysroot to make sure it picks the correct linker.

When using Clang, it's important that you choose the triple to be *identical* to the GCC triple and the sysroot. This will make it easier for Clang to find the correct tools and include headers. But that won't mean all headers and libraries will be found. You'll still need to use -I and -L to locate those extra ones, depending on your distribution.

Most of the time, what you want is to have a native compiler to the platform itself, but not others. So there's rarely a point in compiling all back-ends. For that reason, you should also set the TARGETS_TO_BUILD to only build the back-end you're targeting to.

You must set the CMAKE_INSTALL_PREFIX, otherwise a ninja install will copy ARM binaries to your root filesystem, which is not what you want.

Hacks

There are some bugs in current LLVM, which require some fiddling before running CMake:

1. If you're using Clang as the cross-compiler, there is a problem in the LLVM ARM back-end that is producing absolute relocations on position-independent code (R_ARM_THM_MOVW_ABS_NC), so for now, you should disable PIC:

   ```
   -DLLVM_ENABLE_PIC=False
   ```

   This is not a problem, since Clang/LLVM libraries are statically linked anyway, it shouldn't affect much.

2. The ARM libraries won't be installed in your system. But the CMake prepare step, which checks for dependencies, will check the *host* libraries, not the *target* ones. Below there's a list of some dependencies, but your project could have more, or this document could be outdated. You'll see the errors while linking as an indication of that.

   Debian based distros have a way to add multiarch, which adds a new architecture and allows you to install packages for those systems. See https://wiki.debian.org/Multiarch/HOWTO for more info.

   But not all distros will have that, and possibly not an easy way to install them in any anyway, so you'll have to build/download them separately.

   A quick way of getting the libraries is to download them from a distribution repository, like Debian (http://packages.debian.org/jessie/), and download the missing libraries. Note that the libXXX will have the shared objects (.so) and the libXXX-dev will give you the headers and the static (.a) library. Just in case, download both.

   The ones you need for ARM are: libtinfo, zlib1g, libxml2 and liblzma. In the Debian repository you'll find downloads for all architectures.

   After you download and unpack all .deb packages, copy all .so and .a to a directory, make the appropriate symbolic links (if necessary), and add the relevant -L and -I paths to -DCMAKE_CXX_FLAGS above.

Running CMake and Building

Finally, if you're using your platform compiler, run:

```
$ cmake -G Ninja <source-dir> <options above>
```

If you're using Clang as the cross-compiler, run:

```
$ CC='clang' CXX='clang++' cmake -G Ninja <source-dir> <options above>
```

If you have clang/clang++ on the path, it should just work, and special Ninja files will be created in the build directory. I strongly suggest you to run cmake on a separate build directory, *not* inside the source tree.

To build, simply type:

```
$ ninja
```

It should automatically find out how many cores you have, what are the rules that needs building and will build the whole thing.

You can't run ninja check-all on this tree because the created binaries are targeted to ARM, not x86_64.

Installing and Using

After the LLVM/Clang has built successfully, you should install it via:

```
$ ninja install
```

which will create a sysroot on the install-dir. You can then tar that directory into a binary with the full triple name (for easy identification), like:

```
$ ln -sf <install-dir> arm-linux-gnueabihf-clang
$ tar zchf arm-linux-gnueabihf-clang.tar.gz arm-linux-gnueabihf-clang
```

If you copy that tarball to your target board, you'll be able to use it for running the test-suite, for example. Follow the guidelines at http://llvm.org/docs/lnt/quickstart.html, unpack the tarball in the test directory, and use options:

```
$ ./sandbox/bin/python sandbox/bin/lnt runtest nt \
   --sandbox sandbox \
   --test-suite `pwd`/test-suite \
   --cc `pwd`/arm-linux-gnueabihf-clang/bin/clang \
   --cxx `pwd`/arm-linux-gnueabihf-clang/bin/clang++
```

Remember to add the -jN options to lnt to the number of CPUs on your board. Also, the path to your clang has to be absolute, so you'll need the *pwd* trick above.